On-line-поддержка книги

ВЛАДИМИР ДАВЫДОВ

Visual C++

Разработка Windows-приложений с помощью MFC и API-функций

Низкоуровневая и высокоуровневая технологии программирования

Редактор ресурсов. Элементы интерфейса

Динамически подключаемые библиотеки

Мастера. Каркасы Windows-приложений

Более 35 демонстрационных примеров, 280 вопросов и упражнений

+ CD

Факультет переподготовки специалистов Санкт-Петербургского государственного политехнического университета СПбГПУ

ФПС

http://www.avalon.ru
mailto: info@avalon.ru
+7(812) 7030202

AVALON.RU — СОВЕТУЮТ ПРОФЕССИОНАЛЫ

Владимир Давыдов

Visual C++

Разработка Windows-приложений с помощью MFC и API-функций

Санкт-Петербург
«БХВ-Петербург»
2008

УДК 681.3.068
ББК 32.973.26-018.1
Д13

Давыдов В. Г.

Д13 Visual C++. Разработка Windows-приложений с помощью MFC и API-функций. — СПб.: БХВ-Петербург, 2008. — 576 с.: ил. + CD-ROM

ISBN 978-5-9775-0157-6

Рассмотрены низкоуровневая (API-функции) и высокоуровневая (библиотека классов MFC) технологии прикладного программирования в среде в Microsoft Visual Studio C++ .NET для ОС Windows. Подробно описаны дочерние окна, редактор ресурсов, меню, панели инструментов, строка статуса, диалоговые окна и более 15 самых популярных управляющих элементов для них, динамические подключаемые библиотеки и мастера. Материал сопровождается демонстрационными примерами, вопросами и упражнениями для самопроверки с ответами, тестами и заданиями для курсового проектирования, которые также помещены на прилагаемом компакт-диске.

Для студентов, преподавателей технических вузов и программистов

УДК 681.3.068
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Андрей Смышляев*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 25.12.07.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 46,44.
Тираж 1500 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0157-6

© Давыдов В. Г., 2008
© Оформление, издательство "БХВ-Петербург", 2008

Оглавление

Предисловие

Эта книга, наряду с учебным пособием "Технологии программирования. C++" [1], изданным ранее в "БХВ-Петербург", обеспечивает курс "Технологии программирования" и соответствует разработанной, с участием автора, примерной программе этого курса, рекомендованной Министерством образования для подготовки бакалавров, магистров и инженеров по направлению 220200 "Автоматизация и управление" (региональная часть государственного образовательного стандарта). Поскольку курс "Технологии программирования" является продолжением курса "Программирование и основы алгоритмизации", то и книга, которую вы сейчас читаете, является продолжением учебного пособия по курсу "Программирование и основы алгоритмизации" [2], вышедшего в издательстве "Высшая школа". Пособие ориентировано на студентов, но может быть полезным и преподавателям высших учебных заведений, а также программистам, создающим программные продукты с использованием языка C++.

В книге рассмотрены вопросы, связанные с разработкой оконных приложений для операционных систем Windows с использованием низкоуровневых средств программирования: набор базовых интерфейсов программирования приложений для Windows — Application Programming Interface (API), и набор из средств разработки, утилит и документации, позволяющий создавать приложения для платформы Windows — Software Development Kit (SDK или Platform SDK). А также высокоуровневых средств разработки на основе библиотеки классов фирмы Microsoft — Microsoft Foundation Classes (MFC). Демонстрационные примеры в первых главах части книги выполнены в двух вариантах, с использованием Windows API и MFC. Это сделано для того, чтобы читатель мог сопоставить разные средства достижения одного и того же результата и детально разобраться в особенностях разработки оконных приложений. Все иллюстрирующие программы хорошо структурированы — в них последовательно выполнена файловая, функциональная и объектно-ориентированная декомпозиция.

В книге, применительно к IDE (Integrated Development Environment, интегрированная среда разработки) Microsoft Visual Studio 2005, последовательно рассмотрены следующие основные вопросы:

1. Базовые концепции программирования оконных приложений. Общая характеристика библиотеки классов MFC.
2. Создание простейших Windows-приложений. Контекст, вывод в главное окно текстовой и графической информации.
3. Создание Windows-приложений с дочерними окнами и кнопками. Обработка кнопок.
4. Редактор ресурсов. Создание Windows-приложений с поддержкой меню. Меню, команды, акселераторы, "горячие" клавиши. Таблица строк.
5. Редактор ресурсов. Создание приложений на базе MFC с панелями инструментов и строкой статуса. Всплывающие подсказки, сообщения для строки статуса, конфигурирование панелей инструментов и строки статуса.
6. Редактор ресурсов. Создание приложений на базе MFC с диалоговыми окнами. Стандартные и пользовательские диалоговые окна. Модальные и немодальные диалоговые окна. Использование управляющих элементов в диалоговых и обычных окнах (однострочный редактор, радиокнопки).
7. Редактор ресурсов. Создание приложений на базе MFC с диалоговыми окнами. Использование управляющих элементов в диалоговых окнах (группировка управляющих элементов, отмечаемые и растровые кнопки, полосы прокрутки и т. п.).
8. Создание и использование динамически подключаемых библиотек для приложений на базе MFC.
9. Создание каркаса приложения на базе MFC с помощью мастера MFC Application Wizard. Русификация полученного каркаса.
10. Модификация каркаса приложения на базе MFC, полученного с помощью мастера MFC Application Wizard. Добавление меню и панелей инструментов. Загрузка и выгрузка файлов в дочернее окно (или дочерние окна). Интеграция приложения в полученный каркас. Добавление новых тем в справочную систему и их русификация.

В приложениях к книге приведены следующие полезные сведения:

- ответы к вопросам для самопроверки;
- варианты тестов и заданий на курсовое проектирование;
- методика создания и отладки оконного приложения в IDE Microsoft Visual Studio 2005.

Для удобства читателей книга содержит более 120 вариантов тестовых контрольных заданий по основным разделам, 30 вариантов заданий на курсовое

проектирование и возможный пример тестовых экзаменационных вопросов. В книгу включено более 210 вопросов для самопроверки, снабженных ответами, и более 70 упражнений для самопроверки, что позволяет использовать ее и для *самостоятельного* изучения материала. Прилагаемый компакт-диск содержит файл с вариантами контрольных тестовых заданий по основным разделам и заданий на курсовое проектирование, а также полные исходные тексты демонстрационных программ, имеющихся в книге (36 демонстрационных примеров). Восемь демонстрационных примеров разработано совместно с проф. Лекаревым М. Ф. и часть их опубликована [7]. Из сказанного со всей очевидностью следует, что книга поддерживает не только лекционную часть учебного курса, но и полностью поддерживает практические занятия.

Используемые обозначения

Исходные коды программ и результаты их работы, приводимые в книге, для удобства читателей печатаются с использованием моноширинного шрифта `Courier New`. Названия окон, полей окон, меню, команд, ресурсов, клавиш и кнопок в тексте книги выделяются **полужирным шрифтом**.

Курсивом в тексте выделяются определяющие вхождения новых понятий, а также отдельные слова или выражения, на которые следует обратить внимание.

Имена файлов и их расширения пишутся без кавычек и без выделения.

Кроме шрифтовых выделений, используется три типа специальных абзацев: советы, замечания и примечания.

Совет

Всегда стремитесь использовать масштабируемые шрифты.

Замечание

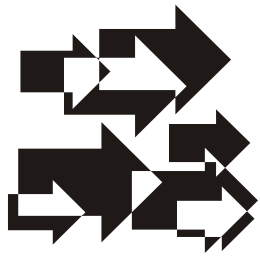
Чтобы вы могли эффективно общаться и легко понимать друг друга, внимательно изучите все описанные далее термины и понятия.

Примечание

Наряду с перечисленными компонентами окна имеются и другие компоненты, такие как: панели инструментов с кнопками, строка статуса, управляющие элементы (всевозможные кнопки, элементы редактирования, различные списки, индикаторы прогресса и т. п.), которые будут рассмотрены далее.

Ваши отзывы о книге, конструктивные замечания и критику направляйте по адресу: **davydov@aivt.ftk.spbstu.ru**.

ГЛАВА 1



Базовые концепции программирования [3]

Особенность создания Windows-приложений, т. е. приложений, предназначенных для работы под управлением операционной системы (ОС) Windows (далее Windows), заключается в том, что здесь применяются специальные методики программирования и используется своя терминология, которую можно разделить на две большие категории:

- терминология, связанная с пользовательским интерфейсом (меню, диалоговые окна, пиктограммы и т. д.);
- терминология, относящаяся непосредственно к программированию (сообщения, вызовы функций и т. д.).

Примечание

Чтобы вы могли эффективно общаться и легко понимать друг друга, внимательно изучите все описанные далее термины и понятия.

1.1. Что представляет собой окно?

Окно — это специальная прямоугольная область экрана. Все элементы окна, его размер и внешний вид контролируются открывающей его программой. Каждый щелчок мышью, выполняемый пользователем на каком-либо элементе окна, вызывает ответные действия приложения. Многозадачность в Windows заключается, в частности, в возможности одновременного открытия окон нескольких приложений или же нескольких окон одного приложения. Активизируя с помощью мыши или клавиатуры то или иное окно, пользователь дает системе понять, что последующие команды и данные следует направлять именно этому окну.

1.2. Компоненты окна

Стандартный внешний вид окон Windows-приложений и предсказуемость работы различных их *компонентов* позволяют пользователям чувствовать себя уверенно с новыми приложениями и легко разбираться в принципах их работы. Основные компоненты любого окна показаны на рис. 1.1.

Рис. 1.1. Основные компоненты окна

1.2.1. Рамка

Обычно каждое окно заключается в небольшую *рамку*. Новичку может показаться, что функции рамки сводятся только к отделению окна от остальных частей экрана. Но в действительности это не так. Рамка окна, как правило, является и средством *масштабирования*. Размер окна приложения при его соответствующем стиле можно изменить. Для этого достаточно поместить указатель мыши на рамку и перетащить его, удерживая нажатой левую кнопку мыши.

1.2.2. Строка заголовка

Имя приложения, которому принадлежит открытое окно, отображается в *строке заголовка*, в верхней части окна. Строка заголовка является обяза-

тельным элементом всех окон приложений и позволяет пользователю легко определить, какому приложению принадлежит конкретное окно, если в системе запущено одновременно несколько приложений. Строка заголовка активного окна выделяется альтернативным цветом, чтобы активное окно легко можно было отличить от неактивных окон.

1.2.3. Значок (пиктограмма) приложения

Другим обязательным элементом любого окна является расположенный в его левом верхнем углу *значок приложения*. Этот значок обычно представляет собой маленький логотип приложения. Щелчок на значке приводит к открытию системного меню.

1.2.4. Системное меню

В *системном меню* представлены стандартные команды управления окном. Например, для локализованной, русскоязычной ОС: **Восстановить**, **Переместить**, **Размер**, **Свернуть**, **Развернуть** и **Закрыть** (рис. 1.2).

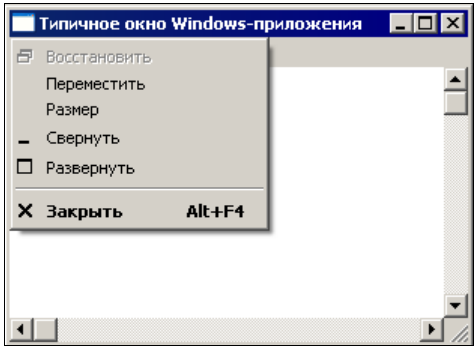


Рис. 1.2. Системное меню

1.2.5. Кнопка свертывания

В правом верхнем углу большинства окон приложений имеются три кнопки. Крайняя левая из них предназначена для *свертывания окна* в пиктограмму на панели задач. Такой же результат можно получить с помощью команды **Свернуть** системного меню.

1.2.6. Кнопка развертывания/восстановления

Средняя кнопка в правом верхнем углу либо *разворачивает окно* на весь экран, либо *восстанавливает* его прежние размеры, если окно уже развернуто.

Такой же результат можно получить с помощью команд **Развернуть** и **Восстановить** системного меню.

1.2.7. Кнопка закрытия

Крайняя правая кнопка в правом верхнем углу предназначена для *закрытия приложения*. После закрытия окна активным автоматически становится окно следующего приложения. Закрыть окно можно также с помощью команды **Закрыть** системного меню.

1.2.8. Вертикальная полоса прокрутки

В некоторых случаях окно приложения может содержать *вертикальную полосу прокрутки*, которая располагается по правому краю окна. В верхней и нижней частях полосы находятся кнопки со стрелками, направленными вверх и вниз соответственно. Вдоль самой полосы располагается бегунок. Положение бегунка показывает, какая часть окна или документа отображается в данный момент на экране. Перетаскивая бегунок с помощью мыши или клавиатуры, можно выбрать нужную часть многостраничного документа. Щелчок мышью на кнопке со стрелкой приведет к смещению содержимого окна на одну строку вверх или вниз, а щелчок на свободном пространстве выше или ниже бегунка — на одну экранную страницу вверх или вниз.

1.2.9. Горизонтальная полоса прокрутки

Окно может быть также оснащено *горизонтальной полосой прокрутки*, которая располагается по нижнему краю окна и работает аналогично вертикальной полосе прокрутки. Горизонтальная полоса прокрутки предназначена для выведения на экран частей документов, состоящих из большого числа столбцов. Щелчок мышью на кнопках со стрелками приведет к смещению содержимого окна на один столбец влево или вправо. Щелчок на областях между кнопками со стрелками и бегунком смещает изображение на одну экранную страницу влево или вправо.

1.2.10. Строка меню

В большинстве приложений, под строкой заголовка, находится *строка меню*, содержащая наборы команд и опций программы. Обычно для выбора команд меню используется мышь, но эти действия можно выполнить и с помощью клавиатуры. Каждому элементу меню, как правило, соответствует *клавиша быстрого вызова* ("горячая" клавиша), выделенная подчеркиванием в названии элемента. Чтобы вызвать данный элемент, нужно последовательно нажать клавишу <Alt> и соответствующую клавишу быстрого вызова. Так, по-

следовательное нажатие клавиш <Alt> и <F> для окна, представленного на рис. 1.1, открывает меню **File** (Файл). С помощью встроенного в IDE редактора ресурсов вы можете создавать собственные меню.

1.2.11. Рабочая область

Рабочая область обычно занимает большую часть окна. Именно в эту область программа выводит результаты своей работы.

Примечание

Наряду с перечисленными компонентами окна имеются и другие компоненты, такие как панели инструментов с кнопками, строка статуса, управляющие элементы (всевозможные кнопки, элементы редактирования, различные списки, индикаторы прогресса и т. п.), которые будут рассмотрены далее.

1.3. Классы окон

Чтобы два окна выглядели и работали совершенно одинаково, они оба должны базироваться на общем *классе окна*. В приложениях класс окна регистрируется программой в процессе инициализации. Зарегистрированный класс становится доступным для всех программ, запущенных в данный момент в системе. Как мы увидим далее, в случае использования библиотеки классов MFC, вся работа по регистрации классов окон выполняется автоматически, что существенно облегчает работу программиста.

Благодаря тому, что окна приложения создаются на основе общего базового класса, значительно сокращается объем информации, которую при этом следует указывать. Поскольку класс окна содержит в себе описания элементов, общих для всех окон данного класса, нет необходимости повторять эти описания при создании каждого нового окна. К тому же в приложениях, использующих функции API, все окна одного класса используют одну общую оконную процедуру (функцию), обеспечивающую работу с различными однотипными окнами. Это позволяет избежать дублирования кода.

1.4. Графические объекты, используемые в окнах

Примерами *графических объектов*, с которыми можно обращаться как с единым целым и которые выступают элементами пользовательского интерфейса, являются: строка меню, кнопки, полосы прокрутки и т. д. Другие, широко используемые графические объекты оконных приложений Windows кратко будут описаны далее.

1.4.1. Значки

Значками называются маленькие графические изображения, выполняющие опознавательную функцию. Так, значки приложений на панели задач позволяют легко определить, какие программы в настоящий момент запущены, даже если названия программ не отображаются целиком. Значки могут быть полезны и в самих приложениях, поскольку с их помощью можно привлекать внимание пользователей к сообщениям об ошибках и различным предупреждениям. В состав операционной системы Windows входит набор стандартных значков, в частности, стилизованные знак вопроса, восклицательный знак и ряд других значков. С помощью встроенного в IDE редактора ресурсов вы можете создавать собственные значки.

1.4.2. Указатели мыши

Указатели мыши также являются графическими объектами, используемыми для отслеживания перемещения мыши. Вид указателя может меняться в зависимости от выполняемого задания и состояния системы. Например, стандартный указатель в виде стрелки изменяет свой вид на изображение песочных часов в том случае, если система занята. С помощью встроенного в IDE редактора ресурсов вы можете создавать собственные указатели мыши.

1.4.3. Текстовые курсоры

Курсоры предназначены для указания места, куда следует осуществлять ввод текстовых данных. Отличительной особенностью курсоров является их мерцание. В большинстве текстовых редакторов и полях диалоговых окон в качестве курсора применяется курсор в виде символа "I". Обратите внимание, что в Windows нет (в отличие от значков и указателей мыши) коллекции готовых курсоров.

1.4.4. Окна сообщений

Окна сообщений представляют собой разновидность диалоговых окон, содержащих строку заголовка, значок (значки), текст сообщения и кнопку (кнопки). На рис. 1.3 показано стандартное окно сообщения, которое появляется при закрытии окна программы Notepad (Блокнот) в том случае, если в нем содержатся несохраненные данные.

Окно сообщений создается путем вызова функции `MessageBox`, аргументы которой задают текст заголовка окна, текст сообщения, какой из стандартных значков Windows использовать (если это необходимо) и какой набор кнопок

выводить. В частности, можно вызывать окна со следующими комбинациями кнопок: **Abort**/**Retry**/**Ignore** (Прервать/Повторить/Игнорировать), **OK**, **Yes**/**No** (Да/Нет), **Yes**/**No**/**Cancel** (Да/Нет/Отмена), **OK**/**Cancel** (ОК/Отмена) и **Retry**/**Cancel** (Повторить/Отмена).

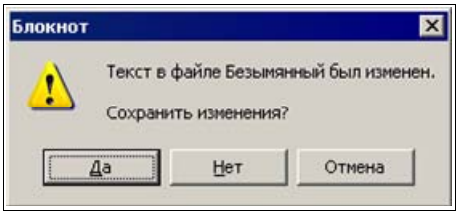


Рис. 1.3. Окно сообщения текстового редактора Notepad

1.4.5. Диалоговые окна

Диалоговые окна содержат наборы различных элементов управления. Они позволяют пользователю задавать опции и параметры программы, которой принадлежит диалоговое окно. Пример диалогового окна для настройки параметров печати показан на рис. 1.4. Windows автоматически отображает элементы управления, содержащиеся в окне. Внешний вид диалогового окна разрабатывается с помощью встроенного в IDE редактора ресурсов.

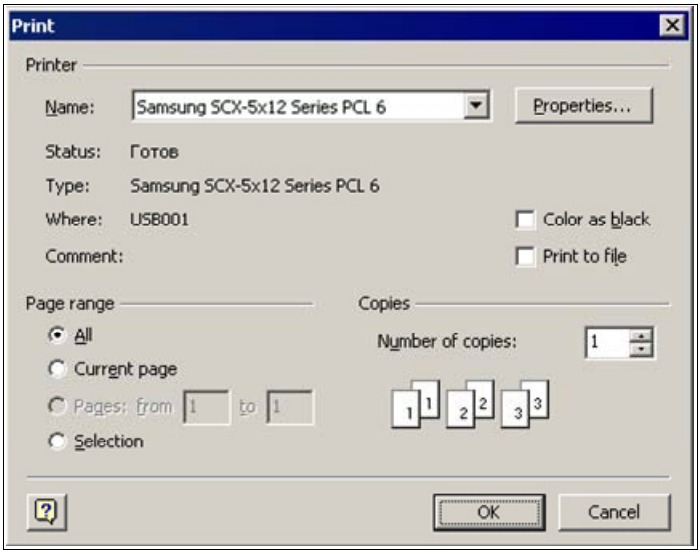


Рис. 1.4. Диалоговое окно настройки параметров печати

1.4.6. Шрифты

Шрифт в Windows — это графический ресурс, содержащий набор символов определенного типа. Шрифты бывают растровые, контурные и масштабируемые.

Растровый шрифт (raster font) представляет символы с помощью растровых изображений. Такие изображения плохо поддаются масштабированию, из-за чего страдает качество образов.

Контурный шрифт (stroke font) представляет символы с помощью линий. Такие шрифты предназначены, прежде всего, для графопостроителей (плоттеров).

Масштабируемый шрифт TrueType, (true type font) представляет символы с помощью линий и сплайновых кривых. Такие шрифты масштабируются практически до любого размера без ухудшения качества образов.

Примечание

Всегда стремитесь использовать масштабируемые шрифты.

Существует набор функций, с помощью которых можно манипулировать начертанием символов для получения форматированного текста. В приложениях можно использовать как стандартные шрифты, так и пользовательские шрифты. Встроенные функции позволяют получать, на базе основного шрифта, полужирное начертание, курсив, подчеркнутый текст и изменять размер шрифта. Внешний вид шрифта можно сделать независимым от типа устройства, на которое выводится текст. Так, при использовании технологии TrueType обеспечивается соответствие между внешним видом шрифта на экране и шрифта, выводимого при печати.

1.4.7. Точечные рисунки

Точечные рисунки представляют собой точную копию части экрана, снятую попиксельно. Тот факт, что изображение является точным образом экрана, устраняет необходимость в каких-либо дополнительных преобразованиях, что существенно сокращает время вывода изображения на экран. В Windows точечные рисунки наиболее широко применяются для двух целей. Во-первых, они служат изображениями всевозможных кнопок и значков, например, стрелок полос прокрутки и кнопок панелей инструментов. Другой областью применения точечных рисунков являются кисти, с помощью которых рисуются и заполняются цветом различные геометрические фигуры на экране.

Точечные рисунки можно создавать и модифицировать с помощью встроенного редактора ресурсов.

1.4.8. Перья

Перья предназначены для рисования геометрических фигур и различных контуров. Перья характеризуются тремя основными параметрами:

- ❒ шириной линии;
- ❒ стилем (точечный, штрихпунктирный, непрерывный);
- ❒ цветом.

Существует два готовых пера: одно для рисования черных линий, другое — для рисования белых. С помощью специальных функций вы можете создавать собственные перья.

1.4.9. Кисти

Кисти предназначены для заливки объектов цветом, выбранным из заданной палитры. Минимальный размер кисти — 8×8 пикселов. Кисть также характеризуется тремя параметрами:

- ❒ размером;
- ❒ шаблоном заливки;
- ❒ цветом.

Заливка может быть сплошной, штриховой, диагональной или представлять собой узор, заданный пользователем.

1.5. Принципы обработки сообщений

Ни одно приложение в Windows не отображает свои данные непосредственно на экране, не обрабатывает напрямую прерывания устройств и не выводит данные непосредственно на печать. Вместо всего этого приложение вызывает встроенные функции Windows и ожидает от системы соответствующих сообщений.

Операционная система Windows является *событийно-управляемой* ОС. Этот термин означает, что отдельные части ОС взаимодействуют между собой, а также с прикладными программами посредством сообщений.

Сообщение — это форма регистрации событий в операционной системе (или просто в системе). Система использует сообщения для того, чтобы сигнализировать о совершении событий.

Подсистема сообщений в Windows, используемая в операционной системе и пользовательских приложениях, — это средство распределения информации в многозадачной среде. С точки зрения приложения, сообщение — это уве-

домление о некотором произошедшем событии, на которое приложение должно отреагировать определенным образом. Такое событие может быть инициировано пользователем, например, нажатием клавиши или перемещением мыши, изменением размера окна или выбором команды из меню. Но события могут порождаться и самим приложением.

Особенность этого процесса состоит в том, что приложение должно быть полностью ориентировано на прием и обработку сообщений. Программа должна быть готова в любой момент принять сообщение, определить его тип, выполнить соответствующую обработку и вновь перейти в режим ожидания до поступления следующего сообщения.

Приложения ОС Windows существенно отличаются от приложений, написанных для MS DOS. ОС Windows открывает приложениям доступ к сотням встроенных функций, которые можно вызывать напрямую (низкий уровень) или косвенно (высокий уровень), посредством библиотек типа MFC. Эти функции содержатся в ряде модулей, таких как KERNEL, GDI и USER. Функции модуля KERNEL отвечают за управление памятью, загрузку приложений, выполнение приложений и распределение системных ресурсов. Модуль GDI содержит функции создания и отображения графических объектов, а модуль USER отвечает за выполнение всех других функций, обеспечивающих взаимодействие приложений с пользователями и средой ОС Windows.

1.5.1. Формат сообщений

Сообщения используются для информирования приложения о том, что в системе произошло то или иное событие. На практике, сообщения направляются не столько самому приложению, сколько определенному окну, открытому этим приложением.

Реально в Windows существует только один механизм обработки сообщений — системная очередь сообщений. Но каждое выполняющееся приложение организовывает и свою очередь. Функции модуля USER, в частности, ответственны за передачу сообщений из системной очереди в очередь конкретного приложения. Таким образом, в очереди конкретного приложения накапливаются сообщения, адресованные любому окну, открытому данным приложением.

Независимо от типа, все сообщения для 32-разрядных версий Windows характеризуются четырьмя параметрами (рис. 1.5):

- дескриптором окна, которому адресуется данное сообщение;
- типом сообщения;
- еще двумя 32-разрядными параметрами.

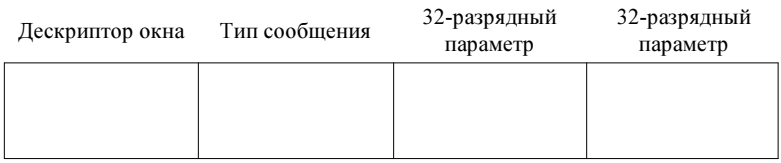


Рис. 1.5. Структура сообщения

Дескрипторы (handle, описатель) широко используются в приложениях ОС Windows. Дескриптором называется уникальный номер (беззнаковое 32-разрядное целое значение), который присваивается всем системным объектам, таким как: окна, элементы управления, меню, значки, перья и кисти, а также областям памяти, устройствам вывода и т. д.

Поскольку Windows позволяет одновременно открывать несколько копий одного приложения, операционная система должна иметь возможность отслеживать каждую копию в отдельности. Это достигается путем присвоения каждому экземпляру программы своего дескриптора.

Дескрипторы обычно служат в качестве индексов системной таблицы объектов. Благодаря тому, что доступ к объектам осуществляется по индексам таблицы, а не по их непосредственным адресам в памяти, Windows может динамически перераспределять ресурсы за счет обновления адресов в таблице. Например, если Windows связала некоторый ресурс приложения с 16-й строкой таблицы, то независимо от того, куда впоследствии Windows переместит этот ресурс, его текущий адрес всегда будет представлен в 16-й строке.

Тип сообщения (второй параметр) задается идентификатором, который определен в одном из файлов заголовков Windows. Для работы с идентификаторами сообщений в программу включается файл windows.h. Как правило, идентификаторы начинаются с двухсимвольного префикса, за которым следует символ подчеркивания. Так, оконные сообщения начинаются с префикса `WM_`: `WM_CREATE`, `WM_DESTROY`, `WM_SIZE`, `WM_PAINT`, `WM_QUIT`, `WM_COMMAND` и др. Сообщения кнопок имеют префикс `BM_`, полей — `EM_` и т. д. В приложении можно также создать и зарегистрировать собственный тип сообщения, предназначенного для частных целей.

Последние два параметра сообщения несут дополнительную информацию. Их содержание может изменяться в зависимости от типа сообщения. Например, посредством этих параметров может передаваться информация о том, какая клавиша была нажата, какая команда меню активизирована и т. д.

Форматы некоторых оконных сообщений приведены на рис. 1.6—1.10. Разберем их.

Система посылает окну Windows сообщение `WM_CREATE` *после* его создания, но *до* появления образа окна на экране. Свой первый параметр сообщение

WM_CREATE не использует. Второй параметр содержит указатель на структуру типа CREATESTRUCT, которая содержит информацию о создаваемом окне. Если приложение обрабатывает WM_CREATE, то при успешной работе оно должно возвращать 0. При этом создание окна продолжается. Если же приложение возвращает –1, то функция создания окна (CreateWindow или CreateWindowEx) возвращает пустой манипулятор (со значением 0).

Рис. 1.6. Структура сообщения WM_CREATE

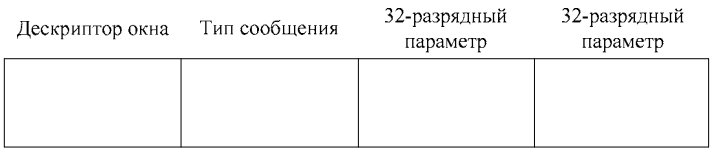


Рис. 1.7. Структура сообщения WM_DESTROY

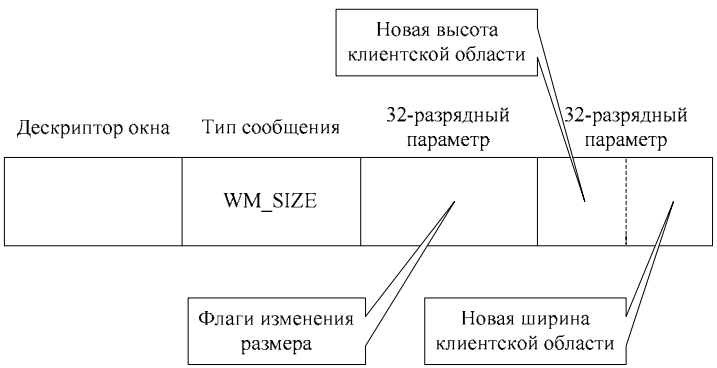


Рис. 1.8. Структура сообщения WM_SIZE

Система посылает сообщение WM_DESTROY окну перед его уничтожением *после* того, как образ окна удален с экрана. Сообщение WM_DESTROY не использует

оба параметра. Если приложение обрабатывает `WM_DESTROY`, то оно должно возвращать значение 0.

Система посылает сообщение `WM_SIZE` окну Windows *после* того, как размер окна изменен. Значения новых размеров клиентской области окна можно получить с помощью макросов: `LOWORD(lParam)` дает ширину, а `HIWORD(lParam)` — высоту. Если приложение обрабатывает `WM_SIZE`, то оно должно возвращать значение 0.

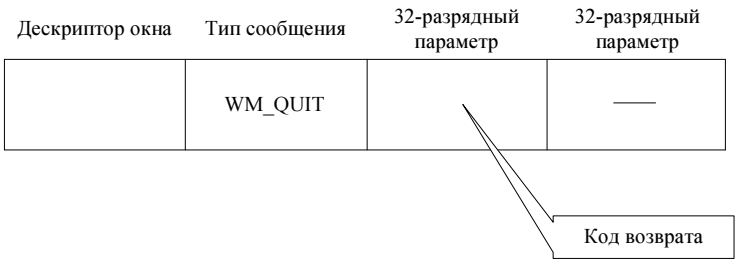


Рис. 1.9. Структура сообщения `WM_QUIT`

Сообщение `WM_QUIT` означает запрос на завершение работы приложения. Оно приводит к тому, что специальная функция `GetMessage` возвращает значение 0. Первый параметр задает значение кода возврата, который приложение передает ОС при своем завершении.

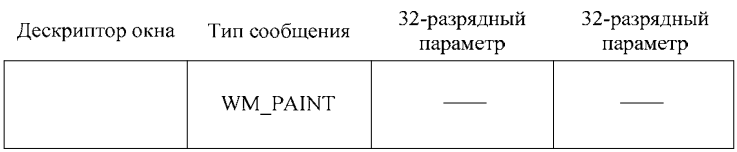


Рис. 1.10. Структура сообщения `WM_PAINT`

Сообщение `WM_PAINT` посылается окну, если система, или другое приложение, выдает запрос на перерисовку окна (или его части). В частности, `WM_PAINT` посылается в результате вызова функций `UpdateWindow` и `RedrawWindow`. Сообщение `WM_PAINT` не использует оба параметра. Если приложение обрабатывает `WM_PAINT`, то оно должно возвращать значение 0.

1.5.2. Генерирование сообщений

Именно реализация концепции обмена сообщениями позволяет Windows быть многозадачной средой. Работа Windows основана на передаче, приеме и обработке сообщений.

Существует четыре основных *источника*, от которых приложение может получить сообщение:

- пользователь;
- ОС Windows;
- само приложение;
- другие приложения.

Сообщения пользователей включают информацию о вводе текста, перемещении мыши, нажатии кнопок мыши, выборе команд меню, перемещении бегунка полос прокрутки и т. д. Большую часть времени приложение занято обработкой именно таких сообщений. Получение сообщения от пользователя означает, что человек, запустивший программу, хочет изменить ход ее выполнения.

Системные сообщения посылаются программе при изменении ее состояния. Например, щелчок на значке приложения означает, что пользователь хочет сделать данное приложение активным. В этом случае Windows сообщает приложению, что на экране открывается его окно, размер и положение окна изменяются и т. д. В зависимости от текущего состояния приложения сообщение от Windows может быть принято и обработано либо проигнорировано.

В следующем разделе мы рассмотрим, как создаются простейшие приложения Windows. Вы узнаете, что, по сути, в приложении используется ряд процедур, каждая из которых отвечает за обработку определенного типа сообщения для определенного окна. Например, одна из таких процедур может отвечать за изменение размеров окна. При этом совсем не обязательно, чтобы это происходило только по команде пользователя, — решение об изменении может принимать и сама программа.

Сообщения от других приложений в настоящее время применяются достаточно редко. Примером межзадачного обмена сообщениями может служить протокол DDE (Dynamic Data Exchange, динамический обмен данными).

1.5.3. Обработка сообщений

Оконные приложения Windows, написанные на языке C++, содержат специальные процедуры для обработки всех типов сообщений, которые могут быть посланы программе. Разные окна программы могут по-разному реагировать на одни и те же сообщения. Это достигается за счет того, что *обработчики сообщений* пишутся отдельно для каждого окна, а Windows знает, какому именно окну адресовано сообщение. Таким образом, приложения содержат процедуры обработки не только сообщений разных типов, но и сообщений для разных окон.

1.5.4. Цикл обработки сообщений

Основным компонентом любого приложения ОС Windows является *цикл обработки сообщений*. Если говорить в целом, то работа приложения организуется следующим образом. Сначала приложение создает и открывает свои окна (в общем случае их может быть несколько), затем запускается цикл обработки сообщений и, в конце концов, при получении сообщения `WM_CLOSE`, работа приложения завершается. Цикл обработки сообщений ответственен за передачу поступающих от Windows сообщений соответствующим оконным процедурам.

На последовательность обработки сообщений влияют два фактора:

- организация очереди, в которую помещается сообщение;
- приоритет приложения.

Сообщения могут поступать из двух очередей — *системной* и очереди *приложения*. Важным является то, что, даже если сообщение поступило от самого приложения, сначала оно будет помещено в системную очередь. Когда в системной очереди подойдет черед сообщения, оно будет направлено в очередь соответствующего приложения. Такая организация работы системной очереди позволяет Windows контролировать прохождение всех сообщений и ограничивает ответственность приложений обработкой только тех сообщений, которые относятся непосредственно к приложениям.

Сообщения обычно помещаются в очередь по принципу FIFO (First In, First Out, первым поступило, первым обслужено). Такие сообщения называются *синхронными*. Но иногда Windows вставляет сообщение сразу в голову очереди. Сообщения такого типа, изменяющие нормальный ход выполнения программы, называются *асинхронными*.

Существует несколько видов асинхронных сообщений, среди них — сообщение о перерисовке, сообщение таймера и сообщение о завершении программы. Например, сообщение таймера может запускать некоторые действия в определенный момент времени, независимо от того, какие сообщения сейчас находятся в очереди; оно имеет наивысший приоритет и передается раньше всех других сообщений.

Возникает вопрос — как Windows выходит из положения, если сообщение предназначено одновременно нескольким приложениям? Эта коллизия разрешается одним из двух способов. Первый состоит в задании *приоритетов*. Когда загружается некоторое приложение, для него автоматически устанавливается нулевой приоритет. В процессе работы приложения его приоритет может быть изменен. Любые конфликты разрешаются в пользу того приложения, чей приоритет выше.

Однако изменять приоритет приложения, заданный по умолчанию, обычно не рекомендуется. Тем более что в распоряжении Windows есть еще один метод,

определяющий очередность передачи сообщений группе приложений с одинаковым приоритетом. Если Windows "видит", что приложение имеет длинную очередь необработанных сообщений, то она автоматически приостанавливает передачу этому приложению новых сообщений, хотя продолжает передавать сообщения другим, более свободным приложениям.

1.5.5. Файл Windows.h

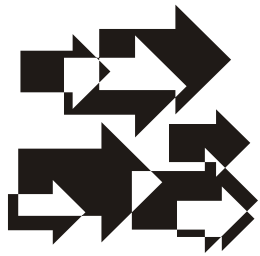
Включаемый файл Windows.h имеет большой размер и открывает доступ к тысячам констант, структур, типов данных и прототипов функций. Он включается в большинство приложений Windows и содержит директивы включения множества других заголовочных файлов. Это является основной причиной того, почему Windows-приложения компилируются так долго (далее, в *главе 2* будет рассмотрен механизм прекомпиляции, устраняющий этот недостаток). В случае использования библиотеки классов MFC файл Windows.h подключается к программе косвенно, через файл afxwin.h.

1.6. Вопросы для самопроверки

1. Что представляет собой окно?
2. Перечислите и охарактеризуйте основные компоненты окна.
3. Что такое класс окна, зачем он нужен?
4. Перечислите и охарактеризуйте графические объекты, используемые в окнах.
5. Что такое событийно-управляемая ОС?
6. Дайте общую характеристику формата сообщения, используемого в Windows.
7. Опишите сообщение `WM_CREATE`.
8. Опишите сообщение `WM_DESTROY`.
9. Опишите сообщение `WM_SIZE`.
10. Опишите сообщение `WM_QUIT`.
11. Опишите сообщение `WM_PAINT`.
12. Перечислите основные источники сообщений Windows.
13. Как в общем виде организован цикл обработки сообщений?
14. Укажите назначение и особенности стандартного заголовочного файла Windows.h.

Ответы на вопросы можно проверить — они приведены в *разд. П1.1 приложения 1*.

ГЛАВА 2



Низкоуровневое проектирование Windows-приложений [3, 4]

Наиболее важными и привлекательными особенностями Windows-приложений следует считать стандартный интерфейс, аппаратную независимость и возможность их одновременного выполнения. Для того чтобы уметь хорошо проектировать Windows-приложения с использованием библиотеки классов MFC (Microsoft Foundation Classes), весьма полезно рассмотреть анатомию простого Windows-приложения, построенного на основе низкоуровневых средств программирования, таких как Windows SDK и Windows API.

Пакет разработки приложений Windows SDK [5] представляет собой набор средств для разработки Windows-приложений. За многие годы использования Windows SDK заложил основу для современного программирования с помощью библиотеки классов MFC. Комплект разработки Windows-приложений содержит следующие элементы:

- довольно объемную литературу со сведениями о функциях, сообщениях, структурах, макросах и ресурсах ОС Windows;
- различные инструменты, в том числе, например, редактор ресурсов;
- файлы оперативной справки;
- набор библиотек и файлов заголовков Windows;
- примеры Windows-приложений, написанные на языке C.

Главное отличие Windows SDK от библиотеки классов MFC примерно то же, что и отличие языка C от языка C++. Применение библиотеки MFC требует перехода от языка C к языку C++ с его средствами объектно-ориентированного программирования (ООП). Далее вы заметите, что имеется очень много общего в Windows-приложениях, написанных на языке C с помощью SDK, и в Windows-приложениях, созданных с помощью библиотеки классов MFC. Это происходит потому, что новые версии Windows работают в основ-

ном так же, как и предшествующие. Библиотека классов MFC просто "прячет" всю сложность механизмов Windows от программиста. Это хорошо, потому что внутреннее строение ОС Windows становится все сложнее и сложнее!

Windows API представляет собой набор функций, совместно вызываемых из одной программы (или из связанного набора программ), который программист использует для создания Windows-приложений и взаимодействия с операционной системой. Разбираться во внутреннем содержании функций совсем не обязательно — *достаточно* иметь лишь их документированные спецификации. Сегодня функции Windows API являются наиболее известными интерфейсами прикладных программ, и не без основания — Windows есть везде. Это неплохой повод для того, чтобы подробнее познакомиться с Windows API, но для этого есть и другие причины. ОС Windows сама использует функции API, демонстрируя свой поразительный графический интерфейс.

Прежде чем рассмотреть простейшее Windows-приложение, созданное на основе Windows API, в качестве справки, перечислим наиболее часто используемые данные (табл. 2.1) и структуры (табл. 2.2) Windows.

Таблица 2.1. *Наиболее часто используемые типы данных Windows*

Тип	Описание
CALLBACK	Замещает спецификацию FAR PASCAL в функциях обратного вызова
HINSTANCE	32-разрядное беззнаковое целое число, используемое в качестве дескриптора
HDC	Дескриптор контекста устройства
HWND	Дескриптор окна
LPARAM	Используется для описания младшего аргумента сообщения
LPCSTR	32-разрядный указатель на константную строку
LPSTR	32-разрядный указатель на строку
LRESULT	Используется для возвращения значений из оконных процедур
UINT	32-разрядное беззнаковое целое число
WINAPI	Замещает спецификацию FAR PASCAL в описаниях функций API
WPARAM	Используется для описания старшего аргумента сообщения

Таблица 2.2. *Наиболее часто используемые структуры Windows*

Структура	Что определяет
MSG	Параметры сообщения
PAINTSTRUCT	Область окна, требующая перерисовки
WNDCLASS	Класс окна

2.1. Основные компоненты приложения

Windows-приложения содержат два общих элемента:

- функцию `WinMain`;
- оконную процедуру.

Функция `WinMain` составляет основу любого приложения. Она служит точкой входа в приложение и выполняет ту же роль, что и функция `main` в обычных программах на языках C и C++.

Оконная процедура исполняет роль диспетчера сообщений. Приложения никогда не получают к ней прямого доступа. Для этого им сначала нужно сделать запрос к Windows на выполнение соответствующего действия. Поэтому все оконные процедуры должны быть *функциями обратного вызова (callback)*. Подобная функция регистрируется в операционной системе и вызывается всякий раз, когда выполняется некоторое действие над окном.

2.2. Функция *WinMain* и цикл обработки сообщений

Функция `WinMain`, как было сказано, является обязательным компонентом любого Windows-приложения. С нее начинается выполнение программы и ею же обычно заканчивается. Эта функция состоит из трех функциональных частей, в которых, соответственно, выполняются:

- начальная инициализация приложения (подготовка данных класса окна и его регистрация);
- создание главного окна приложения;
- запуск цикла обработки сообщений, извлекаемых из очереди сообщений приложения (цикл обработки сообщений и приложение завершаются после получения сообщения `WM_QUIT`).

Совет

В соответствии с принципом функциональной декомпозиции, начальную инициализацию приложения (подготовку данных класса окна и его регистрацию) и создание главного окна приложения рекомендуем оформить в виде соответствующих функций, вызываемых из функции `WinMain`.

На листинге 2.1 приведена стандартная реализация простейшего низкоуровневого Windows-приложения.

Листинг 2.1. Файл FrameWnd.cpp (начало)

```
/*
   Файл                : FrameWnd.cpp

   Проект              : простейшее приложение Win32API с
                         использованием стандартных функций API
                         (создается главное окно, в которое
                         производится вывод текста и графики)

   Microsoft Visual Studio C++ .NET 2005
*/

// Прекомпилируемый заголовочный файл
#include "stdafx.h"

// Прототипы используемых функций (!!! имена функций не системные
//   и могут быть иными — здесь нет связи с классами или
//   библиотеками)
LRESULT CALLBACK WndProc( HWND hwnd, UINT message, WPARAM wParam,
                          LPARAM lParam );
BOOL InitApplication( HINSTANCE hInstance );
BOOL InitInstance( HINSTANCE hInstance, int nCmdShow );

// Указатель на имя регистрируемого класса
LPCSTR                  szClassName = "FrameWndAPI";
// Указатель на заголовок окна приложения
LPCSTR                  szTitle = "Создание Windows-приложения"
    " FrameWnd с использованием стандартных функций SDK и API";

// ************************************************************
// Главная функция приложения
int WINAPI WinMain(                 // Возвращает TRUE при успехе
    // Дескриптор данного приложения
    HINSTANCE           hInstance,
```

```
    // Дескриптор предыдущей запущенной копии приложения.
    //   В Win32API этот параметр всегда равен NULL и оставлен
    //   исключительно для совместимости с версиями ниже четвертой.
    //   Связано это с тем, что каждое 32-разрядное приложение
    //   запускается в собственном адресном пространстве, в
    //   котором, естественно, нет никаких копий или других
    //   приложений
    HINSTANCE           hPrevInstance,
    // Указатель на командную строку, которую можно в
    //   интегрированной среде разработки задать по команде
    //   "Properties" контекстного меню проекта в элементе "Program
    //   arguments" вкладки "Debugging"
    LPSTR               lpCmdLine,
    // Режим начального отображения главного окна приложения —
    //   после вызова параметр получает значение SW_SHOWNORMAL (1),
    //   что соответствует отображению окна в нормальном виде
    int                 nCmdShow )
{
    // Хотя параметр hPrevInstance в Win32API всегда равен NULL,
    //   все же продолжаем для совместимости проверять его значение
    if ( !hPrevInstance )
    {
        // Инициализируем приложение — подготавливаем данные класса
        //   окна и регистрируем его
        if ( !InitApplication( hInstance ) )
            return FALSE;
    }

    // Завершаем создание приложения, создаем и отображаем главное
    //   окно приложения
    if ( !InitInstance( hInstance, nCmdShow ) )
        return FALSE;

    MSG                 msg;        // Для очередного сообщения
    // Стандартный цикл обработки сообщений
    while ( GetMessage( &msg, NULL, 0, 0 ) )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
    /*
       Данный цикл работает в течение всего времени выполнения
    программы. Каждая итерация представляет собой прием одного
```

сообщения из очереди сообщений приложения. За это отвечает системная функция GetMessage( ), второй аргумент которой (NULL) говорит о том, что читаться должны сообщения, адресованные любому окну приложения. Третий и четвертый аргументы формируют фильтр сообщений, который ограничивает получаемые сообщения определенным диапазоном. В нашем случае эти аргументы нулевые и будут приниматься любые сообщения, адресованные данному приложению. Когда сообщение выбирается из очереди, оно помещается в специальную структуру с типом MSG, на которую указывает первый аргумент и которая выглядит следующим образом:

```
typedef struct tagMSG
{
    // Дескриптор (логический номер) окна, чья оконная
    //   процедура получает сообщение
    HWND            hwnd;
    UINT            message;    // Номер сообщения
    WPARAM          wParam;     // Дополнительная информация о
                                //   сообщении, зависящая
    LPARAM          lParam;     //   от типа сообщения (wParam,
                                //   lParam)
    DWORD           time;       // Время возникновения
                                //   сообщения
    // Позиция указателя мыши на момент возникновения сообщения
    POINT           pt;
    } MSG;
```

После того как сообщение взято из очереди сообщений, оно передается в функцию TranslateMessage( ), которая вызывает драйвер клавиатуры ОС Windows для преобразования виртуальных кодов клавиш в ASCII-значения, которые ставятся в очередь программных событий в виде сообщения WM_CHAR. Это позволяет программе отличить, например, 'A' от 'a' без анализа состояния клавиши регистра. Эта функция необходима только тем приложениям, которые обрабатывают ввод данных с клавиатуры. Важность функции TranslateMessage( ) заключается в том, что она позволяет пользователям выбирать команды меню не только шелчками мыши, но и путем нажатия клавиш.

Последняя функция цикла — DispatchMessage( ) берет данные о сообщении из структуры msg и передает их в соответствующую оконную процедуру для обработки. После того как сообщение передано, снова вызывается функция GetMessage( ), чтобы взять из очереди следующее сообщение, если таковое имеется. За исключением WM_QUIT, каждое

```
        сообщение заставляет GetMessage( ) возвратить значение
        TRUE. Принимая сообщение WM_QUIT, программа выходит из
        цикла обработки сообщений и завершает работу.
           Важно понимать, что этот, или подобный ему, цикл без
        конца повторяется в течение всей жизни программы,
        написанной для ОС Windows.
    */

    return static_cast< int >( msg.wParam );
}
// ... Продолжение файла следует далее
```

2.3. Подготовка данных класса окна и его регистрация

В приложении можно определить собственный класс окна, задав его свойства и зарегистрировав его в Windows. Следующий фрагмент файла FrameWnd.cpp, приведенный на листинге 2.2, и демонстрирует это.

Листинг 2.2. Файл FrameWnd.cpp (продолжение)

```
// ****************************************************************
/*
   Любое окно ОС Windows принадлежит одному из существующих в данный
момент в системе классов, который должен быть определен до того,
как окно будет отображено на экране. Класс окна задает наиболее
общие свойства окон, например, форму курсора при перемещении его в
область окна, или имя меню, определенного для окон этого класса.
   Другими словами, класс окна — это шаблон, в котором определяются
выбранные стили, шрифты, заголовки, пиктограммы, размер,
расположение окна и т. д. Поскольку каждый класс имеет свою
структуру параметров окна, доступную всем, не происходит ненужного
дублирования данных. При этом следует учитывать, что имя класса
должно быть уникальным, чтобы не возникало конфликтов с классами
окон других приложений. Кроме того, два окна одного и того же
класса используют общую функцию окна (оконную процедуру) со всеми
ее вспомогательными функциями
*/
// Подготовка данных класса окна и его регистрация в ОС Windows
BOOL InitApplication(               // Возвращает TRUE при успешном
                                    //   завершении
```

```
    // Дескриптор (уникальное число), ассоциируемый с текущим
    //   приложением
    HINSTANCE           hInstance )
{
    // Сведения о регистрируемом классе
    WNDCLASS            wc;
    /*
       Объявление структуры WNDCLASS имеет вид:
    typedef struct tagWNDCLASSA
    {
        // Определяет свойства окна, которые могут комбинироваться
        //   при помощи операции | (ИЛИ). При программировании для
        //   ОС Windows этот термин относится к набору параметров,
        //   каждый из которых управляется одним или двумя битами.
        //   Если этому полю присвоить значение NULL, то ОС Windows
        //   автоматически установит значения по умолчанию. Окно
        //   является основой приложения
        UINT            style;
        //   Для того чтобы больше не возвращаться к этому
        //   вопросу, перечислим список допустимых стилей окна:
    // CS_BYTEALIGNCLIENT
    // Оказывает влияние на ширину окна, задавая для клиентской
    //   области окна выравнивание по границе байта. Это позволяет
    //   увеличить производительность операций рисования в окне
    // CS_BYTEALIGNWINDOW
    // Задает выравнивание ширины окна по границе байта. Это
    //   позволяет увеличить производительность некоторых типов
    //   операций, таких как перемещение и изменение размеров окна,
    //   рисование пунктов меню
    // CS_CLASSDC
    // Указывает, что все окна данного класса должны использовать
    //   один и тот же контекст устройства
    // CS_DBLCLKS
    // Служит для установки таймера после получения первого
    //   сообщения о нажатии кнопки мыши. Сообщение о двойном
    //   нажатии будет сформировано только в том случае, если
    //   второе нажатие произойдет по истечении определенного
    //   времени таймера. Обеспечивает всем окнам класса
    //   возможность воспринимать сообщения о двойном нажатии одной
    //   из кнопок мыши
    // CS_GLOBALCLASS
    // Позволяет приложению создавать окно этого класса, не
    //   считаясь со значением параметра hInstance. Если вы не
```

```
//   определите этот стиль, то параметр hInstance, передаваемый
//   в функции создания окна, должен быть таким же, который
//   передается в функцию RegisterClass( ). Другими словами,
//   установка этого стиля позволяет создавать оконные классы,
//   предназначенные для совместного использования несколькими
//   приложениями
// CS_HREDRAW
// Определяет, что окно будет перерисовываться, как только
//   изменится его горизонтальный размер
// CS_VREDRAW
// Определяет, что окно будет перерисовываться, как только
//   изменится его вертикальный размер
// CS_NOCLOSE
// Запрещает пункт Close (Закрыть) системного меню. Этот стиль
//   следует использовать с окнами, имеющими системное меню,
//   которые не могут быть закрытыми пользователем
// CS_OWNDC
// Предоставляет частный контекст устройства для каждого окна в
//   данном классе. Этот тип контекста устройства наиболее
//   расточителен в смысле использования памяти, но
//   обеспечивает самую высокую скорость реакции
// CS_PARENTDC
// Устанавливает, что дочерние окна не могут рисовать в своем
//   родительском окне. Этот стиль обычно применяется для
//   предварительно определенных классов, необходимых для
//   создания элементов управления блоков диалога
// CS_SAVEBITS
// Предписывает системе сохранять копию части экрана,
//   поврежденную окном. После удаления окна поврежденная часть
//   может быть восстановлена. При этом Windows не посылает
//   сообщения WM_PAINT для перерисовки области, которую
//   занимало окно. Этот стиль применяется для небольших окон,
//   например меню или блоков диалога, которые появляются на
//   экране на короткие промежутки времени
    WNDPROC         lpfnWndProc;// Адрес функции окна, которая
                                //   обрабатывает сообщения для
                                //   окон данного класса
    // Задает число байт, которое необходимо дополнительно
    //   запросить у ОС Windows под эту структуру для хранения
    //   собственных данных, присоединенных к классу. Это поле
    //   может иметь значение NULL
    int             cbClsExtra;
    // Задает число байт, которое необходимо дополнительно
    //   запросить у ОС Windows для размещения всех структур,
```

```
//   создаваемых совместно с данным классом для хранения
//   собственных данных, присоединенных к окну. Это поле
//   также может иметь значение NULL
int             cbWndExtra;
// Кто создает определение класса. Это необходимо для
//   служебных действий внутри ОС Windows. Когда
//   завершается последний экземпляр программы, ОС Windows
//   удаляет все связанные определения классов
HINSTANCE       hInstance;
// Определяет пиктограмму, которая будет использоваться для
//   изображения приложения, например, на панели задач. Вы
//   можете создать пиктограмму сами или использовать одну
//   из предопределенных пиктограмм:
//   IDI_APPLICATION — стандартная для приложения,
//   IDI_HAND — знак "стоп",
//   IDI_QUESTION — "вопросительный знак",
//   IDI_EXCLAMATION — "восклицательный знак",
//   IDI_ASTERISK — "информация"
//   Это поле может иметь значение NULL
HICON           hIcon;
// Идентифицирует курсор мыши по умолчанию, используемый в
//   данном окне. Если нет желания создавать свой курсор,
//   то можно воспользоваться одним из представляемых
//   системой:
//   IDC_ARROW — стрелка;
//   IDC_IBEAM — вертикальная черта;
//   IDC_WAIT — песочные часы;
//   IDC_CROSS — перекрестье;
//   IDC_UPARROW — вертикальная стрелка;
//   IDC_SIZE — четыре стрелки, указывающие в разные
//              стороны;
//   IDC_ICON — курсор, используемый при переносе файлов;
//   IDC_SIZENWSE — двунаправленная стрелка
//                  северо-запад/юго-восток;
//   IDC_SIZENESW — двунаправленная стрелка
//                  северо-восток/юго-запад;
//   IDC_SIZEWE — двунаправленная стрелка восток/запад;
//   IDC_SIZENS — двунаправленная стрелка север/юг
// Имеются и другие предопределенные стили курсора —
//   см. справку по функции LoadCursor( )
HCURSOR         hCursor;
// Задает цвет фона для окна. Для его определения можно
//   воспользоваться любой из 20 системных констант цвета,
//   которые определены в заголовочном файле winuser.h:
```

```
    //   #define COLOR_SCROLLBAR           0
    //   #define COLOR_BACKGROUND          1
    //   #define COLOR_ACTIVECAPTION       2
    //   #define COLOR_INACTIVECAPTION     3
    //   #define COLOR_MENU                4
    //   #define COLOR_WINDOW              5
    //   #define COLOR_WINDOWFRAME         6
    //   #define COLOR_MENUTEXT            7
    //   #define COLOR_WINDOWTEXT          8
    //   #define COLOR_CAPTIONTEXT         9
    //   #define COLOR_ACTIVEBORDER        10
    //   #define COLOR_INACTIVEBORDER      11
    //   #define COLOR_APPWORKSPACE        12
    //   #define COLOR_HIGHLIGHT           13
    //   #define COLOR_HIGHLIGHTTEXT       14
    //   #define COLOR_BTNFACE             15
    //   #define COLOR_BTNSHADOW           16
    //   #define COLOR_GRAYTEXT            17
    //   #define COLOR_BTNTEXT             18
    //   #define COLOR_INACTIVECAPTIONTEXT 19
    //   #define COLOR_BTNHIGHLIGHT        20
    // При использовании этих констант к их значению
    //   обязательно нужно добавлять единицу, т. к. первое
    //   значение для них равно нулю, а нуль является
    //   недействительным значением для логического номера
    //   кисти. Кроме того, необходимо выполнить приведение к
    //   типу HBRUSH — иначе компилятор выдаст сообщение об
    //   ошибке.
    // Если значение этого поля равно нулю, то приложение
    //   должно самостоятельно управлять заливкой фона окна при
    //   поступлении соответствующего сообщения
    HBRUSH          hbrBackground;
    // Указатель на имя меню окна, определенное в файле
    //   ресурсов. Это поле может иметь значение NULL
    LPCSTR          lpszMenuName;
    // Указатель на строку, содержащую имя класса (имя класса
    //   должно быть уникальным)
    LPCSTR          lpszClassName;
} WNDCLASS;
*/

// Заполняем структуру класса окна WNDCLASS — смысл
//   инициализирующих значений рассмотрен выше
wc.style         = CS_HREDRAW | CS_VREDRAW;
```

```
    wc.lpfnWndProc   = static_cast< WNDPROC >( WndProc );
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = hInstance;
    wc.hIcon         = LoadIcon( NULL, IDI_ASTERISK );
    wc.hCursor       = LoadCursor( NULL, IDC_CROSS );
    wc.hbrBackground = reinterpret_cast< HBRUSH >( COLOR_WINDOW+1 );
    wc.lpszMenuName  = NULL;
    wc.lpszClassName = szClassName;

    // Регистрируем класс окна
    return RegisterClass( &wc );
    // Функция RegisterClass( ) определяет класс окна, которое
    //   выводится на экран. Все окна ОС Windows — объекты, каждый
    //   из которых имеет набор характерных черт. Заполняя
    //   структуру WNDCLASS, вы сообщаете ОС Windows, какими хотите
    //   видеть конкретные объекты.
    // Чаще всего вообще не нужно думать о методе регистрации.
    //   Просто скопируйте приведенный фрагмент функции
    //   InitApplication( ) в приложение и вызывайте ее даже не
    //   думая о ее содержимом. Регистрация класса окна не является
    //   чем-то таким, о чем необходимо постоянно заботиться — чаще
    //   всего фрагмент кода, отвечающий за нее, просто переносится
    //   из одного приложения в другое
}
// ... Продолжение файла следует далее
```

2.4. Создание главного окна

Регистрация класса окна еще не свидетельствует о создании самого окна. Следующая функция из листинга 2.1, которую необходимо реализовать самостоятельно, это функция `InitInstance`, представленная в следующем фрагменте файла FrameWnd.cpp (листинг 2.3).

Листинг 2.3. Файл FrameWnd.cpp (продолжение)

```
// ****************************************************************
// Создание окна
BOOL InitInstance(                  // Возвращает TRUE при успехе
    // Дескриптор текущего приложения
    HINSTANCE           hInstance,
    // Режим отображения главного окна — определяет, в каком виде
    //   будет отображено окно приложения
```

```
    int                 nCmdShow )
{
    // Дескриптор главного окна
    HWND                hWnd;

    // Создание окна
    hWnd = CreateWindow(
        // Указатель на строку зарегистрированного имени класса
        szClassName,
        // Указатель на строку заголовка окна
        szTitle,
        WS_OVERLAPPEDWINDOW,        // Стиль окна
        // Горизонтальная координата левого верхнего угла окна
        //   (используется умалчиваемое значение)
        CW_USEDEFAULT,
        // Вертикальная координата левого верхнего угла окна
        //   (используется умалчиваемое значение)
        CW_USEDEFAULT,
        CW_USEDEFAULT,              // Ширина окна (используется
                                    //   умалчиваемое значение)
        CW_USEDEFAULT,              // Высота окна (используется
                                    //   умалчиваемое значение)
        NULL,                       // Дескриптор родительского
                                    //   окна (его нет)
        NULL,                       // Дескриптор меню окна (его
                                    //   нет)
        hInstance,                  // Дескриптор экземпляра
                                    //   приложения
        NULL );                     // Указатель на дополнительные
                                    //   данные окна (их нет)
    if ( !hWnd )
        return FALSE;
    /*
       Аргументов в вызове CreateWindow( ) много, но назначение
    каждого из них очевидно. Первые два аргумента — имя класса и
    заголовок — выбраны нами произвольно и позволяют как системе,
    так и пользователю идентифицировать окно. Аргументы с
    четвертого по седьмой задают начальное положение и размеры
    окна. Константа CW_USEDEFAULT предписывает ОС Windows самой
    выбрать эти параметры. Это значение выбирается наиболее часто,
    но если необходимы некоторые конкретные величины, то можно
    указать их здесь. Следующие три аргумента определяют
    дескрипторы соответственно родительского окна, меню и самого
```

приложения. Последний (одиннадцатый) аргумент позволяет ассоциировать с окном некоторые дополнительные данные. Функция возвращает дескриптор созданного окна ОС Windows.

После возврата из функции CreateWindow( ) ОС Windows записывает в свою внутреннюю базу данных информацию, необходимую для сопровождения данного конкретного окна. Но при этом окно на экране не появляется — требуется вызов оставшихся двух функций, благодаря которым полностью оформленное окно появляется на экране.

В вызове функции CreateWindow( ) остался еще один (третий) аргумент — стиль окна, который, ввиду важности, надо рассмотреть более подробно.

Существует всего три основных типа окон — перекрывающиеся, всплывающие и дочерние, из которых программист может создавать множество самых разнообразных объектов, комбинируя предопределенные биты стиля. Рассмотрим эти биты, их влияние на внешний вид и некоторые свойства окон:

WS_BORDER

Окно имеет рамку без заголовка

WS_CAPTION

Окно имеет заголовок и рамку. Как правило, этот стиль используется для перекрывающихся окон и не может применяться совместно со стилем WS_DLGFRAME

WS_CHILD или WS_CHILDWINDOW

Создаваемое окно является дочерним. Не может использоваться совместно со стилем WS_POPUP

WS_CLIPCHILDREN

Исключает область, занятую дочерним окном из области рисования родительского окна, и используется только для родительских окон

WS_CLIPSIBLINGS

Исключает все другие дочерние окна из своей области рисования. Другими словами, если дочерние окна перекрываются, а этот стиль не указан, то при изменении рабочей области одного из окон могут быть испорчены рабочие области других дочерних окон. Этот стиль используется только вместе со стилем WS_CHILD

WS_DISABLED

Создается неактивное окно, т. е. сразу после создания оно не доступно для ввода с клавиатуры или с помощью мыши

WS_DLGFRAME

Окно имеет двойную рамку и не имеет заголовка

WS_GROUP

Определяет первый элемент управления группы окон, к которым пользователь может переходить при помощи клавиш со стрелками.

Все элементы управления, определенные с этим стилем, заканчивают текущую и начинают новую группу (одна группа заканчивается там, где начинается другая)

WS_HSCROLL

Окно имеет горизонтальную полосу прокрутки

WS_VSCROLL

Окно имеет вертикальную полосу прокрутки

WS_ICONIC или WS_MINIMIZE

Создаваемое окно будет отображено в виде пиктограммы. Используется только со стилем WS_OVERLAPPED

WS_MAXIMIZE

Создаваемое окно при отображении будет иметь максимально возможный для него размер

WS_MAXIMIZEBOX

Создаваемое окно будет иметь кнопку максимизации

WS_MINIMIZEBOX

Создаваемое окно будет иметь кнопку минимизации

WS_OVERLAPPED или WS_TILED

Создаваемое окно является перекрывающимся. Обычно имеет заголовок и рамку

WS_OVERLAPPEDWINDOW или WS_TILEDWINDOW

Перекрывающееся окно с комбинацией стилей WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX и WS_MAXIMIZEBOX

WS_POPUP

Создается всплывающее окно. Не может использоваться совместно со стилем WS_CHILD

WS_POPUPWINDOW

Всплывающее окно с комбинацией стилей WS_BORDER, WS_POPUP и WS_SYSMENU

WS_SIZEBOX или WS_THICKFRAME

Создаваемое окно имеет утолщенную рамку, при помощи которой можно изменять размер окна

WS_SYSMENU

Создает окно с кнопкой системного меню (control-menu box) в полосе заголовка. Может быть использован только для окон с полосой заголовка (необходимо дополнительно указать стиль WS_CAPTION)

WS_TABSTOP

Определяется для одного или нескольких элементов управления для того, чтобы между ними можно было перемещаться при помощи клавиши табуляции

WS_VISIBLE

```
    Окно становится видимым сразу после создания
*/

// Показать окно с дескриптором hWnd: передает в ОС Windows
//   информацию (nCmdShow) о том, в каком виде необходимо
//   отобразить окно
if ( ShowWindow( hWnd, nCmdShow ) )
    return FALSE;
/*
   Обратите внимание, что второй аргумент, указанный в вызове
функции ShowWindow( ), можно переопределить. Возможные
значения аргумента nCmdShow:
#define SW_HIDE             0
#define SW_SHOWNORMAL       1
#define SW_NORMAL           1
#define SW_SHOWMINIMIZED    2
#define SW_SHOWMAXIMIZED    3
#define SW_MAXIMIZE         3
#define SW_SHOWNOACTIVATE   4
#define SW_SHOW             5
#define SW_MINIMIZE         6
#define SW_SHOWMINNOACTIVE  7
#define SW_SHOWNA           8
#define SW_RESTORE          9
#define SW_SHOWDEFAULT      10
#define SW_FORCEMINIMIZE    11
#define SW_MAX              11
    Значения аргумента nCmdShow определяют следующие варианты
отображения окна:
SW_HIDE
    Скрыть указанное окно и активировать другое окно
SW_MINIMIZE
    Свернуть указанное окно и активировать окно верхнего уровня
(top level) в системном списке окон
SW_SHOWMINIMIZED
    Активировать и свернуть указанное окно
SW_RESTORE или SW_NORMAL или SW_SHOWNORMAL
    Активировать и показать указанное окно. Если окно свернуто
или максимизировано, то система восстанавливает его
первоначальное положение и размеры
SW_SHOW
    Активировать указанное окно и показать его с текущим
положением и размерами
```

```
    SW_MAXIMIZED
        Максимизировать указанное окно
    SW_SHOWMAXIMIZED
        Активизировать и максимизировать указанное окно
    SW_SHOWMINNOACTIVE
        Показать указанное окно свернутым в пиктограмму. Эффект этой
    константы похож на эффект от SW_SHOWMINIMIZED, за исключением
    того, что окно не активируется
    SW_SHOWNA
        Показать указанное окно с текущими положением и размерами.
    Прежнее активное окно остается активным
    SW_SHOWNOACTIVATE
        Показать указанное окно с его последним по времени
    положением и размерами. Прежнее активное окно остается активным
    */

    // Перерисовать окно: для перерисовки окна функция предписывает
    //   ОС Windows послать окну сообщение WM_PAINT
    if ( !UpdateWindow( hWnd ) )
        return FALSE;

    return TRUE;
}
```

Примечание

Приведенные описания, безусловно, полезны и нужны, но очень полезно также поэкспериментировать, создавая окна с комбинациями различных стилей, чтобы увидеть их влияние.

Теперь дадим краткую характеристику упомянутых выше основных типов окон Windows.

- *Перекрывающиеся окна* (overlapped windows). Это основной и наиболее универсальный тип окон Windows. Для их создания чаще всего используется комбинированный стиль `WS_OVERLAPPEDWINDOW`. Главное окно приложения, как правило, имеет именно этот тип.
- Вспомогательные или *всплывающие окна* (popup windows). Чаще всего этот тип окон создается с использованием стиля `WS_POPUP`. Обычно они используются для отображения какой-либо информации на короткий промежуток времени. Наиболее часто такие окна применяются для отображения диалоговых окон или окон сообщений. Основное их отличие от других окон заключается в том, что даже если они имеют родительское окно, то все равно всегда отображаются поверх всех окон на экране, выскакивая, как поплавки, наверх даже тогда, когда пользователь делает активным другое окно. Для окон этого типа, как правило, организуется своя оконная

процедура. Они могут и не иметь родителя. Если такое окно не имеет родительского, то оно является независимым от создавшего его окна и, по своим свойствам, практически неотличимо от перекрывающихся окон. Поведение всплывающего окна, имеющего родителя, зависит от того, что происходит с родительским окном. Когда главное окно минимизируется, всплывающее окно "без родителя" скрывается, а "с родителем" остается на экране сверху. Подчеркнем еще один важный момент. Когда мы говорим, что вспомогательное окно имеет родителя, то это совсем не означает, что оно является дочерним.

- *Дочерние окна* (child windows). Дочерние окна создаются тогда, когда у приложения уже есть главное окно. Такие окна связаны некоторыми характеристиками (подчинены) с тем окном, из которого они были созданы — отсюда и такое название. Назначение таких окон может быть самым разнообразным: от простого деления родительского окна на области до многодокументного интерфейса. Все элементы управления также являются дочерними окнами. Дочерние окна никогда не отображаются вне своего родительского окна ни в раскрытом виде, ни в виде пиктограммы — они целиком принадлежат родителю. Располагаются они в родительском окне относительно верхнего левого угла его рабочей (клиентской) области. Более того, при перемещении родительского окна по экрану его дочерние окна перемещаются вместе с ним. И, наконец, дочернее окно никогда не может стать активным.

2.5. Оконная процедура

Чтобы программа могла взаимодействовать с "внешним миром", необходимо это каким-либо образом обеспечить. В любом Windows-приложении такое взаимодействие обеспечивает *оконная процедура*. Она регистрируется в системе одновременно с регистрацией оконного класса и вызывается всякий раз, когда Windows выполняет какую-либо операцию над окном приложения. При каждом вызове оконной процедуры ей передается для обработки очередное сообщение, поступающее из очереди сообщений приложения.

В следующем фрагменте (листинг 2.4), являющемся одним из вариантов завершения файла FrameWnd.cpp, показан пример реализации оконной процедуры.

Листинг 2.4. Файл FrameWnd.cpp (первый вариант завершения)

```
// ************************************************************
// Оконная процедура — вариант 1.
//   Получает очередное сообщение и индивидуально обрабатывает его.
```

```
//   Вывод в клиентскую область окна не производится
LRESULT CALLBACK WndProc(           // Возвращает результат
                                    //   обработки сообщения,
                                    //   зависящий от посланного
                                    //   сообщения: 0 — успех
    HWND                hwnd,       // Дескриптор созданного окна
    UINT                message,    // Номер сообщения
    WPARAM              wParam,     // Дополнительная информация о
    LPARAM              lParam )    //   сообщении, зависящая от
                                    //   типа сообщения (wParam,
                                    //   lParam)
{
    // Обработчик сообщения — в данном случае приложение явно
    //   отвечает только на сообщение WM_DESTROY, все остальные
    //   сообщения передаются в DefWindowProc( ) — функцию,
    //   управляющую поведением окна по умолчанию
    switch( message )
    {
    case WM_DESTROY:

        // Указывает системе, что сделан запрос о завершении
        //   приложения: 0 — код завершения. Вызов этой функции
        //   обычно используется в ответ на поступившее сообщение
        //   WM_DESTROY
        PostQuitMessage( 0 );
        break;

    default:

        // Обработка сообщения по умолчанию
        return DefWindowProc( hwnd, message, wParam, lParam );

    }

    return 0;
}
```

Windows "знает", что имя этой процедуры указано в поле `lpfnWndProc` структуры типа `WNDCLASS`, описывающей класс окна. Все окна, созданные на базе этого класса, будут управляться процедурой `WndProc`.

В Windows существует более двухсот различных оконных сообщений, которые могут посылаться окнам. Все они имеют префикс `WM_` (от англ. *window message* — оконное сообщение), например: `WM_CREATE`, `WM_SIZE`, `WM_PAINT` и др.

Первым параметром процедуры `WndProc` является дескриптор окна, которому высылается сообщение. Поскольку одна процедура может обрабатывать сообщения сразу нескольких окон, этот дескриптор позволяет определить, какому именно окну адресовано сообщение. Второй параметр содержит идентификатор (номер) сообщения. Два оставшихся параметра, `wParam` и `lParam`, несут дополнительную информацию, зависящую от конкретного сообщения.

Обработка самих сообщений осуществляется с помощью стандартной конструкции `switch`, которая может быть довольно большой — это зависит от числа обрабатываемых пользователем сообщений.

В нашем простейшем случае приложение явно отвечает только на сообщение `WM_DESTROY`, а все остальные сообщения передаются в функцию `DefWindowProc`, управляющую поведением окна по умолчанию. Она, по умолчанию, умеет обрабатывать почти все виды сообщений, ассоциированные с конкретным типом окна. Например, она "знает" как обрабатывать действия мыши, изменяющие форму и местоположение окна.

Однако функция `DefWindowProc` не обрабатывает сообщение `WM_DESTROY`. Поэтому мы должны сами обработать это сообщение. Когда пользователь выбирает в системном меню команду **Close** (Закрыть) или нажимает кнопку закрытия окна, ОС Windows посылает приложению сообщение `WM_DESTROY`. В ответ на это приложение завершает свою работу, вызывая функцию `PostQuitMessage( )`, которая направляет в очередь сообщений приложения сообщение `WM_QUIT`, вследствие чего цикл обработки сообщений и работа приложения завершаются.

2.6. Сообщение *WM_PAINT*. Вывод на экран текстовой и графической информации

В каких случаях Windows-приложению необходимо осуществлять вывод? Например, при изменении данных, чтобы пользователь мог это увидеть. В Windows вывод должен осуществляться еще и в следующем ряде случаев:

- в момент создания окна;
- при изменении размеров или местоположения окна;
- при появлении окна или его части из-за другого окна;
- при минимизации или максимизации окна;
- при отображении данных из только что открытого файла;
- при прокрутке, изменении или выборе данных, находящихся в окне и т. д.

Во многих случаях Windows берет на себя инициирование процесса отображения, отмечая область перерисовки и посылая функции соответствующего окна сообщение `WM_PAINT`. Это сообщение обрабатывает само приложение, осуществляя вывод данных в окно.

В ряде случаев приложение, изменив данные, может инициировать процесс вывода самостоятельно, используя то же сообщение. Однако может оказаться необходимым осуществлять вывод, продолжая выполнение некоторых действий. Характерный пример такого рода — чтение из файла с одновременным выводом обработанных данных в окно. В этом случае приложение может осуществлять вывод непосредственно в окно, а не посылать сообщение `WM_PAINT` самому себе.

В любом случае, Windows-приложению, для начала вывода, необходимо получить *дескриптор контекста* устройства вывода. Для этого можно использовать функцию `BeginPaint`, с помощью полученного дескриптора выполнить вывод в окно и освободить контекст устройства путем вызова функции `EndPaint`. Что же такое контекст устройства?

2.6.1. Контекст устройства

Одна из основных особенностей Windows API — независимость вывода от конкретной модели устройства. Программное обеспечение, которое поддерживает эту независимость, содержится в двух библиотеках динамической компоновки. Первая, GDI.DLL, обеспечивает графический интерфейс устройства (GDI, Graphics Device Interface), вторая — является драйвером устройства. Использование того или иного драйвера зависит, естественно, от конкретной разновидности устройства вывода.

Перед операцией вывода на некоторое устройство приложение должно запросить GDI о загрузке соответствующего драйвера (обычно это происходит автоматически и не требует от программиста дополнительных усилий по программированию). После загрузки драйвера приложение настраивает все параметры вывода. Windows обеспечивает хранение всех необходимых параметров в специальной структуре, называемой *контекстом устройства*. Эта структура определяет набор графических объектов, связанных с ними атрибутов и графических режимов, которые собственно и воздействуют на вывод.

Одним из контекстов устройств Windows API является *контекст экрана*. Используя контекст устройства, приложение может осуществлять следующий набор операций:

- перечисление графических объектов;
- выбор новых графических объектов;
- удаление графических объектов;

- сохранение графических объектов, их атрибутов и графических режимов;
- восстановление графических объектов, их атрибутов и графических режимов.

Windows API предоставляет для экрана три типа контекстов: контекст класса, общий и частный контексты.

Контекст класса поддерживается только для совместимости с предыдущими версиями Windows. По этой причине контекст класса экрана далее не рассматривается.

Частные контексты экрана следует использовать в прикладных программах, которые выполняют многочисленные операции рисования. Это обеспечит повышение их быстродействия. Приложение создает частный контекст устройства, определяя стиль `CS_OWNDC` для класса окна в момент инициализации структуры `WNDCLASS`, с последующей регистрацией класса окна, с помощью функции `RegisterClass`.

Общие контексты экрана используются обычно в приложениях, которые лишь время от времени выполняют операции рисования.

Приложение получает общий или частный контекст экрана, вызывая одну из функций: `BeginPaint`, `GetDC` или `GetDCEx`. Windows инициализирует общие контексты экрана значениями по умолчанию, которые можно менять, по мере необходимости, при помощи соответствующих функций Windows API. Количество общих контекстов экрана ограничено и, следовательно, по завершении операций вывода необходимо освободить их, сообщив об этом системе вызовом функции `EndPaint` или `ReleaseDC`. В результате вызова одной из этих функций изменения параметров контекста, которые были произведены приложением, будут отменены. Таким образом, параметры отображения необходимо устанавливать каждый раз. Частные контексты устройства отличаются от общих тем, что сохраняют любые изменения для заданных по умолчанию данных, даже после вызова функции `EndPaint` или `ReleaseDC`. Частные контексты устройства лишь однократно инициализируются значениями по умолчанию. Они автоматически удаляются после разрушения последнего окна класса.

2.6.2. Вывод на экран текстовой и графической информации

В листинге 2.5 показан еще один возможный пример реализации оконной процедуры, являющийся более сложным вариантом завершения файла FrameWnd.cpp. Этот пример иллюстрирует обработку сообщения `WM_PAINT`, обеспечивающую вывод в окно приложения текста и графических объек-

тов — линии, эллипса и прямоугольника с изменением цвета и фона символов текста, параметров пера и кисти.

Листинг 2.5. Файл FrameWnd.cpp (второй вариант завершения)

```
// ****************************************************************
// Оконная процедура — вариант 2
//   Получает очередное сообщение и индивидуально обрабатывает его.
//   Обрабатывается вывод в клиентскую область окна
LRESULT CALLBACK WndProc(           // Возвращает результат
                                    //   обработки сообщения,
                                    //   зависящий от посланного
                                    //   сообщения: 0 — успех
    HWND                hwnd,       // Дескриптор созданного окна
    UINT                message,    // Номер сообщения
    WPARAM              wParam,     // Дополнительная информация о
    LPARAM              lParam )    //   сообщении, зависящая от
                                    //   типа сообщения (wParam,
                                    //   lParam)
{
    // Дескриптор контекста
    HDC                 hDC;
    // Указатель на структуру с информацией для приложения о
    //   клиентской области окна, используется для вывода в окно
    PAINTSTRUCT         ps;

    // Обработчик сообщения — в данном случае приложение явно
    //   отвечает только на сообщения WM_DESTROY и WM_PAINT, все
    //   остальные сообщения передаются в DefWindowProc( ) —
    //   функцию, управляющую поведением окна по умолчанию
    switch( message )
    {
    case WM_PAINT:

        COLORREF        oldColor,   // Старый цвет символов
                        oldBkColor; // Старый цвет фона символов
        HBRUSH          hNewBrush,  // Новая кисть
                        hOldBrush;  // Старая кисть
        HPEN            hNewPen,    // Новое перо
                        hOldPen;    // Старое перо

        // Для любого вывода в окно ОС Windows необходимо
        //   использовать функции GDI (Graphics Device Inerface),
```

```
//   которые в качестве параметра используют контекст
//   устройства. Обычно контекст устройства получают с
//   помощью фунции BeginPaint( )
hDC = BeginPaint( hwnd, &ps );
if ( !hDC )
{
    MessageBox( NULL, "Контекст устройства не получен",
                "Ошибка 1", MB_OK );
    exit( 1 );
}

// Вывод заданного текста (четвертый аргумент, 27 — длина
//   текста) в определенное место окна (второй и третий
//   аргументы). Перед выводом текста задается цвет
//   символов (зеленый) и цвет фона(черный). После вывода
//   текста восстанавливаются прежние значения указанных
//   параметров. Здесь макрос RGB(0, 255, 0) задает тройку
//   чисел, определяющих цвет — первое число определяет
//   значение красной составляющей, второе — зеленой и
//   третье — синей
oldColor = SetTextColor( hDC, RGB(0, 255, 0) );
if ( oldColor == CLR_INVALID )
{
    MessageBox( NULL, "Ошибка SetTextColor",
                "Ошибка 2", MB_OK );
    exit( 2 );
}
oldBkColor = SetBkColor( hDC, RGB(0, 0, 0) );
if ( oldBkColor == CLR_INVALID )
{
    MessageBox( NULL, "Ошибка SetBkColor",
                "Ошибка 3", MB_OK );
    exit( 3 );
}
if ( !TextOut( hDC, 150, 0, "Пример вывода текста в окно",
     27 ) )
{
    MessageBox( NULL, "Неверное использование TextOut",
                "Ошибка 4", MB_OK );
    exit( 4 );
}
SetTextColor( hDC, oldColor );
SetBkColor( hDC, oldBkColor );
```

```
// Об обработке ошибок этих функций см. выше — для
//   сокращения размера исходного текста она опущена

// Рисование линии: вначале выводится поясняющий текст
//   (используется текущий цвет и фон), а затем — рисуется
//   линия. Линия рисуется цветом текущего пера на текущем
//   фоне. В функции MoveToEx( ) (125, 50)  — новая текущая
//   позиция в окне, а NULL означает, что старая текущая
//   позиция в окне не запоминается. Функция LineTo( )
//   рисует линию из текущей позиции до точки (175, 100)
// Об обработке ошибки этой функции см. выше — для
//   сокращения размера исходного текста она опущена
TextOut( hDC, 50, 30, "Пример вывода линии в окно", 26 );
if ( !MoveToEx( hDC, 125, 50, NULL ) )
{
    MessageBox( NULL, "Неверное использование MoveToEx",
                "Ошибка 5", MB_OK );
    exit( 5 );
}
LineTo( hDC, 175, 100 );
if ( !LineTo( hDC, 175, 100 ) )
{
    MessageBox( NULL, "Неверное использование LineTo",
                "Ошибка 6", MB_OK );
    exit( 6 );
}

// Рисование эллипса: вначале выводится поясняющий текст
//   (используется текущий цвет и фон), а затем — рисуется
//   эллипс. В функции Ellipse( ) (100, 140) и (200, 240) —
//   координаты левого верхнего и правого нижнего углов
//   прямоугольника, в который вписан эллипс. Эллипс
//   рисуется существующим пером, а внутренняя область
//   эллипса закрашивается новой кистью (красной). После
//   рисования старая кисть восстанавливается
// Об обработке ошибки этой функции см. выше — для
//   сокращения размера исходного текста она опущена
TextOut( hDC, 50, 120, "Пример вывода эллипса в окно",
         28 );
// Создаем зеленую кисть
hNewBrush = CreateSolidBrush( RGB(255, 0, 0) );
if ( !hNewBrush )
{
```

```
    MessageBox( NULL, "Неверное использование "
                "CreateSolidBrush", "Ошибка 7", MB_OK );
    exit( 7 );
}
// Включаем ее в контекст
hOldBrush = static_cast< HBRUSH >( SelectObject( hDC,
    hNewBrush ) );
if ( !hOldBrush )
{
    MessageBox( NULL, "Неверное использование "
                "SelectObject", "Ошибка 8", MB_OK );
    exit( 8 );
}
if ( !Ellipse( hDC, 100, 140, 200, 240 ) )
{
    MessageBox( NULL, "Неверное использование "
                "Ellipse", "Ошибка 9", MB_OK );
    exit( 9 );
}
// Удаляем новую кисть
if ( !DeleteObject( hNewBrush ) )
{
    MessageBox( NULL, "Неверное использование "
                "DeleteObject", "Ошибка 10", MB_OK );
    exit( 10 );
}
// Восстанавливаем старую кисть. Об обработке ошибки этой
//   функции см. выше — для сокращения размера исходного
//   текста она опущена
SelectObject( hDC, hOldBrush );

// Рисование прямоугольника: вначале выводится поясняющий
//   текст, а затем — рисуется прямоугольник. Перед
//   рисованием прямоугольника задаются параметры пера для
//   рисования прямоугольника. В функции Rectangle( )
//   (100, 270) и (200, 370)  — координаты левого верхнего
//   и правого нижнего углов прямоугольника. Прямоугольник
//   закрашивается текущей кистью. После рисования старое
//   перо восстанавливается.
// Об обработке ошибки этой функции см. выше — для
//   сокращения размера исходного текста она опущена
TextOut( hDC, 50, 250, "Пример вывода прямоугольника"
         " в окно", 35 );
```

```
    // Создаем новое перо: точечное, толщиной 1, красного цвета
    hNewPen = CreatePen( PS_DOT, 1, RGB(255, 0, 0) );
    if ( !hNewPen )
    {
        MessageBox( NULL, "Неверное использование "
                    "CreatePen", "Ошибка 11", MB_OK );
        exit( 11 );
    }
    // Включаем его в контекст. Об обработке ошибки этой
    //   функции см. выше — для сокращения размера исходного
    //   текста она опущена
    hOldPen =
        static_cast< HPEN >( SelectObject( hDC, hNewPen ) );
    if ( !Rectangle( hDC, 100, 270, 200, 370 ) )
    {
        MessageBox( NULL, "Неверное использование "
                    "Rectangle", "Ошибка 12", MB_OK );
        exit( 12 );
    }
    // Удаляем новое перо. Об обработке ошибки этой функции см.
    //   выше — для сокращения размера исходного текста она
    //   опущена
    DeleteObject( hNewPen );
    // Восстанавливаем старое перо. Об обработке ошибки этой
    //   функции см. выше — для сокращения размера исходного
    //   текста она опущена
    SelectObject( hDC, hOldPen  );

    // Вывод информации в окно завершается вызовом функции
    //   EndPaint( ). Для каждого вызова BeginPaint( ) должен
    //   существовать соответствующий вызов EndPaint( )
    if ( !EndPaint( hwnd, &ps ) )
    {
        MessageBox( NULL, "Контекст устройства не освобожден",
                    "Ошибка 13", MB_OK );
        exit( 13 );
    }
    break;

case WM_DESTROY:

    // Указывает системе, что сделан запрос о завершении
    //   приложения: 0 — код завершения. Вызов этой функции
```

```
        //   обычно используется в ответ на поступившее сообщение
        //   WM_DESTROY
        PostQuitMessage( 0 );
        break;

    default:

        // Обработка сообщения по умолчанию
        return DefWindowProc( hwnd, message, wParam, lParam );

    }

    return static_cast< LRESULT >( 0 );
}
```

Совет

Для хорошего усвоения рассмотренного материала весьма полезно будет поэкспериментировать с этим кодом. Для рисования многоугольников существует функция GDI с прототипом `BOOL Polygon( HDC hdc, const POINT *lpPoints, int nCount )`. Изучите ее, пользуясь справочной системой, и используйте для вывода на экран закрашенных многоугольников.

Внимание!

Подробные рекомендации по использованию справочной системы приведены в *разд. П3.7 приложения 3*.

2.7. Создание нового проекта FrameWnd на основе Windows API. Использование прекомпиляции

Многие стандартные заголовочные файлы, и в частности использованный в проекте FrameWnd на основе Windows API заголовочный файл afxwin.h, имеют большие размеры (десятки тысяч строк) и в процессе отладки на их компиляцию тратится большое время. Вместе с тем, стандартные заголовочные файлы не подлежат изменению и можно такой файл (такие файлы) компилировать однократно, результаты компиляции поместить во вспомогательную папку Debug или Release и многократно использовать их при отладке. Для включения подобного механизма, называемого *прекомпиляцией*, необходимо задать соответствующие свойства проекта (об этом будет сказано далее) и включить в проект файлы StdAfx.h и StdAfx.cpp (листинги 2.6 и 2.7 соответственно).

Листинг 2.6. Файл StdAfx.h

```
/*
   Файл                : StdAfx.h

   Проект              : любой проект, использующий стандартные
                         функции API-SDK

   Назначение          : стандартный включаемый файл для
                         прекомпиляции (результаты компиляции в
                         папке Debug в файле .PCH)

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _StdAfx_h
    #define _StdAfx_h

    // Здесь следует подключить все стандартные (неизменяемые)
    //   заголовочные файлы большого размера

    // Заголовочный файл для низкоуровневых Windows-приложений
    #include <windows.h>

#endif
```

Листинг 2.7. Файл StdAfx.cpp

```
/*
   Файл                : StdAfx.cpp

   Проект              : любой проект, использующий стандартные
                         функции API-SDK

   Назначение          : стандартный включаемый
                         файл для прекомпиляции (результаты
                         компиляции в папке Debug в файле .PCH)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAFX.h"
```

Как указывалось ранее, для хорошего усвоения рассмотренного материала весьма полезно будет поэкспериментировать с проектом FrameWnd на основе Windows API. Для этой цели создайте копию проекта. Запустите IDE Microsoft Visual Studio C++ .NET 2005. В поле **Resent Projects** (Последние проекты) вкладки **Start Page** (Стартовая страница) (эту вкладку, если ее нет, можно сделать видимой, выбрав пункт меню **View** (Просмотр) **Other Windows** (Другие окна) | **Start Page** (Стартовая страница)) выберите **Create: Project...** (Создать: Проект). В результате на экране появится диалоговое окно **New Project** (Новый проект).

Здесь вы должны выбрать тип создаваемого приложения — в поле **Project Types** (Тип проекта) выберите **Win32**, в поле **Templates** (Шаблоны) — **Win32 Project**. Укажите путь для нового проекта в поле **Location** (Местоположение). Это можно сделать, набрав путь вручную или воспользовавшись расположенной справа кнопкой **Browse...** (Указать). Разумеется, что соответствующий подкаталог должен быть предварительно создан. Укажите в поле **Name** (Имя) имя проекта FrameWnd. После выполнения указанных действий для создания проекта нажмите кнопку **OK**, в результате чего на экране появится диалоговое окно мастера создания приложения **Win32 Application Wizard — FrameWnd** (Мастер создания Win32-приложений).

В этом окне выберите вкладку **Application Settings** (Установки приложения), выберите опцию **Empty Project** (Пустой проект) и нажмите кнопку **Finish** (Завершить).

В окне **Solution Explorer — FrameWnd** (Обзор проекта), для проекта **FrameWnd**, правой кнопкой мыши вызовите контекстное меню и выполните команду **Properties** (Свойства). В поле **Character Set** (Символы) вкладки **General** (Основное) выберите **Use Multi-Byte Character Set** и нажмите кнопку **OK**.

Чтобы наполнить созданный проект, необходимо добавить в него файлы, содержащие текст программы. В окне **Solution Explorer — FrameWnd** для проекта FrameWnd правой кнопкой мыши вызовите контекстное меню и выполните команду **Add** (Добавить) | **New Item...** (Новый элемент). В появившемся окне **Add | New Item — FrameWnd**, в поле **Categories** (Категория) выберите **Code** (Код), в поле **Templates** (Шаблоны) выберите **C++ File (.cpp)**, а в поле **Name** (Имя) задайте имя файла FrameWnd и нажмите кнопку **Add**. Аналогичным образом добавьте в проект пустые файлы StdAfx.cpp и StdAfx.h. В добавленные пустые файлы введите исходные тексты, в соответствии с листингами 2.1—2.7 (или скопируйте в них содержимое соответствующих файлов из папки \Глава 02\FrameWnd на прилагаемом компакт-диске).

Чтобы обеспечить прекомпиляцию стандартных заголовочных файлов, выполните следующие действия. В окне **Solution Explorer — FrameWnd** пра-

вой кнопкой мыши вызовите контекстное меню и выберите пункт **Properties**. В появившемся окне **FrameWnd Property Pages** откройте вкладку **C/C++**, выберите категорию **Precompiled Headers** (Прекомпиляция) и в поле **Create/Use Precompiled Headers** (Создать/Использовать прекомпиляцию) выберите **Use Precompiled Headers (/Yu)**. Нажмите кнопку **OK**. Аналогично, в окне **Solution Explorer — FrameWnd**, для файла StdAfx.cpp, правой кнопкой мыши вызовите контекстное меню и выберите **Properties**. В появившемся окне **StdAfx.cpp Property Pages** откройте вкладку **C/C++**, выберите категорию **Precompiled Headers**, в поле **Create/Use Precompiled Headers** выберите **Create Precompiled Headers (/Yc)** и нажмите кнопку **OK**.

В заключение, выполните последовательную компиляцию файлов StdAfx.cpp и FrameWnd.cpp, командой **Build FrameWnd** (Построить проект) постройте проект и запустите его командой **Start Without Debugging** (Запустить без отладки) (рис. 2.1).

Как указывалось ранее, папка проекта FrameWnd для IDE Microsoft Visual Studio C++ .NET 2005, использующего Windows API, имеется на прилагаемом компакт-диске (в папке \Глава 02\FrameWnd).

Замечание

Более подробно создание нового проекта Windows-приложения, использующего Windows SDK и API, описано в *приложении 3*.

Рис. 2.1. Главное окно приложения

Примечание

Наряду с только что рассмотренным способом повторного создания проекта, при наличии готовых исходных файлов можно использовать, для экспериментов, более "быстрые" способы создания копии проекта. Первый из таких способов заключается в создании пустого проекта, копировании в папку проекта готовых файлов и включении их в проект. В рамках второго способа папка проекта просто копируется с компакт-диска.

2.8. Вопросы и упражнения для самопроверки

Для самопроверки ответьте на следующие вопросы:

1. С какой целью рассматриваются проекты Windows-приложений, использующих Windows API?
2. Укажите возможные способы создания копии проекта Windows-приложения FrameWnd, использующего Windows API.
3. Опишите наиболее часто используемые типы данных Windows.
4. Перечислите наиболее часто используемые структуры данных Windows и укажите их назначение.
5. Перечислите и дайте краткую характеристику функциональных частей функции `WinMain` Windows-приложения, использующего Windows API.
6. Опишите работу стандартного цикла обработки сообщений.
7. Опишите основные свойства оконного класса (поля структуры `WNDCLASS`).
8. В чем заключается регистрация класса окна? Можно ли для одного и того же класса окна создавать несколько экземпляров окон?
9. Что нужно сделать для создания конкретного окна? Что является параметрами конкретного окна?
10. Перечислите и дайте краткую характеристику основных типов окон Windows.
11. Что такое оконная процедура? Укажите, как она устроена и работает.
12. Что такое контекст экрана? Перечислите и охарактеризуйте разновидности контекста экрана.
13. Когда нужно работать с контекстом экрана? Как следует работать с контекстом экрана?
14. Перечислите основные стандартные функции для рисования на экране графических примитивов.

15. Зачем нужна прекомпиляция? Как задать прекомпиляцию в проекте Windows-приложения, использующего Windows API?

Ответы на вопросы можно проверить — они приведены в *разд. П1.2 приложения 1*.

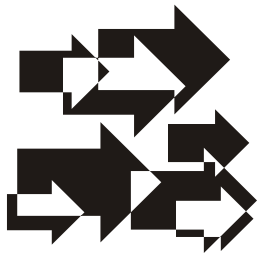
Совет

Одним из указанных выше способов создайте копию проекта Windows-приложения, использующего Windows API.

При экспериментах с созданной копией проекта выполните следующие упражнения:

1. Задавая различные стили класса окна и их комбинации (константы, задающие стили, начинаются с префикса `CS_`), изучите их влияние на свойства окна.
2. Попробуйте использовать другие значки (пиктограммы) приложения (константы, задающие стандартные значки, начинаются с префикса `IDI_`).
3. Попробуйте использовать другие курсоры приложения (константы, задающие стандартные курсоры, начинаются с префикса `IDC_`).
4. Попробуйте использовать другие значения цвета фона приложения (константы, задающие цвет фона, начинаются с префикса `COLOR_`).
5. Попробуйте изменить заголовок, местоположение и размеры окна.
6. Задавая различные стили окна и их комбинации (константы, задающие стили, начинаются с префикса `WS_`), изучите их влияние на свойства окна.
7. Задавая различные варианты начального отображения окна — различные значения аргумента `nCmdShow` в вызове функции `ShowWindow( hWnd, nCmdShow )` (константы, задающие вид начального отображения, начинаются с префикса `SW_`), изучите их влияние на свойства окна.
8. Реализуйте вывод в окно различных текстов и других графических объектов, варьируя цвета символов и фона, характеристики перьев и кистей для закрашивания замкнутых контуров.
9. Нарисуйте в окне некоторую цветную, содержательную картинку.

ГЛАВА 3



Основы библиотеки классов MFC [3]

Итак, из *главы 2* следует, что создание даже самых простых Windows-приложений с использованием стандартных функций API является довольно сложной задачей, для решения которой требуется много времени и труда. Для *снижения* указанных затрат Microsoft, в составе Visual Studio, предлагает мощную *библиотеку классов MFC* (Microsoft Foundation Classes), предназначенную для разработки оконных приложений, отвечающих всем стандартам и требованиям сегодняшнего дня.

Примечание

Разработка приложений на базе MFC обеспечивает существенное снижение затрат времени, и это хорошо. Но при этом приложение, по сравнению с использованием Windows API, проигрывает в быстродействии и гибкости. Поэтому можно ожидать, что программист, пишущий под Windows и начавший писать с помощью MFC, позже перейдет к использованию Windows API или к комбинации MFC и Windows API.

Библиотека MFC предоставляет программистам удобные классы объектов, в которых инкапсулированы все наиболее важные структуры и функции API. Библиотека классов MFC предоставляет большие возможности и значительно проще в использовании, чем функции API.

Использование библиотеки классов MFC обеспечивает следующие *основные преимущества*:

- устраняются конфликты, связанные с совпадением имен стандартных и обычных функций и переменных;
- код и данные инкапсулируются в классах, доступ к ним можно ограничить;
- обеспечивается наследование возможностей базовых классов;
- размер исходного кода, который пишет программист, сокращается.

3.1. Принципы работы и ключевые особенности библиотеки классов MFC

Создатели библиотеки классов MFC при разработке данного продукта придерживались строгих правил и стандартов, базирующихся на следующих *основных принципах*:

- возможность *комбинировать* вызовы стандартных функций с использованием методов классов. Это означает, что в одной программе могут применяться как библиотека классов MFC, так и функции API;
- *баланс между производительностью и эффективностью* базовых и производных классов. Создатели библиотеки классов MFC осознавали, что от совершенства ее программного кода зависит эффективность работы основанного на ее использовании приложения. Четко соблюдались требования к размеру классов и быстроте выполнения их методов. В результате, по скорости работы классы библиотеки MFC незначительно уступают библиотечным функциям языка C/C++;
- *преемственность* при переходе от использования функций API к библиотеке классов MFC. Одна из поставленных перед разработчиками библиотеки классов MFC задач состояла в том, чтобы программисты, уже знакомые с API, не заучивали новый, далеко не малый набор имен функций и констант. Это требование строго соблюдалось при именовании членов классов, что выгодно отличает библиотеку классов MFC от других библиотек классов;
- *простота переноса* библиотеки классов MFC на разные операционные платформы — от Windows 95 до Windows 2000 и Windows XP;
- *органичная взаимосвязь* с традиционными средствами программирования на языке C++, позволяющая избежать чрезмерного усложнения программного кода.

Еще одно важное свойство, на которое обращали внимание разработчики Microsoft, — это возможность непосредственного использования базовых классов в программных проектах Windows-приложений. Библиотеки классов, существовавшие до MFC, были чересчур абстрактными. В Microsoft их называли "тяжелыми классами", поскольку основанные на них приложения отличались большим размером программного кода и недостаточно быстро выполнялись. Разработчикам библиотеки классов MFC удалось найти золотую середину между разумным уровнем абстрактности и размером кода.

Библиотека классов MFC обладает следующими *основными особенностями*:

- *расширенная система обработки исключительных ситуаций*, благодаря которой приложения менее чувствительны к ошибкам и сбоям;

- *улучшенная система диагностики*, позволяющая записывать в файл информацию об используемых объектах и контролировать значения членов классов;
- *полная поддержка* всех функций API, элементов управления, сообщений, GDI, графических примитивов, меню и диалоговых окон;
- *возможность определить тип объекта во время выполнения программы*. Это позволяет осуществлять динамическое управление членами классов в случае использования различных экземпляров класса;
- все *сообщения* связываются с *обработчиками* внутри классов, что некоторым образом уменьшает количество ошибок и затраты на поиск и исправление;
- *небольшой размер* и *быстрота выполнения* (но по этим показателям библиотека классов MFC незначительно уступает стандартным библиотечным функциям языка C);
- *поддержка модели компонентных объектов* COM (Component Object Model);
- использование тех же *соглашений об именовании* методов классов, которые применялись при подборе имен функций API. Это существенно облегчает идентификацию действий, выполняемых классами.

3.2. Иерархия классов библиотеки MFC

Большинство библиотек классов, в том числе и библиотека MFC, имеют какой-нибудь общий родительский класс, от которого порождаются все остальные. В библиотеке классов MFC таким классом является класс `CObject`. Объявление и реализацию этого класса можно посмотреть в стандартном заголовочном файле MFC:
<диск:>\Program Files\Microsoft Visual Studio 8\Vc\altmfc\include\afx.h.

Библиотека классов MFC поставляется вместе с исходными текстами классов. Иерархия классов библиотеки MFC приведена, например, в [3, стр. 426 — 430]. Отметим, что некоторые классы библиотеки MFC не порождены от базового класса `CObject` и являются самостоятельными.

Примечание

Самые точные и полные сведения о составе и иерархии классов библиотеки MFC можно получить из встроенной справочной системы IDE Microsoft Visual Studio C++ .NET 2005. Для этого в IDE достаточно выполнить команду **Help | Index** и в поле **Look for:** появившегося окна **MFC Reference — Microsoft Visual Studio 2005 Documentation — Microsoft Document Explorer** набрать **MFC, hierarhy** и посмотреть вкладки **Hierarhy Chart** и **Hierarhy Chart Categories**.

3.3. Соглашение об именах библиотеки классов MFC

В качестве *префикса*, обозначающего имя класса, библиотека классов MFC использует заглавную букву `C` (от слова *Class* — класс), за которой идет имя, характеризующее назначение класса. Например, `CWinApp` — класс, определяющий приложение, `CWnd` — базовый класс всех оконных объектов, `CDialog` — класс окон диалога и т. д.

Для *имен методов классов* используются три способа. При *первом способе* имя объединяет глагол и существительное, например, `LoadIcon` (загрузить пиктограмму) или `DrawText` (нарисовать текст). При *втором способе* имя метода состоит только из существительных, например, `DialogBox` (блок диалогового окна). Для методов, предназначенных для преобразования одного типа в другой, обычными являются такие имена, как `XtoY` (из X в Y).

Для данных-членов классов библиотеки MFC принят следующий *способ назначения имен*: обязательный префикс `m_` (от *member* — элемент или член класса), за которым идет префикс, характеризующий тип данного, и завершается все именем данного. Например, `m_pMainWnd`, где `p` — префикс, обозначающий указатель.

3.4. Подключаемые файлы библиотеки классов MFC

В любом из файлов Windows-приложения на базе библиотеки классов MFC одной из первых является строка:

```
#include <stdafx.h>
```

Этот *подключаемый* (или *заголовочный*) файл служит для подключения к программе необходимых ей библиотечных файлов и включает в себя все наиболее часто используемые заголовочные файлы. Это и файл afxwin.h, который содержит описание основных классов библиотеки MFC и сводит воедино все включаемые файлы, необходимые для базового функционирования MFC (ядро MFC и стандартные компоненты). Это и файл afxcmn.h (общие управляющие элементы MFC), файл afxext.h, содержащий описание классов общего назначения, макросы, базовые типы данных MFC (классы расширений MFC), файл afxmt.h (многопоточность MFC) и файл afxres.h (ресурсы MFC). При разработке больших приложений нам обязательно потребуется использование заранее откомпилированных заголовочных файлов (прекомпиляция), что позволит существенно сократить время компиляции. Файл stdafx.h как раз и выступает в такой роли.

Здесь же отметим изменения, которые претерпел основной включаемый файл Windows — файл windows.h. Как и раньше, каждое Windows-приложение должно содержать этот файл. Отличие же заключается в том, что в своем последнем виде он уже не представляет собой огромного "монстра", а разбит на 35 сравнительно небольших файлов, в том числе:

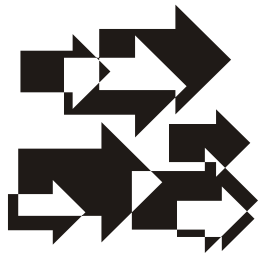
- windef.h — содержит определения базовых типов;
- winbase.h — описывает базовые функции API;
- wingdi.h и winuser.h — включают определения процедур соответственно для модулей GDI и USER, а также определения констант и макросов для этих модулей системы.

3.5. Вопросы для самопроверки

1. Для чего нужна библиотека классов MFC, что она обеспечивает?
2. Какие основные преимущества обеспечивает использование библиотеки классов MFC?
3. Каких основных принципов придерживались разработчики библиотеки классов MFC?
4. Перечислите основные особенности библиотеки классов MFC.
5. Как можно получить самые точные и полные сведения о составе и иерархии классов библиотеки MFC?
6. Какие соглашения об именах приняты в библиотеке классов MFC?
7. Укажите особенности работы со стандартными включаемыми файлами библиотеки классов MFC.

Ответы на вопросы можно проверить — они приведены в *разд. П1.3 приложения 1*.

ГЛАВА 4



Проектирование оконных приложений на базе библиотеки классов MFC [3, 4]

Прежде чем приступить к созданию более сложных Windows-приложений, рассмотрим простое оконное приложение, аналогичное по возможностям приведенному ранее, в *главе 2*, проекту FrameWnd. Это сделает возможным сопоставление двух *технологий создания оконных приложений* — на базе Windows API и с использованием библиотеки классов MFC, что позволит лучше понять анатомию высокоуровневых оконных приложений.

Сначала приведем полный исходный текст демонстрационного примера FrameWnd, а затем проведем его анализ.

4.1. Простое оконное приложение на базе библиотеки классов MFC

Программный проект простого оконного приложения FrameWnd, использующего библиотеку классов MFC, содержится на прилагаемом компакт-диске в папке \Глава 04\FrameWnd, а описание создания проекта приведено в *разд. П3.2 приложения 3*. Данное приложение можно рассматривать как основу любого оконного приложения на базе MFC.

Исходный код демонстрационного примера FrameWnd приведен в листингах 4.1—4.5. Пусть вас не смущает то, что, на первых порах, многое будет не понятно. При дальнейшем анализе и обсуждении исходного кода все возникшие вопросы получат необходимые разъяснения.

Примечание

Особенностью оформления исходного кода проекта данного оконного приложения является включение важной информации в виде комментариев прямо в

них, а не в текст книги. Это делает возможным, при работе с проектом, получать ответы на возникающие вопросы более оперативно, не прибегая к чтению текста.

Совет

Исходные тексты проекта демонстрируют *профессиональное оформление*. Используйте такой стиль оформления в вашем коде.

Листинг 4.1. Файл MainThread.hpp

```
/*
   Файл                : MainThread.hpp

   Проект              : любой проект на основе библиотеки классов
                         MFC

   Назначение          : объявление класса "MainThread",
                         производного от класса "CWinApp" библиотеки
                         классов MFC (инициализация приложения перед
                         запуском)

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _MainThread_hpp
    #define _MainThread_hpp

    #include <afxwin.h>              // Ядро MFC и стандартные
                                     //   компоненты

    // Объявление класса
    class MainThread : public CWinApp
    {
        // Методы

    public:

        // Инициализация приложения перед запуском
        virtual BOOL InitInstance( void );
    };

#endif
```

Листинг 4.2. Файл MainThread.cpp

```
/*
   Файл                : MainThread.cpp

   Проект              : демонстрация создания Windows-приложения
                         на основе библиотеки классов MFC,
                         выполняющего вывод в главное окно

   Назначение          : реализация класса "MainThread",
                         производного от класса "CWinApp" библиотеки
                         классов MFC (инициализация приложения перед
                         запуском)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "MainThread.hpp"           // Объявление класса MainThread
#include "FrameWnd.hpp"             // Объявление класса FrameWnd

// Переопределение виртуальной функции для инициализации приложения
//   перед запуском
BOOL MainThread :: InitInstance( void )
{
    // Указатель на фрейм окна
    FrameWnd            *pFrame;

    // Размещение фрейма окна в динамической памяти
    pFrame = new FrameWnd;
    /*
       Макрос ASSERT_VALID (подтверждение правильности) используется
    для проверки некоторых логических условий, которые должны
    выполняться в данной точке программы. Всякий раз, когда
    логическое выражение (в нашем случае pFrame), переданное
    макросу, будет принимать значение 0 (FALSE), выполнение
    приложения будет останавливаться и на экран будет выводиться
    окно сообщения. В данном окне указывается имя исполняемого
    файла (не всегда), имя исходного файла и номер строки, в
    которой произошла ошибка
    */
    ASSERT_VALID( pFrame );         // Обработка ошибки размещения
```

```
    /*
       Альтернативное решение
    // Создание фрейма окна
    pFrame->Create(
        // Используется предопределенное OS WINDOWS имя оконного
        //   класса
        NULL,
        // Заголовок фрейма окна
        "Создание фрейма и обработка сообщений на основе MFC" );

    // Задание размеров и расположения фрейма окна
    pFrame->MoveWindow(
        // Координата левой стороны фрейма
        100,
        // Координата верхней стороны фрейма
        200,
        600,                        // Ширина фрейма
        300,                        // Высота фрейма
        FALSE );                    // Окно не перерисовывается

    // Вывод окна на экран
    pFrame->ShowWindow( SW_SHOW );
    // Перерисовка окна
    pFrame->UpdateWindow( );
    */

    // Создание и отображение на экране окна приложения -
    //   эквивалент расположенного выше закомментированного
    //   фрагмента программы
    pFrame->Create(
        // Используется предопределенное ОС Windows имя оконного
        //   класса
        NULL,
        // Заголовок фрейма окна
        "Создание фрейма и обработка сообщений на основе MFC",
        WS_VISIBLE | WS_OVERLAPPEDWINDOW,    // Стиль окна
        // Местоположение и размеры окна: 100 — левая сторона,
        //   200 — верх, 700 — правая сторона, 500 — низ окна
        CRect( 100, 200, 700, 500 ) );

    // Созданный фрейм делаем главным окном приложения
    this->m_pMainWnd = pFrame;

    return TRUE;
}
```

Листинг 4.3. Файл FrameWnd.hpp

```
/*
   Файл                : FrameWnd.hpp

   Проект              : демонстрация создания Windows-приложения
                         на основе библиотеки классов MFC,
                         выполняющего вывод в главное окно

   Назначение          : объявление класса "FrameWnd", производного
                         от класса "CFrameWnd" библиотеки классов
                         MFC (обработка сообщений главного окна)

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _FrameWnd_hpp
    #define _FrameWnd_hpp

    #include <afxwin.h>              // Ядро MFC и стандартные
                                     //   компоненты

    // Объявление класса
    class FrameWnd : public CFrameWnd
    {
        // Методы

    protected:

        // Так нужно записывать прототип функции, обрабатывающей
        //   сообщение WM_PAINT
        afx_msg void OnPaint( void );

        // Объявление таблицы сообщений
        DECLARE_MESSAGE_MAP( );
/*
   Библиотека классов MFC предоставляет альтернативу многочисленным
операторам switch, используемым в низкоуровневых Windows-
приложениях, для обработки сообщений, посылаемых окну. Взаимосвязь
сообщений и их обработчиков может быть определена таким образом,
что, когда сообщение поступает в оконную процедуру, автоматически
```

вызывается соответствующий обработчик. Реализация такой взаимосвязи основана на понятии карты (или таблицы) сообщений (message map).

Карта сообщений представляет собой механизм пересылки сообщений и команд Windows в окна, представления и другие объекты приложения, реализованного на базе MFC. Такие карты преобразуют сообщения Windows, извещения элементов управления, а также команды меню, кнопок панелей инструментов, акселераторов клавиатуры в вызовы функций соответствующих классов, которые их обрабатывают. Эта отличительная черта карты сообщений реализована по аналогии с виртуальными функциями C++, но имеет дополнительные преимущества, недоступные для них. Каждый класс, который может получить сообщение, должен иметь свою карту сообщений, чтобы иметь возможность соответствующим образом обрабатывать сообщения. При этом следует иметь в виду, что карта сообщений должна определяться вне какой-либо функции или объявления класса. Она также не может размещаться внутри блока.

Для определения карты сообщений используются три макроса: BEGIN_MESSAGE_MAP, END_MESSAGE_MAP и DECLARE_MESSAGE_MAP. Макрос DECLARE_MESSAGE_MAP располагается в конце объявления класса, использующего карту сообщений, как это было показано выше.
*/
    };

#endif

Листинг 4.4. Файл FrameWnd.cpp

```
/*
   Файл                : FrameWnd.cpp

   Проект              : демонстрация создания Windows-приложения
                         на основе библиотеки классов MFC,
                         выполняющего вывод в главное окно

   Назначение          : реализация класса "FrameWnd", производного
                         от класса "CFrameWnd" библиотеки классов
                         MFC (обработка сообщений главного окна)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "FrameWnd.hpp"             // Объявление класса FrameWnd
```

```
// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( FrameWnd, CFrameWnd )
    ON_WM_PAINT( )
END_MESSAGE_MAP( )
/*
   !!! Обратите внимание на место расположения определения карты
сообщений!!!
   Макрос DECLARE_MESSAGE_MAP, расположенный в конце объявления
класса LFramtWnd, применяется для объявления таблицы сообщений.
   Структура карты сообщений достаточно проста и представляет собой
набор макросов, заключенных в специальные "операторные скобки".
   Начинается карта сообщений с вызова макроса
BEGIN_MESSAGE_MAP( FrameWnd, CFrameWnd ), который имеет следующие
аргументы: FrameWnd — задает имя класса, который является
владельцем карты сообщений; CFrameWnd — определяет имя базового
класса. Заканчивается карта сообщений вызовом макроса
END_MESSAGE_MAP( ). Между этими двумя вызовами располагаются
специальные макросы, называемые компонентами карты сообщений. Вот
они-то, собственно, и позволяют сопоставить сообщение с конкретным
обработчиком.
   Для каждого стандартного сообщения Windows определен свой макрос
в форме ON_WM_XXX, где XXX — имя сообщения, например, ON_WM_PAINT.
Имена обработчиков определяются при распаковке параметров каждого
сообщения Windows на основе простого соглашения. Имена обработчиков
всегда начинаются с префикса On, за которым следует имя
соответствующего сообщения Windows (без префикса WM_), записанное
строчными буквами. Например, для сообщения WM_PAINT в классе CWnd
определен обработчик (см. в файле FrameWnd.hpp)
   afx_msg void OnPaint( void );
Описания всех обработчиков стандартных сообщений Windows можно
найти в файле afxwin.h в объявлении класса CWnd.
   Отметим также, что для командных сообщений Windows от меню,
акселераторов и кнопок панелей инструментов, для команд обновления,
для извещений, посылаемых дочерними окнами своим родителям, для
кнопок, элементов редактирования, списков и комбинированных списков
используются другие макросы. Они будут рассмотрены в последующих
проектах.
*/

// Обработчик сообщения WM_PAINT: демонстрирует задание различных
//   контекстов (перья, кисти, цвета, толщина пера, стиль закраски
//   кистью и др.)
afx_msg void FrameWnd :: OnPaint( void )
```

```
{
    COLORREF            oldColor,   // Старый цвет символов
                        oldBkColor; // Старый цвет фона символов
    CBrush              NewBrush,   // Новая кисть
                        *OldBrush;  // Указатель на старую кисть
    CPen                NewPen,     // Новое перо
                        *OldPen;    // Указатель на старое перо

    // Создаем объект для рисования и вывода
    CPaintDC            PaintDC( this );
    /*
       Рисовать очень удобно с помощью класса CPaintDC. Объект
    CPaintDC представляет поверхность окна. Чтобы отчетливо понять
    его назначение, напомним как выполняется рисование с
    использованием функций API:
    // Дескриптор контекста устройства
    HDC                 hDC;
    // Указатель на структуру с информацией для приложения о
    //   клиентской области окна, используется для вывода в окно
    PAINTSTRUCT         ps;
    // Для любого вывода в окно Windows необходимо использовать
    //   функции GDI, которые в качестве параметра используют
    //   контекст устройства. Обычно контекст устройства получают с
    //   помощью фунции BeginPaint
    hDC = BeginPaint( hwnd, &ps );
    // Вывод заданного текста (четвертый аргумент, 28 — длина
    //   текста) в определенное место окна (второй и третий
    //   аргументы)
    TextOut( hDC, 150, 0, "Обработка сообщения WM_PAINT", 28 );
    // Вывод информации в окно завершается вызовом функции EndPaint
    //   (для каждого вызова BeginPaint должен существовать
    //   соответствующий вызов EndPaint)
    EndPaint( hwnd, &ps );
       Объект CPaintDC выполняет те же действия, но только
    абстрагирует их в класс MFC. Основные этапы использования
    объекта CPaintDC:
       1. Создание объекта CPaintDC.
       2. Рисование на объекте CPaintDC.
       3. Разрушение объекта CPaintDC.
       Конструктор CPaintDC вызывает функцию BeginPaint( ) и
    возвращает графический DC для рисования. Деструктор вызывает
    EndPaint( )
    */
```

```
// Вывод заданного текста (третий аргумент) в определенное
//   место окна (первый и второй аргументы)
PaintDC.TextOut( 0, 0, "Текст с цветом умолчания" );

// Вывод заданного текста (третий аргумент) в определенное
//   место окна (первый и второй аргументы). Перед выводом
//   текста задается цвет символов (зеленый) и цвет фона
//   (черный). После вывода текста восстанавливаются прежние
//   значения указанных параметров. Здесь макрос RGB(0, 255, 0)
//   задает тройку чисел, определяющих цвет — первое число
//   определяет значение красной составляющей, второе — зеленой
//   и третье — синей
oldColor = PaintDC.SetTextColor( RGB(0, 255, 0) );
oldBkColor = PaintDC.SetBkColor( RGB(0, 0, 0) );
PaintDC.TextOut( 250, 0, "Текст с изменением цвета" );
PaintDC.SetTextColor( oldColor );
PaintDC.SetBkColor( oldBkColor );

// Рисование линии: вначале выводится поясняющий текст
//   (используется текущий цвет и фон), а затем — рисуется
//   линия. Линия рисуется цветом текущего пера на текущем
//   фоне. В функции MoveTo( ) (125, 50) — новая текущая
//   позиция в окне. Функция LineTo( ) рисует линию из текущей
//   позиции до точки (175, 100)
PaintDC.TextOut( 50, 30, "Пример вывода линии в окно" );
PaintDC.MoveTo( 125, 50 ); PaintDC.LineTo( 175, 100 );

// Рисование эллипса: вначале выводится поясняющий текст
//   (используется текущий цвет и фон), а затем — рисуется
//   эллипс. В функции Ellipse( ) (100, 140) и (200, 240) —
//   координаты левого верхнего и правого нижнего углов
//   прямоугольника, в который вписан эллипс. Эллипс рисуется
//   существующим пером, а внутренняя область эллипса
//   закрашивается новой кистью (красной). После рисования
//   старая кисть восстанавливается
PaintDC.TextOut( 50, 120, "Пример вывода эллипса в окно" );
// Создаем красную кисть
NewBrush.CreateSolidBrush( RGB(255, 0, 0) );
// Включаем ее в контекст
OldBrush = PaintDC.SelectObject( &NewBrush );
PaintDC.Ellipse( 100, 140, 200, 240 );
// Удаляем новую кисть
NewBrush.DeleteObject( );
```

```
    // Восстанавливаем старую кисть
    PaintDC.SelectObject( OldBrush );

    // Рисование прямоугольника: вначале выводится поясняющий
    //   текст, а затем — рисуется прямоугольник. Перед рисованием
    //   прямоугольника задаются параметры пера для рисования
    //   прямоугольника. В функции Rectangle( ) (370, 140) и
    //   (470, 200) — координаты левого верхнего и правого нижнего
    //   углов прямоугольника. Прямоугольник закрашивается текущей
    //   кистью. После рисования старое перо восстанавливается
    PaintDC.TextOut( 300, 120,
                     "Пример вывода прямоугольника в окно" );
    // Создаем новое перо: точечное, толщиной 1, красного цвета
    NewPen.CreatePen( PS_DOT, 1, RGB(255, 0, 0) );
    // Включаем его в контекст
    OldPen = PaintDC.SelectObject( &NewPen );
    PaintDC.Rectangle( 370, 140, 470, 200 );
    // Удаляем новое перо
    NewPen.DeleteObject( );
    // Восстанавливаем старое перо
    PaintDC.SelectObject( OldPen );

    return afx_msg void( );
}
```

Листинг 4.5. Файл Main.cpp

```
/*
   Файл               : Main.cpp

   Проект             : любой проект на основе библиотеки классов
                        MFC

   Назначение         : инициализация и запуск приложения

   Microsoft Visual Studio C++ .NET 2005
*/

#include "MainThread.hpp"           // Объявление класса MainThread

// Инициализация и запуск приложения
MainThread              MainThread;
```

Обратите внимание на оформление идентификаторов в коде программы в соответствии с венгерской нотацией. *Венгерская нотация* — соглашение программистов об именовании переменных, констант и прочих идентификаторов в коде программы. Свое название венгерская нотация получила благодаря программисту компании Microsoft венгерского происхождения Чарльзу Шимоньи, впервые предложившему ее. Со временем венгерская нотация стала не только внутренним стандартом Microsoft, но и широко распространенным правилом программистов всего мира.

Суть венгерской нотации сводится к тому, что имена идентификаторов предваряются заранее оговоренными префиксами из одного или нескольких символов. Например, в качестве префикса для обозначения указателя может использоваться p (от слова pointer). При этом, как правило, ни само наличие префиксов, ни их написание не являются требованием языков программирования, у каждого программиста (или коллектива программистов) они могут быть своими. Использование в каждом из языков программирования своей терминологии также вносит особенности в выбор префиксов.

Далее рассмотрим и проанализируем исходный код проекта, и начнем с листингов 4.1 и 4.2.

4.2. Базовый класс *CWinApp* библиотеки классов MFC. Создание главного окна приложения

Место класса `CWinApp` в иерархии классов библиотеки MFC показано на рис. 4.1.

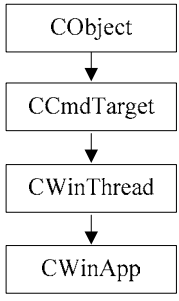


Рис. 4.1. Место класса `CWinApp` в иерархии классов библиотеки MFC

Класс `CWinApp` является классом Windows-приложения. Основными задачами объекта этого класса являются инициализация и создание главного окна, а

затем опрос системных сообщений. Определение класса CWinApp приведено в заголовочном файле afxwin.h. Но, рассмотрев приведенный код программы, вы зададите вопрос: а где же функция WinMain, которая является точкой входа в любую программу, написанную для ОС Windows? Пусть вас не удивляет то, что мы не встретили даже упоминания этой функции. Дело в том, что разработчики библиотеки MFC проделали большую работу и избавили программистов от написания многих строк вспомогательного кода, предоставив возможность заниматься только решением своей конкретной задачи. Однако при этом библиотека скрывает и часть того, что заставляет Windows-приложение работать. Сказанное в полной мере относится к функции WinMain. Именно с целью получения ясности в этом вопросе мы и применяем две разные технологии, использующие, соответственно, Windows API и библиотеку классов MFC, для реализации одного и того же проекта простого оконного приложения FrameWnd.

В нашем случае, функция WinMain, при создании объекта класса CWinApp или производного от него класса, операционной системой не вызывается. Вместо этого происходит обращение к стартовой функции _WinMainCRTStartup библиотеки времени выполнения. В числе прочих операций, выполняемых этой функцией, имеется вызов функции WinMain библиотеки MFC. Эта функция, как и обычная функция WinMain, выполняет следующее:

- инициализацию первого экземпляра приложения (метод InitApplication);
- инициализацию текущего экземпляра приложения (виртуальный метод InitInstance класса CWinApp или производного от него класса);
- запуск цикла обработки сообщений (виртуальный метод Run класса CWinApp);
- корректное завершение работы приложения (виртуальный метод Run класса CWinApp).

В классе CWinApp очень важную роль играет переопределяемый (виртуальный) метод InitInstance, который имеет следующий вид:

```
virtual BOOL CWinApp :: InitInstance( void )
{
    return TRUE;
}
```

В своем первозданном виде этот метод не делает "ничего" и наполнение его содержанием является задачей программиста. Обычно это то место, где конструируется объект главного окна приложения. Это единственный метод класса CWinApp, который вы должны обязательно переопределить для того, чтобы Windows-приложение что-нибудь делало. С этой целью в нашей про-

грамме используется класс `MainThread`, производный от класса `CWinApp`, в котором и переопределяется метод `InitInstance`.

Для создания окна используется последовательность действий (см. листинг 4.2, реализация метода `InitInstance`), аналогичная использованной в рассмотренном, в *главе 2*, низкоуровневом Windows-приложении.

Вначале в динамической памяти размещается объект класса `FrameWnd`, производного от класса `CFrameWnd` библиотеки MFC. Далее, в низкоуровневом Windows-приложении выполнялась регистрация класса окна. При использовании библиотеки MFC можно вообще не думать о регистрации оконного класса — это делается автоматически по умолчанию, с использованием имеющихся в библиотеке MFC предопределенных классов окон, которых практически вполне достаточно для всех возможных случаев. В рассматриваемом нами примере как раз и использовано подобное умолчание.

Для создания фрейма окна используется метод `Create`, наследованный из класса `CFrameWnd`:

```
// Возвращает ненулевое значение при успехе, иначе — 0
BOOL CFrameWnd :: Create(
    // Указатель на строку, завершающуюся нулевым символом. Задает
    //   имя класса Windows. Имя класса может быть любым именем,
    //   зарегистрированным с помощью глобальной функции
    //   AfxRegisterWndClass( ) или функции Windows
    //   RegisterClass( ). Если задано значение NULL, то
    //   используется предопределенное имя класса Windows
    LPCTSTR             lpszClassName,
    // Указатель на строку, завершающуюся нулевым символом. Задает
    //   текст в заголовке окна
    LPCTSTR             lpszWindowName,
    // Специфицирует стиль окна. Обратите внимание, что в качестве
    //   элементарных стилей окна можно использовать стили с
    //   префиксом WS_, перечисленные ранее в разд. 2.4 главы 2
    DWORD               dwStyle = WS_OVERLAPPEDWINDOW,
    // Определяет размер и местоположение окна
    const RECT          &rect = rectDefault,
    // Специфицирует радительское окно (NULL — родительское окно
    //   отсутствует)
    CWnd                *pParentWnd = NULL,
    // Определяет имя ресурса меню, используемого в окне. Для
    //   приведения значения целого идентификатора к типу строки
    //   используйте MAKEINTRESOURCE. Этот параметр может иметь
    //   значение NULL (меню не используется)
    LPCTSTR             lpszMenuName = NULL,
```

```
// Специфицирует атрибуты расширенного стиля окна
DWORD               dwExStyle = 0,
// Специфицирует указатель на структуру CCreateContext
//   (может иметь значение NULL)
CcreateContext      *pContext = NULL );
```

Местоположение и размеры окна задаются с помощью метода `MoveWindow`, наследованного из класса `CWnd`. Для отображения и перерисовки окна используются, соответственно, методы `ShowWindow` и `UpdateWindow`, наследованные от класса `CWnd`.

Замечание

Обратите внимание, в качестве аргумента метода `ShowWindow` можно использовать стили с префиксом `SW_`, рассмотренные в *разд. 2.4*.

После того как объект окна создан, указатель на этот объект присваивается переменной `CWinThread::m_pMainWnd`. Это необходимо сделать, чтобы корректно завершить приложение, когда будет закрыто его главное окно.

Примечание

Реализацию метода `InitInstance` можно существенно упростить, если более полно использовать возможности метода `Create`, как это показано в листинге 4.2. В этом случае становятся ненужными вызовы методов `MoveWindow`, `ShowWindow` и `UpdateWindow`.

4.3. Базовый класс *CFrameWnd* библиотеки классов MFC. Обработка сообщений главного окна приложения

Место класса `CFrameWmd` в иерархии классов библиотеки MFC показано на рис. 4.2.

Класс `CWnd` — это базовый класс для всех окон, созданных на базе библиотеки MFC. Помимо предоставления всем своим производным классам большого числа методов для работы с окнами, этот класс служит для создания разнообразных дочерних окон. В отличие от своего базового класса, класс `CFrameWnd` служит для создания перекрывающихся или всплывающих окон и спроектирован так, чтобы взять на себя выполнение всех основных функций Windows-приложения на базе SDK. Описание классов `CWnd` и `CFrameWnd` библиотеки MFC можно посмотреть в файле afxwin.h.

Класс `FrameWnd`, производный от класса `CFrameWnd`, предназначен для *обработки сообщений* главного окна Windows-приложения. Библиотека классов

MFC предоставляет альтернативу многочисленным операторам `switch`, используемым в оконных процедурах традиционных низкоуровневых Windows-программ для обработки сообщений, посылаемых окну. Взаимосвязь сообщений и их обработчиков может быть определена таким образом, что, когда сообщение поступает в оконную процедуру, автоматически вызывается соответствующий обработчик. Реализация такой взаимосвязи основана на понятии *карты* (или *таблицы*) *сообщений* (message map).

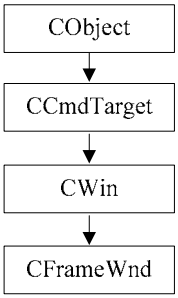


Рис. 4.2. Место класса `CFrameWnd` в иерархии классов библиотеки MFC

Таблица сообщений представляет собой механизм пересылки сообщений и команд Windows в окна, представления и другие объекты приложения, реализованного на базе библиотеки классов MFC. Такие таблицы преобразуют сообщения Windows, извещения элементов управления, а также команды меню, кнопок панелей инструментов, акселераторов клавиатуры в вызовы функций соответствующих классов, которые их обрабатывают. Эта отличительная черта таблицы сообщений реализована по аналогии с виртуальными функциями языка C++, но имеет дополнительные преимущества, недоступные для них. Каждый класс, который может получить сообщение, должен иметь свою таблицу сообщений, чтобы иметь возможность соответствующим образом обрабатывать сообщения. При этом следует иметь в виду, что таблица сообщений должна определяться вне какой-либо функции или объявления класса. Как уже указывалось ранее, в нашем примере для обработки сообщений используется класс `FrameWnd`, производный от класса `CFrameWnd` библиотеки MFC (см. листинги 4.3, 4.4):

```
// Объявление класса
class FrameWnd : public CFrameWnd
{
    // Методы

protected:
```

```
    // Так нужно записывать прототип функции, обрабатывающей
    //   сообщение WM_PAINT
    afx_msg void OnPaint( void );

    // Объявление таблицы сообщений
    DECLARE_MESSAGE_MAP( );
};
```

Для определения таблицы сообщений используются три макроса: в файле реализации класса `FrameWnd` макросы `BEGIN_MESSAGE_MAP`, `END_MESSAGE_MAP` и файле объявления класса `FrameWnd` макрос `DECLARE_MESSAGE_MAP`. Макрос `DECLARE_MESSAGE_MAP` располагается в конце объявления класса, использующего карту сообщений. Непосредственно перед ним располагаются прототипы обрабатывающих функций — в нашем примере используется единственный прототип. В прототипах обрабатывающих функций используется макрос `afx_msg`, который разработчики библиотеки классов MFC предусмотрели для будущего применения. Однако он до сих пор реально не используется. Макрос `afx_msg` в определении не раскрывается и употребляется только как метка-заполнитель, чтобы идентифицировать обработчики сообщений в объявлении класса.

Два остальных макроса, определяющих карту сообщений, располагаются в реализации класса на внешнем уровне (см. листинг 4.4):

```
// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( FrameWnd, CFrameWnd )
    ON_WM_PAINT( )
END_MESSAGE_MAP( )
```

Структура карты сообщений достаточно проста и представляет собой набор макросов (в данном примере набор содержит один макрос), заключенных в специальные "операторные скобки-макросы". Начинается карта сообщений с вызова макроса `BEGIN_MESSAGE_MAP( FrameWnd, CFrameWnd )`, который имеет следующие аргументы: `FrameWnd` — задает имя класса, который является владельцем карты сообщений, `CFrameWnd` — определяет имя базового класса. Заканчивается карта сообщений вызовом макроса `END_MESSAGE_MAP( )`. Между этими двумя вызовами располагаются специальные макросы, называемые компонентами карты сообщений. Вот они-то, собственно, и позволяют сопоставить сообщение с конкретным обработчиком.

Для каждого стандартного сообщения Windows определен свой макрос в форме `ON_WM_XXX`, где `XXX` — имя сообщения, например, `ON_WM_PAINT`. Имена обработчиков определяются при распаковке параметров каждого сообщения

Windows на основе простого соглашения. Имена обработчиков всегда начинаются с префикса `On`, за которым следует имя соответствующего сообщения Windows (без префикса `WM_`), записанное строчными буквами и начинающееся с прописной буквы. Например, для сообщения `WM_PAINT` в классе `CWnd` определен обработчик (см. листинг 4.3)

```
afx_msg void OnPaint( void );
```

Замечание

Обратите внимание, что макрос `afx_msg` выполняет в данном случае роль служебного слова `virtual`, что позволяет переопределить виртуальный метод `OnPaint` в производном классе `FrameWnd`.

Описания всех обработчиков стандартных сообщений Windows можно найти в файле afxwin.h, в объявлении класса `CWnd`. Отметим также, что для командных сообщений от меню, акселераторов и кнопок панелей инструментов, для команд обновления, для извещений, посылаемых дочерними окнами своим родителям, для кнопок, элементов редактирования, списков и комбинированных списков используются макросы с другим форматом. Они будут рассмотрены далее в последующих проектах.

4.4. Обработка сообщения *WM_PAINT*. Вывод текстовой и графической информации

Об обработке сообщения `WM_PAINT` и выводе в окно текстовой и графической информации мы довольно подробно говорили ранее, в *главе 2*, при рассмотрении "скелета" Windows-приложения, полученного с использованием средств низкоуровневого программирования. Поэтому здесь можно ограничиться лишь кратким описанием особенностей обработки сообщения `WM_PAINT` и вывода текстовой и графической информации в окно. Самое важное — все стало гораздо проще и удобнее в результате использования библиотеки классов MFC.

Реализация метода `OnPaint` класса `FrameWnd` приведена ранее, в листинге 4.4. Рисовать стало очень удобно с помощью класса `CPaintDC` библиотеки классов MFC. Объект класса `CPaintDC` представляет собой поверхность окна. Чтобы отчетливо понять его назначение, еще раз рассмотрим, как выполняется рисование с использованием стандартных функций API:

```
// Дескриптор контекста устройства
HDC                     hDC;
```

```
// Указатель на структуру с информацией для приложения о клиентской
//   области окна используется для вывода в окно
PAINTSTRUCT             ps;
// Для любого вывода в окно Windows необходимо использовать функции
//   GDI, которые в качестве параметра используют контекст
//   устройства. Обычно контекст устройства получают с помощью
//   фунции BeginPaint
hDC = BeginPaint( hwnd, &ps );
// Вывод заданного текста (четвертый аргумент, 28 — длина текста) в
//   определенное место окна (второй и третий аргументы)
TextOut( hDC, 150, 0, "Обработка сообщения WM_PAINT", 28 );
// Вывод информации в окно завершается вызовом функции EndPaint
//   (для каждого вызова BeginPaint должен существовать
//   соответствующий вызов EndPaint)
EndPaint( hwnd, &ps );
```

Объект `CPaintDC` выполняет те же действия, но только абстрагирует их в класс библиотеки MFC. Основными этапами использования объекта `CPaintDC` являются:

1. Создание объекта `CPaintDC`.
2. Рисование на объекте `CPaintDC`.
3. Разрушение объекта `CPaintDC`.

Конструктор `CPaintDC` автоматически активизируется при создании объекта с типом `CPaintDC`, вызывает функцию `BeginPaint` и возвращает графический контекст для рисования. Деструктор автоматически активизируется при разрушении объекта с типом `CPaintDC` и вызывает `EndPaint`. Реализация метода `OnPaint`, обеспечивающая вывод в окно, эквивалентный рассмотренному, имеет более простой и удобный вид:

```
// Обработчик сообщения WM_PAINT
afx_msg void FrameWnd :: OnPaint( void )
{
    // Создаем объект для рисования и вывода
    CPaintDC            PaintDC( this );

    // Вывод заданного текста (третий аргумент) в определенное
    //   место окна (первый и второй аргументы)
    PaintDC.TextOut( 0, 0, "Текст с цветом умолчания" );

    return afx_msg void( );
}
```

Примечание

Обратите внимание, что методы класса `CPaintDC` стали иметь более удобный интерфейс. Так, в методе `TextOut` количество параметров уменьшилось на два — стал ненужным дескриптор контекста устройства (он стал членом класса `CPaintDC`), а длина выводимого текста определяется (по третьему аргументу) автоматически.

Реализация метода `OnPaint` класса `FrameWnd`, демонстрирующая различные виды рисования в окне, рассмотрена в файле FrameWnd.cpp (см. листинг 4.4) и не требует особых пояснений. Отметим лишь общую схему рисования:

- создание новых инструментов с требуемыми свойствами (перо, кисть и т. д.);
- включение созданных инструментов в контекст с запоминанием информации об аналогичных инструментах, используемых в контексте по умолчанию;
- рисование новыми инструментами;
- по окончании рисования — разрушение созданных инструментов;
- включение в контекст старых инструментов, используемых по умолчанию.

В заключение выполните последовательную компиляцию файлов демонстрационного проекта, постройте проект и запустите его командой **Start Without Debugging** (Запуск без отладки) и, если нет ошибок, вы увидите окно приложения (рис. 4.3).

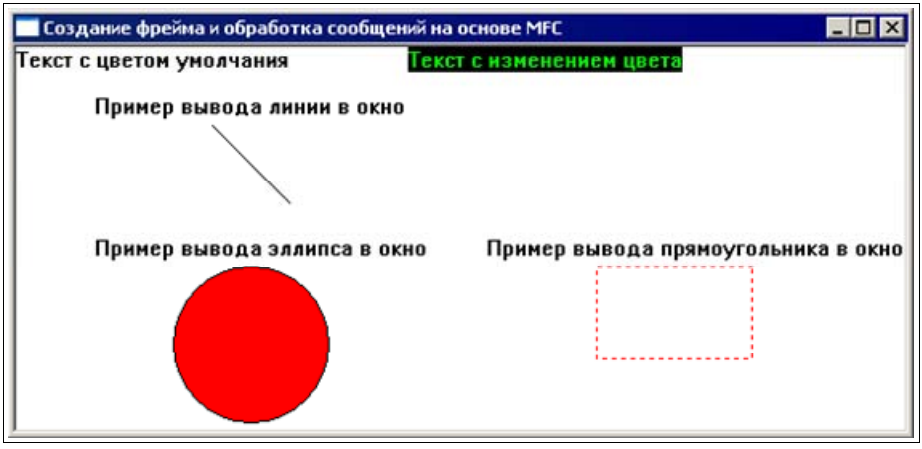


Рис. 4.3. Главное окно приложения

4.5. Прекомпиляция стандартных заголовочных файлов приложений на базе MFC

Прекомпиляция стандартных заголовочных файлов приложений на базе MFC имеет следующие отличия от низкоуровневых приложений:

- в прекомпилируемый файл StdAfx.h включаются другие стандартные заголовочные файлы, отличные от файла Windows.h, используемого в низкоуровневых Windows-приложениях (листинг 4.6);
- прекомпилируемый файл StdAfx.h включается во все файлы проекта на базе MFC с расширениями cpp, а не в файлы с расширениями h или hpp.

Листинг 4.6. Файл StdAfx.h

```
/*
   Файл               : StdAfx.h

   Проект             : любой проект, использующий библиотеку
                        классов MFC

   Назначение         : стандартный включаемый файл для
                        прекомпиляции (результаты компиляции в
                        папке Debug в файле .PCH)

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _StdAfx_h
    #define _StdAfx_h

    // Здесь следует подключить все стандартные (неизменяемые)
    //  заголовочные файлы большого размера
    #include <afxwin.h>              // Ядро MFC и стандартные
                                     //   компоненты
    //#include <afxcmn.h>            // Общие управляющие элементы
                                     //   MFC
    //#include <afxdlgs.h>           // Диалоги MFC
    //#include <afxext.h>            // Классы расширений MFC
    //#include <afxmt.h>             // Многопоточность MFC
    //#include <afxres.h>            // Ресурсы MFC

#endif
```

Файл StdAfx.cpp соответствует листингу 2.7. Опции, задающие прекомпиляцию, задаются точно так же, как это указано в *разд. 2.7*. В окне **Solution Explorer — FrameWnd** для проекта **FrameWnd** правой кнопкой мыши вызовите контекстное меню и выполните команду **Properties**. В появившемся окне **FrameWnd Property Pages** откройте вкладку **C/C++**, выберите категорию **Precompiled Headers** и в поле **Create/Use Precompiled Headers** выберите **Use Precompiled Headers (/Yu)** и нажмите кнопку **OK**. Аналогично в окне **Solution Explorer — FrameWnd** для файла StdAfx.cpp правой кнопкой мыши вызовите контекстное меню и выполните команду **Properties**. В появившемся окне **StdAfx.cpp Property Pages** откройте вкладку **C/C++**, выберите категорию **Precompiled Headers** и в поле **Create/Use Precompiled Headers** выберите **Create Precompiled Headers (/Yc)** и нажмите кнопку **OK**.

Совет

Для ускорения отладки в приложениях на базе MFC используйте прекомпиляцию стандартных заголовочных файлов.

Замечание

Программный проект FrameWnd_PCH с прекомпиляцией имеется на прилагаемом компакт-диске, в папке \Глава 04\FrameWnd_PCH, а его главное окно показано на рис. 4.3.

4.6. Вопросы и упражнения для самопроверки

Для самопроверки ответьте на приведенные далее вопросы.

1. С какой целью рассматривается один и тот же проект простого Windows-приложения с использованием двух технологий создания — на базе Windows API и библиотеки классов MFC?
2. В чем заключаются особенности оформления исходных текстов приложения на базе MFC FrameWnd?
3. Укажите место класса `CWinApp` в иерархии классов библиотеки MFC и его назначение.
4. Что происходит при создании объекта с типом `CWinApp` или с типом производного от него класса?
5. Какую роль в классе `CWinApp` играет виртуальный метод `InitInstance` и с какой целью его следует переопределять?
6. Укажите назначение методов `Create`, `MoveWindow`, `ShowWindow` и `UpdateWindow`, наследованных из класса `CWnd`.

7. Можно ли обойтись только методом `Create`, не используя методов `MoveWindow`, `ShowWindow` и `UpdateWindow`?
8. Как сделать созданный объект окна главным?
9. Укажите место класса `CFrameWnd` в иерархии классов библиотеки MFC и его назначение.
10. Для чего предназначен класс, производный от класса `CFrameWnd`?
11. Что такое таблица сообщений? Укажите ее назначение.
12. Укажите структуру таблицы сообщений.
13. Охарактеризуйте объект класса `CPaintDC`.
14. Перечислите основные методы класса `CPaintDC` для рисования на экране графических примитивов.
15. Укажите общую схему рисования на экране с использованием класса `CPaintDC`.
16. Какие недостатки имеются в использовании методов `Create`, `TextOut`, `CreateSolidBrush`, `Ellipse`, `DeleteObject`, `CreatePen` и `Rectangle`? Как их можно исправить?

Ответы на вопросы можно проверить — они приведены в *разд. П1.4 приложения 1*.

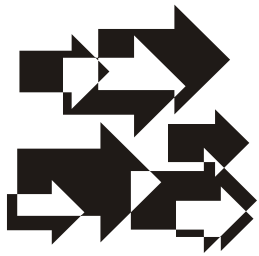
Совет

Одним из указанных в *главе 2* и *приложении 3* способов создайте копию проекта FrameWnd_PCH, использующего библиотеку классов MFC.

При экспериментах с созданной копией проекта выполните перечисленные упражнения.

1. Попробуйте изменить заголовок, местоположение и размеры окна.
2. Задавая различные стили окна и их комбинации (константы, задающие стили, начинаются с префикса `WS_`), изучите их влияние на свойства окна.
3. Задавая различные варианты начального отображения окна — различные значения аргумента в вызове метода `ShowWindow` (константы, задающие вид начального отображения, начинаются с префикса `SW_`), изучите их влияние на свойства окна.
4. Реализуйте вывод в окно различных текстов и других графических объектов, варьируя цвета символов и фона, характеристики перьев и кистей для закрашивания замкнутых контуров.
5. Нарисуйте в окне цветную и содержательную картинку.

ГЛАВА 5



Windows-приложения с дочерними окнами

Многие Windows-приложения в качестве интерфейсных, управляющих элементов используют кнопки, являющиеся специальной разновидностью дочерних окон. Как и в *главах 2 и 4*, далее рассматриваются оконные приложения с кнопками и дочерними окнами с использованием *технологий* Windows API, а также библиотеки классов MFC, что, как и ранее, позволит выполнить их сравнительный анализ и лучше понять анатомию Windows-приложений на базе MFC.

Совет

Новым в этой главе является использование дочерних окон и блоков сообщений. Обратите на них первоочередное внимание.

Чтобы познакомиться с созданием кнопок и обработкой их нажатия, вначале рассматриваются два более простых проекта: WndBtn и WndBtn_MFC, использующих соответственно Windows API и библиотеку классов MFC. Далее рассматриваются еще два, более сложных проекта — WndWndBtn и WndWndBtn_MFC — с полноценными дочерними окнами.

Замечание

Обычно кнопки, как разновидность управляющих элементов, используются в диалоговых окнах. В дальнейшем, после рассмотрения диалоговых окон, использующих кнопки, вы сможете выполнить сравнительный анализ обоих вариантов — с кнопками в обычном и диалоговом окнах.

Для всех перечисленных проектов сначала приводится их полный исходный код (с подробными комментариями), а затем проводится их анализ.

5.1. Низкоуровневое Windows-приложение с дочерним окном-кнопкой

Соответствующий программный проект имеется на прилагаемом компакт-диске, в папке \Глава 05\WndBtn, а его файловый состав представлен на рис. 5.1.

Рис. 5.1. Файловый состав проекта WndBtn

Совет

Проект WndBtn создавайте аналогично низкоуровневому проекту FrameWnd из *главы 2*, используя в качестве основы его файлы StdAfx.h, StdAfx.cpp (эти файлы не изменяются) и файл FrameWnd.cpp (скопируйте и переименуйте его в WndBtn.cpp, и модифицируйте в соответствии с листингом 5.1). Более подробные сведения о создании проекта из имеющихся файлов будут рассмотрены далее, в *приложении 3*.

Листинг 5.1. Файл WndBtn.cpp

```
/*
   Файл                : WndBtn.cpp

   Проект              : Windows-приложение с использованием
                         стандартных функций API (создается
                         главное окно, в него передается сообщение и
                         кнопка, нажатие кнопки обрабатывается)

   Microsoft Visual Studio C++ .NET 2005
*/

// Прекомпилируемый заголовочный файл
#include "stdafx.h"
```

```
// Прототипы используемых функций
LRESULT CALLBACK WndProc( HWND hwnd, UINT message, WPARAM wParam,
                          LPARAM lParam );
BOOL InitApplication( HINSTANCE hInstance );
BOOL InitInstance( HINSTANCE hInstance, int nCmdShow );

// Дескриптор кнопки (дочернего окна)
HWND                  hWnd1;

#define BTN1 1                        // Идентификатор кнопки

// ****************************************************************
// Главная функция приложения
int WINAPI WinMain(                   // Возвращает TRUE при успехе
    // Дескриптор данного приложения
    HINSTANCE           hInstance,
    // Дескриптор предыдущей запущенной копии приложения.
    //   В Win32API этот параметр всегда равен NULL и оставлен
    //   исключительно для совместимости с версиями ниже четвертой.
    //   Связано это с тем, что каждое 32-разрядное приложение
    //   запускается в собственном адресном пространстве, в
    //   котором, естественно, нет никаких копий или других
    //   приложений
    HINSTANCE           hPrevInstance,
    // Указатель на командную строку, которую можно в
    //   интегрированной среде разработки задать по команде
    //   "Properties" контекстного меню проекта в элементе "Program
    //   arguments" вкладки "Debugging"
    LPSTR               lpCmdLine,
    // Режим начального отображения главного окна приложения -
    //   после вызова параметр получает значение SW_SHOWNORMAL (1),
    //   что соответствует отображению окна в нормальном виде
    int                 nCmdShow )
{
    // Инициализируем приложение — подготавливаем данные класса
    //   окна и регистрируем его
    if ( !InitApplication( hInstance ) )
        return FALSE;

    // Завершаем создание приложения — создаем и отображаем главное
    //   окно приложения
    if ( !InitInstance( hInstance, nCmdShow ) )
        return FALSE;
```

```
    MSG                 msg;        // Для очередного сообщения
    // Стандартный цикл обработки сообщений
    while ( GetMessage( &msg, NULL, 0, 0 ) )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }

    return static_cast< int >( msg.wParam );
}

// ****************************************************************
BOOL InitApplication(               // Возвращает TRUE при успехе
    // Дескриптор (уникальное число), ассоциируемый с текущим
    //   приложением
    HINSTANCE           hInstance )
{
    // Сведения о регистрируемом классе
    WNDCLASS            wc;

    // Заполняем структуру класса окна WNDCLASS — смысл
    //   инициализирующих значений рассмотрен в проекте FrameWnd
    wc.style         = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc   = static_cast< WNDPROC >( WndProc );
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = hInstance;
    wc.hIcon         = LoadIcon( NULL, IDI_ASTERISK );
    wc.hCursor       = LoadCursor( NULL, IDC_CROSS );
    wc.hbrBackground = reinterpret_cast< HBRUSH >( COLOR_WINDOW+1 );
    wc.lpszMenuName  = NULL;
    wc.lpszClassName = "WndBtnAPI";

    // Регистрируем класс окна
    return RegisterClass( &wc );

    // !!! Класс окна для кнопки ("BUTTON") является стандартным и
    //   не требует специальной регистрации !!!

    return TRUE;
}
```

```
// ****************************************************************
// Создание главного окна приложения и кнопки
BOOL InitInstance(                  // Возвращает TRUE при успехе
    // Дескриптор текущего приложения
    HINSTANCE           hInstance,
    // Режим отображения главного окна — определяет, в каком виде
    //   будет отображено окно приложения
    int                 nCmdShow )
{
    // Дескриптор главного окна
    HWND                hWnd;

    // Создание главного окна приложения
    hWnd = CreateWindow(
        // Имя зарегистрированного класса
        "WndBtnAPI",
        // Заголовок окна
        "Создание приложения WndBtn с использованием"
        "Windows API с текстом и кнопкой в главном окне",
        // Комбинированный стиль окна: WS_OVERLAPPED | WS_CAPTION |
        //   WS_SYSMENU | WS_THICKFRAME | WS_MINIMIZEBOX |
        //   WS_MAXIMIZEBOX
        WS_OVERLAPPEDWINDOW,        // Стиль окна
        // Горизонтальная координата левого верхнего угла окна
        //   (используется умалчиваемое значение)
        CW_USEDEFAULT,
        // Вертикальная координата левого верхнего угла окна
        //   (используется умалчиваемое значение)
        CW_USEDEFAULT,
        CW_USEDEFAULT,              // Ширина окна (используется
                                    //   умалчиваемое значение)
        CW_USEDEFAULT,              // Высота окна (используется
                                    //   умалчиваемое значение)
        NULL,                       // Дескриптор родительского
                                    //   окна (его нет)
        NULL,                       // Дескриптор меню окна (его
                                    //   нет)
        hInstance,                  // Дескриптор экземпляра
                                    //   приложения
        NULL );                     // Указатель на дополнительные
                                    //   данные окна (их нет)
    if ( !hWnd )
        return FALSE;
```

```
// Показать окно: передает в ОС Windows информацию (nCmdShow) о
//   том, в каком виде необходимо отобразить окно. Значение
//   этого аргумента можно выбирать по своему усмотрению
if ( ShowWindow( hWnd, nCmdShow ) )
    return FALSE;

// Перерисовать окно: для перерисовки окна функция предписывает
//   ОС Windows послать окну сообщение WM_PAINT
if ( !UpdateWindow( hWnd ) )
    return FALSE;

// Создание кнопки. Здесь будьте внимательнее — это новая
//   информация, которая отсутствовала в проекте FrameWnd
hWnd1 = CreateWindow(
    // Имя зарегистрированного класса (в данном случае это
    //   стандартное имя)
    "BUTTON",
    "Кнопка",                    // Текст на кнопке
    // Стиль окна (WS_VISIBLE использовать обязательно), кнопка
    //   является дочерним окном
    WS_VISIBLE | WS_CHILD,
    // Горизонтальная координата левого верхнего угла кнопки
    210,
    // Вертикальная координата левого верхнего угла кнопки
    20,
    80,                          // Ширина кнопки
    20,                          // Высота кнопки
    // Дескриптор родительского окна
    hWnd,
    // Для кнопки — ее идентификатор (см. определение на
    //   внешнем уровне)
    reinterpret_cast< HMENU >( BTN1 ),
    // Дескриптор экземпляра приложения
    hInstance,
    // Указатель на дополнительные данные окна (они
    //   отсутствуют)
    NULL );
if ( !hWnd1 )
{
    MessageBox( NULL, "Кнопка не создана", "Ошибка 1", MB_OK );
    exit( 1 );
}
```

```
/*
   Наряду с использованием предопределенного имени оконного
класса, в качестве еще одной особенности создания кнопки
отметим возможность комбинирования обычных стилей окна,
перечисленных в проекте FrameWnd, со специфическими стилями
кнопок (в данном примере это, правда, не использовано).
   Существуют следующие стили кнопок.
BS_3STATE
    Это стиль отмечаемой кнопки (checkbox), но окно может быть
помечено или затемнено (показывая, что кнопка деактивирована)
BS_AUTO3STATE
    Это стиль отмечаемой кнопки, но окно может быть помечено
или затемнено (показывая, что кнопка деактивирована). Когда
пользователь выбирает эту кнопку, ее состояние автоматически
переключается
BS_AUTOCHECKBOX
    Это стиль отмечаемой кнопки, но когда пользователь выбирает
эту кнопку, автоматически включается ее отмеченное состояние
BS_AUTORADIOBUTTON
    Это стиль радиокнопки. Когда пользователь выбирает эту
кнопку, она помечается. При этом автоматически снимаются
пометки с других кнопок того же стиля в той же группе
BS_CHECKBOX
    Это стиль отмечаемой кнопки (небольшой квадрат с названием,
расположенным с правой стороны по умолчанию или с левой
стороны, если стиль совмещен со стилем BS_LEFTTEXT)
BS_DEFPUSHBUTTON
    Это стиль стандартной нажимаемой кнопки (кнопка с жирной
темной рамкой), что дает возможность пользователю, нажимая на
клавишу Enter, быстро выбирать команду по умолчанию
BS_GROUPBOX
    Это стиль обрамляющего прямоугольника с названием для
визуальной группировки элементов управления
BS_LEFTTEXT
    В сочетании со стилями радиокнопки или отмечаемой кнопки
этот стиль приводит к тому, что название кнопки располагается
слева от нее самой
BS_OWNERDRAW
    Стиль создает изображаемую пользователем кнопку. MFC
автоматически вызывает метод DrawItem( ), когда кнопка
изменяется визуально. Этот стиль чаще всего используется для
кнопок класса CBitmapButton
```

```
    BS_PUSHBUTTON
        Стиль создает кнопку, посылающую сообщение WM_COMMAND окну-
    хозяину, когда пользователь ее нажимает
    BS_RADIOBUTTON
        Этот стиль создает радиокнопку (небольшой кружок с текстом
    с правой стороны по умолчанию или с левой, если стиль совмещен
    со стилем BS_LEFTTEXT). Кнопки этого стиля обычно используются
    в группе, чтобы обозначить связанные взаимоисключающие
    возможности
    BS_BITMAP
        Стиль создает кнопку с изображением рисунка
    BS_BOTTOM
        Стиль размещает текст кнопки внизу прямоугольника
    BS_CENTER
        Стиль выполняет горизонтальное центрирование текста в
    прямоугольнике
    BS_ICON
        Стиль создает кнопку с изображением пиктограммы

        Существуют и другие стили кнопок, такие как BS_FLAT,
    BS_LEFT, BS_MULTILINE, BS_NOTIFY, BS_PUSHLIKE, BS_RIGHT,
    BS_RIGHTBUTTON, BS_TEXT, BS_TOP, BS_VCENTER (см. встроенную
    справку)
    */

    return TRUE;
}

// ************************************************************
// Оконная процедура
// Получает очередное сообщение и индивидуально обрабатывает его
LRESULT CALLBACK WndProc(           // Возвращает результат
                                    //   обработки сообщения,
                                    //   зависящий от посланного
                                    //   сообщения: 0 — успех
    HWND                hwnd,       // Дескриптор созданного окна
    UINT                message,    // Номер сообщения
    WPARAM              wParam,     // Дополнительная информация о
    LPARAM              lParam )    //   сообщении, зависящая от
                                    //   типа сообщения (wParam,
                                    //   lParam)
{
    // Обработчик сообщения — в данном случае приложение явно
    //   отвечает только на сообщения WM_DESTROY, WM_PAINT и
```

```
//   WM_COMMAND от нажатия кнопки, все остальные сообщения
//   передаются в DefWindowProc( ) — функцию, управляющую
//   поведением окна по умолчанию. Обработка сообщений
//   WM_DESTROY и WM_PAINT подробно рассмотрена в проекте
//   FrameWnd
switch( message )
{
case WM_PAINT:

    // Дескриптор контекста устройства
    HDC             hDC;
    // Указатель на структуру с информацией для приложения
    //   о клиентской области окна
    PAINTSTRUCT     ps;
    // Получение контекста устройства
    hDC = BeginPaint( hwnd, &ps );
    if ( !hDC )
    {
        MessageBox( NULL, "Контекст устройства не получен",
                    "Ошибка 2", MB_OK );
        exit( 2 );
    }
    // Вывод заданного текста
    if ( !TextOut( hDC, 150, 0,
                  "Обработка сообщения WM_PAINT", 28 ) )
    {
        MessageBox( NULL, "Неверное использование TextOut",
                    "Ошибка 3", MB_OK );
        exit( 3 );
    }
    // Освобождение контекста устройства
    if ( !EndPaint( hwnd, &ps ) )
    {
        MessageBox( NULL, "Контекст устройства не освобожден",
                    "Ошибка 4", MB_OK );
        exit( 4 );
    }
    break;

case WM_DESTROY:

    // Указывает системе, что сделан запрос о завершении приложения:
    //   0 — код завершения. Вызов этой функции обычно используется
    //   в ответ на поступившее сообщение WM_DESTROY
```

```
        PostQuitMessage( 0 );
        break;

        // !!! Здесь будьте внимательнее — новая информация !!!
        // Сообщение WM_COMMAND генерируется очень большим
        //   количеством управляющих элементов (всевозможные
        //   кнопки, команды меню и т. п.). Для того чтобы
        //   можно было правильно обработать конкретное
        //   сообщение WM_COMMAND, нужно знать, какой же из
        //   управляющих элементов создал сообщение. Информация
        //   об этом (идентификатор управляющего элемента)
        //   хранится в младшем слове параметра wParam и
        //   получить ее можно путем вызова макроса с
        //   параметром LOWORD( wParam )
        case WM_COMMAND:
        {
            switch( LOWORD( wParam ) )
            {
            case BTN1:              // Нажата "Кнопка"

                // Звуковой сигнал
                if ( !MessageBeep( MB_OK ) )
                {
                    MessageBox( NULL, "Ошибка MessageBeep",
                                "Ошибка 5", MB_OK );
                    exit( 5 );
                }
                MessageBox( NULL, "Нажата кнопка", "Информация",
                            MB_OK );
                // Обратите внимание, что вместо такой несложной
                //   обработки можно выполнить обработку любой
                //   сложности
            }
        }
        break;

        default:

            // Обработка сообщения по умолчанию
            return DefWindowProc( hwnd, message, wParam, lParam );

        }

    return 0;
}
```

После компиляции и запуска проекта на экране появляется главное окно приложения, показанное на рис. 5.2.

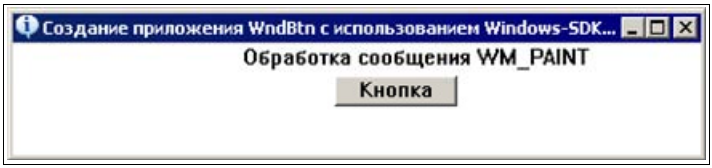


Рис. 5.2. Главное окно приложения WndBtn

Обработка нажатия кнопки в этом окне заключается в подаче звукового сигнала и выводе окна сообщения, показанного на рис. 5.3.

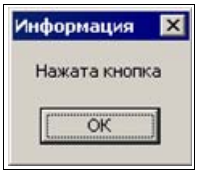


Рис. 5.3. Блок сообщения

Далее кратко проанализируем исходный текст, приведенный в листинге 5.1. Большая часть этого кода ясна из ранее рассмотренного проекта FrameWnd *(см. главу 2)*, при создании которого также были использованы стандартные функции API. Подчеркнем лишь несколько особенностей, связанных с *созданием* и *обработкой* нажатия кнопки.

Первая особенность состоит в том, что оконные классы, используемые при создании кнопок, являются *предопределенными* и их не нужно регистрировать. В нашем примере был использован предопределенный оконный класс с именем `"BUTTON"`.

Вторая особенность заключается в том, что при создании кнопки можно комбинировать обычные стили окна, рассмотренные ранее, в *разд. 2.4*, со специфическими стилями кнопок. Но, отметим — эта возможность в нашем примере не использована. Такими специфическими стилями кнопок являются стили, указанные, для удобства читателей, в виде комментариев в тексте файла WndBtn.cpp (см. листинг 5.1).

Третья особенность связана с тем, что сообщение `WM_COMMAND` генерируется очень большим количеством управляющих элементов (всевозможные кнопки, команды меню и т. п.). Чтобы можно было правильно обработать конкретное сообщение `WM_COMMAND`, нужно знать, какой же из управляющих элементов создал его. Информация об этом (идентификатор управляющего эле-

мента) хранится в младшем слове параметра сообщения `wParam` и получить ее можно с помощью параметризованного макроса `LOWORD(wParam)`.

Совет

В рассмотренном проекте используются стандартные функции `MessageBox` и `MessageBeep`. Изучите их самостоятельно, пользуясь встроенной справочной системой — это несложно.

Далее, для сравнения, рассмотрим аналогичное Windows-приложение с дочерними окнами-кнопками на базе MFC.

5.2. Windows-приложение с дочерними окнами-кнопками на базе библиотеки классов MFC

Соответствующий программный проект имеется на прилагаемом компакт-диске, в папке \Глава 05\WndBtn_MFC, а его файловый состав представлен на рис. 5.4.

Рис. 5.4. Файловый состав проекта WndBtn_MFC

Совет

Проект WndBtn создавайте аналогично высокоуровневому проекту FrameWnd_MFC *(см. главу 4)*, используя, в качестве основы, его файлы StdAfx.h, StdAfx.cpp, Main.cpp, MainThread.hpp (эти файлы не изменяются) и файлы MainThread.cpp, FrameWnd.hpp и FrameWnd.cpp (модифицируйте их в соответствии с листингами 5.2—5.4). Более подробные сведения о создании проекта вы найдете в *приложении 3*. Для добавления в класс проекта переменной или функции (метода), при модификации файлов, пользуйтесь мастерами (рис. 5.5 и 5.6).

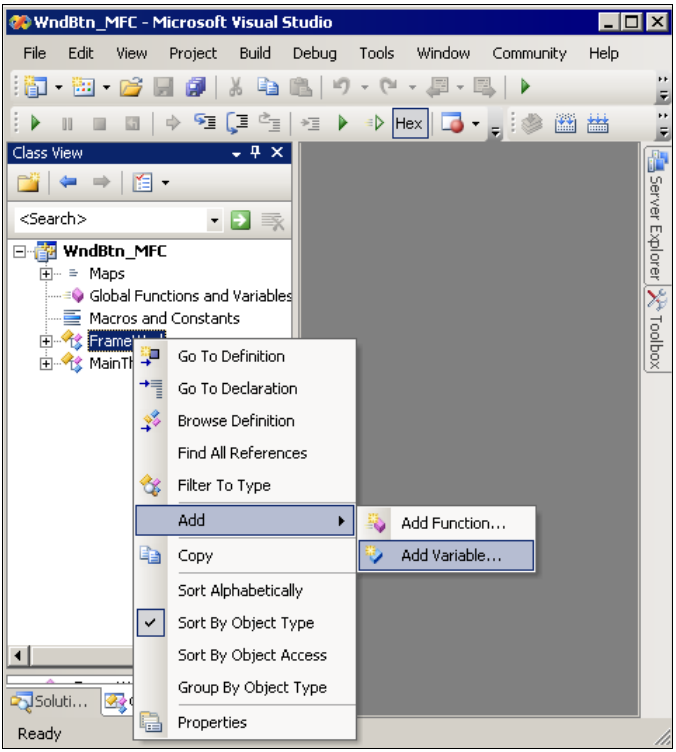


Рис. 5.5. Добавление в класс переменной или функции (метода)

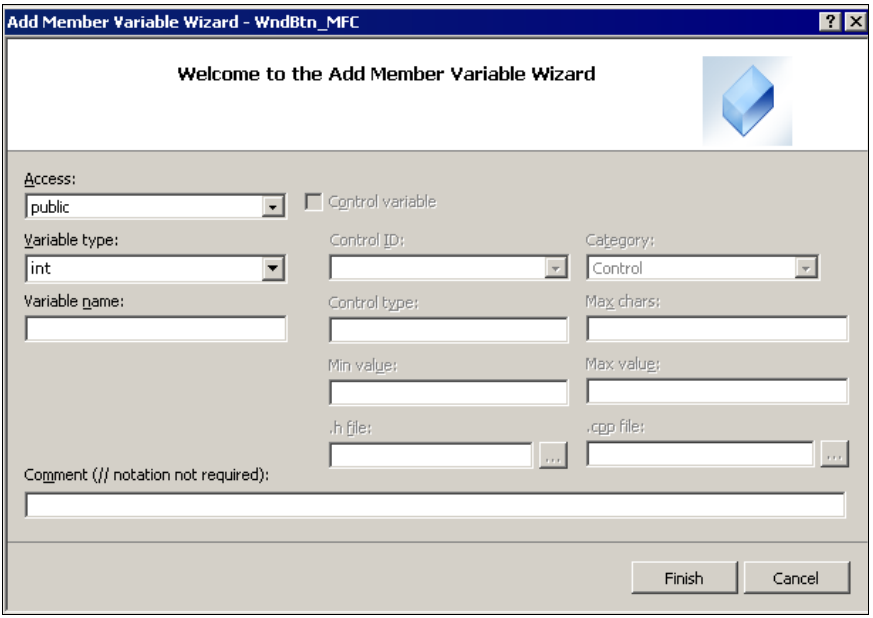


Рис. 5.6. Окно мастера для добавления в класс переменной

Листинг 5.2. Файл MainThread.cpp

```
/*
   Файл                : MainThread.cpp

   Проект              : демонстрация создания Windows-приложения на
                         основе библиотеки классов MFC с кнопками в
                         главном окне и выводом текста

   Назначение          : реализация класса "MainThread",
                         производного от класса "CWinApp" библиотеки
                         классов MFC (инициализация приложения перед
                         запуском)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAfx.h"                  // Прекомпилируемый файл
#include "MainThread.hpp"            // Объявление класса MainThread
#include "FrameWnd.hpp"              // Объявление класса FrameWnd

// Переопределение виртуальной функции для инициализации приложения
//   перед запуском
BOOL MainThread :: InitInstance( void )
{
    // Указатель на фрейм окна
    FrameWnd            *pFrame;

    // Размещение фрейма окна в динамической памяти
    pFrame = new FrameWnd;
    ASSERT_VALID( pFrame );          // Обработка ошибки размещения

    // Создание окна приложения
    BOOL                rc = pFrame->Create(
        // Используется предопределенное OS WINDOWS имя оконного
        //   класса
        NULL,
        // Заголовок фрейма окна
        "Демонстрация MFC-кнопок",
        // Стиль окна
        WS_SIZEBOX | WS_POPUPWINDOW | WS_DLGFRAME,
        // Местоположение и размеры окна: 0 — левая сторона,
        //   0 — верх, 530 — правая сторона, 290 — низ окна
        CRect( 0, 0, 530, 290 ) );
```

```
    if( !rc )
    {
        TRACE0( "\n Ошибка 1. Фрейм окна не был создан \n" );
        exit( 1 );
    }
    /*
    Содержимое строки, указанной в круглых скобках макроса TRACE0,
    выводится в окно Debug. В него же выводится код
    завершения приложения. Чтобы это происходило, нужно запускать
    приложение в режиме отладки (F5)
    */

    // Создание кнопок
    pFrame->CreateChildControls( );
    // Отображение окна
    pFrame->ShowWindow( SW_SHOW );

    // Созданный фрейм делаем главным окном приложения
    this->m_pMainWnd = pFrame;

    return TRUE;
}
```

Листинг 5.3. Файл FrameWnd.hpp

```
/*
   Файл                : FrameWnd.hpp

   Проект              : демонстрация создания Windows-приложения на
                         основе библиотеки классов MFC с кнопками в
                         главном окне и выводом текста

   Назначение          : объявление класса "FrameWnd", производного
                         от класса "CFrameWnd" библиотеки классов
                         MFC (обработка сообщений главного окна)

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _FrameWnd_hpp
    #define _FrameWnd_hpp
```

```
    // Объявление класса
    class FrameWnd : public CFrameWnd
    {
        // Данные

    private:

        // Указатель на стандартную нажимаемую кнопку (выход из
        //   приложения)
        CButton         *m_pBtnExit;
        // Указатель на стандартную нажимаемую кнопку (MessageBox,
        //   выбрана по умолчанию)
        CButton         *m_pBtnAbout;

        // Методы

    public:

        // Конструктор по умолчанию (инициализация указателей на
        //   кнопки)
        FrameWnd( void );
        // Деструктор (освобождает память, занятую кнопками)
        ~FrameWnd( void );

        // Создание кнопок (перегружаемый виртуальный метод)
        virtual void CreateChildControls( void );

    protected:

        // Так нужно записывать прототипы функций, обрабатывающих
        //   сообщения
        // Перерисовка окна
        afx_msg void OnPaint( void );
        // Обработчик кнопки "О кнопке"
        afx_msg void OnBtnAboutClick( void );
        // Обработчик кнопки "Выход"
        afx_msg void OnBtnExitClick( void );

        // Объявление таблицы сообщений
        DECLARE_MESSAGE_MAP( );
    };

#endif
```

Листинг 5.4. Файл FrameWnd.cpp

```
/*
   Файл                : FrameWnd.cpp

   Проект              : демонстрация создания Windows-приложения на
                         основе библиотеки классов MFC с кнопками в
                         главном окне и выводом текста

   Назначение          : реализация класса "FrameWnd", производного
                         от класса "CFrameWnd" библиотеки классов
                         MFC (обработка сообщений главного окна)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAfx.h"                 // Прекомпилируемый файл
#include "FrameWnd.hpp"             // Объявление класса FrameWnd

// Идентификаторы кнопок
#define IDC_BTNABOUT 100            // Кнопка "О кнопке"
#define IDC_BTNEXIT  101            // Кнопка "Выход"

// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( FrameWnd, CFrameWnd )
    ON_COMMAND( IDC_BTNABOUT, OnBtnAboutClick )
    ON_COMMAND( IDC_BTNEXIT, OnBtnExitClick )
    ON_WM_PAINT( )
END_MESSAGE_MAP( )
/*
   Командные сообщения Windows от меню, акселераторов и кнопок
панелей инструментов обрабатываются макросом
ON_COMMAND( id, memberFn ). Этот макрос в качестве параметров
использует идентификатор команды id и произвольное имя обработчика
команды memberFn. Прототип обработчика должен быть описан в
соответствующем классе (см. файл FrameWnd.hpp).
    Для команд обновления, для извещений, посылаемых дочерними
окнами своим родителям, для элементов редактирования, списков и
комбинированных списков используются другие макросы. Они будут
рассмотрены в последующих проектах
*/

// Конструктор по умолчанию (инициализация указателей на кнопки)
FrameWnd :: FrameWnd( void )
```

```
{
    m_pBtnAbout = 0;
    m_pBtnExit  = 0;
}

// Деструктор (освобождает память, занятую кнопками)
FrameWnd :: ~FrameWnd( void )
{
    if ( m_pBtnAbout )
    {
        delete m_pBtnAbout;
        m_pBtnAbout = 0;
    }
    if ( m_pBtnExit )
    {
        delete m_pBtnExit;
        m_pBtnExit = 0;
    }
}

// Создание кнопок (перегружаемый виртуальный метод)
void FrameWnd :: CreateChildControls( void )
{
    // Размещаем в динамической памяти и создаем кнопку "О кнопке"
    //   (выбрана по умолчанию)
    m_pBtnAbout = new CButton;
    ASSERT_VALID( m_pBtnAbout );    // Обработка ошибки размещения
    BOOL                rc = m_pBtnAbout->Create( "О кнопке",
        WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON,
        CRect( 424, 40, 513, 65 ), this, IDC_BTNABOUT );
    if( !rc )
    {
        TRACE0( "\n Ошибка 2. Кнопка \"О кнопке\" не была"
                " создана \n" );
        exit( 2 );
    }

    // Размещаем в динамической памяти и создаем кнопку "Выход"
    m_pBtnExit = new CButton;
    ASSERT_VALID( m_pBtnExit );     // Обработка ошибки размещения
    rc = m_pBtnExit->Create( "Выход", WS_VISIBLE | WS_CHILD
        | BS_PUSHBUTTON, CRect( 424, 8, 513, 33 ), this,
        IDC_BTNEXIT );
```

```
    if( !rc )
    {
        TRACE0( "\n Ошибка 3. Кнопка \"Выход\" не была"
                " создана \n" );
        exit( 3 );
    }

    return;
}

// Обработчик кнопки "О кнопке"
afx_msg void FrameWnd :: OnBtnAboutClick( void )
{
    // Звуковой сигнал
    BOOL                rc = MessageBeep( -1 );
    /*
    В этой функции можно использовать следующие значения аргумента:
    0xFFFFFFFF — standard beep using the computer speaker
                 (экв. -1),
    MB_ICONASTERISK — SystemAsterisk,
    MB_ICONEXCLAMATION — SystemExclamation,
    MB_ICONHAND — SystemHand,
    MB_ICONQUESTION — SystemQuestion,
    MB_OK — SystemDefault
    */
    if( !rc )
    {
        TRACE0( "\n Ошибка 4. Неверное завершение MessageBeep \n" );
        exit( 4 );
    }
    AfxMessageBox( "Активизирована кнопка \"О кнопке\"" );

    return afx_msg void( );
}

// Обработчик кнопки "Выход"
afx_msg void FrameWnd :: OnBtnExitClick( void )
{
    BOOL                rc = MessageBeep( -1 );
    if( !rc )
    {
        TRACE0( "\n Ошибка 5. Неверное завершение MessageBeep \n" );
        exit( 5 );
    }
```

```
    rc = DestroyWindow( );
    if( !rc )
    {
        TRACE0( "\n Ошибка 6. Окно не было разрушено \n" );
        exit( 6 );
    }

    return afx_msg void( );
}

// Обработчик сообщения WM_PAINT
afx_msg void FrameWnd :: OnPaint( void )
{
    // Создаем объект для рисования и вывода
    CPaintDC            PaintDC( this );

    // Вывод заданного текста (третий аргумент) в определенное
    //   место окна (первый и второй аргументы)
    PaintDC.TextOut( 120, 0, "Обработка сообщения WM_PAINT" );

    return afx_msg void( );
}
```

Сделаем несколько весьма полезных замечаний, а затем проанализируем исходные тексты проекта.

Замечание

Заголовочные (или подключаемые) файлы проекта могут иметь как расширение h, так и расширение hpp. Мастер создания проектов предлагает расширение h *(см. главу 13)*. В рассматриваемых демонстрационных примерах использовано, для унификации, расширение hpp (язык C++ — расширения cpp и hpp, язык C — расширения c и h).

Замечание

Для предотвращения возможности многократного включения заголовочного (подключаемого) файла, наряду с директивами условной компиляции, предусмотрено использование директивы `#pragma once`. В демонстрационных примерах эта возможность не используется, т. к. она не предусмотрена стандартом языка C++, а является особенностью Microsoft Visual C++.

Вкладка **Object Browser** (ее можно сделать видимой с помощью команды **View | Object Browser**) является удобным средством работы с исходным кодом проекта, позволяющим наглядно представить структуру проекта в целом, пользовательских классов проекта и их состав. Вид вкладки для программно-

го проекта WndBtn_MFC, после выбора глобального объекта `MainThread`, показан на рис. 5.7 (обратите внимание на настройку вкладки).

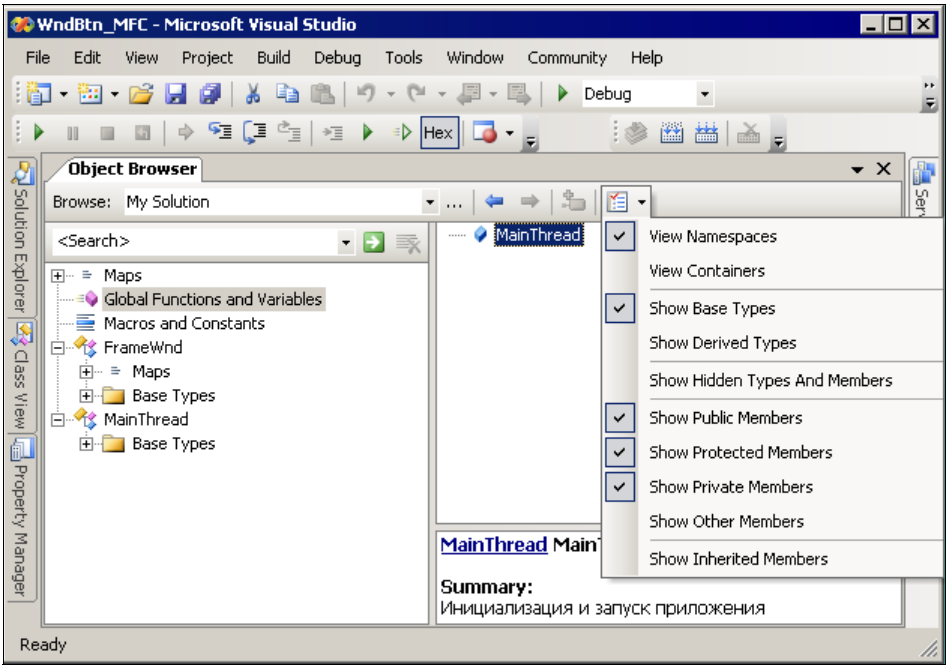


Рис. 5.7. Вкладка **Object Browser** проекта WndBtn_MFC

Из рисунка видно, что проект использует два пользовательских класса: класс `MainThread`, производный от класса `CWinApp` библиотеки классов MFC, класс `FrameWnd`, производный от класса `CFrameWnd` библиотеки MFC, и один глобальный объект `MainThread`. Класс `MainThread` содержит единственный метод `InitInstance` со спецификатором доступа `public` (рис. 5.8).

Класс `FrameWnd` содержит переменные `m_pBtnAbout` и `m_pBtnExit` — члены класса со спецификатором доступа `private`, методы `CreateChildControls`, `LButtonFrame`, `~LButtonFrame` со спецификатором доступа `public` и методы `OnBtnAboutClick`, `OnBtnExitClick` и `OnPaint` со спецификатором доступа `protected` (рис. 5.9). Обратите внимание, что в правом окне вкладки **Object Browser** перечисляются сначала методы, а потом переменные класса. И те и другие следуют в определенном порядке — сначала общедоступные, потом защищенные и в конце закрытые. Посмотрите и запомните, какими пиктограммами снабжаются члены класса в списке. Если во вкладке **Object Browser** "щелкнуть" мышью по названию класса, то в активном окне редактирования появится файл с объявлением соответствующего класса, причем

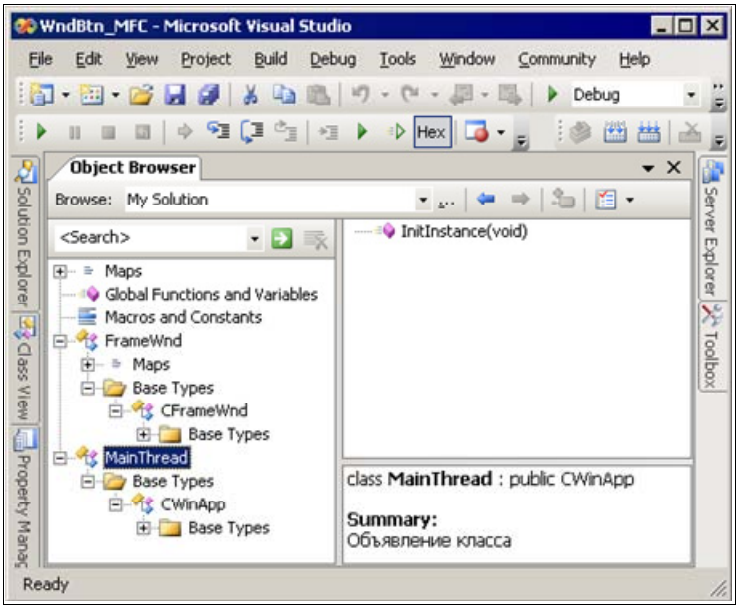


Рис. 5.8. Состав класса `MainThread`

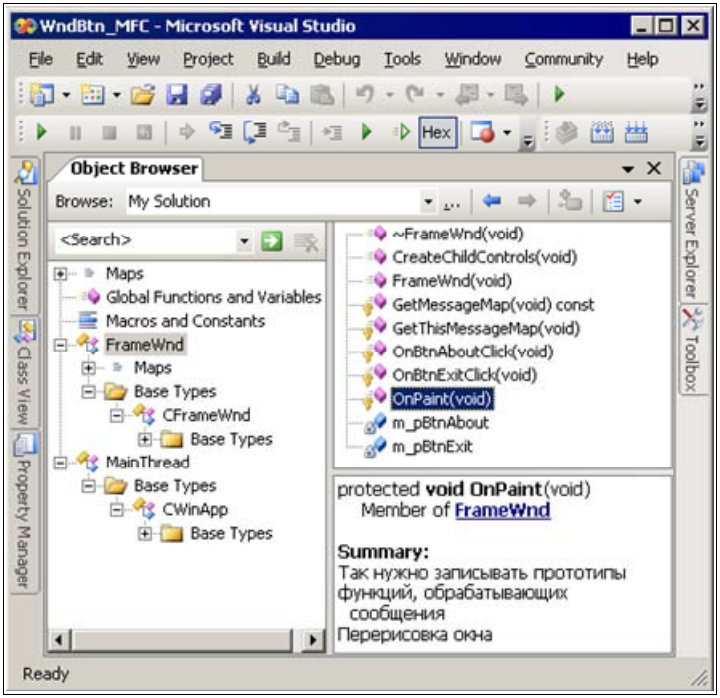


Рис. 5.9. Состав класса `FrameWnd`

текущей строкой будет строка заголовка класса. Аналогично, используя эту вкладку, можно поместить, в активное окно редактирования, файл с текущей строкой, соответствующей началу определения члена класса, который был выбран во вкладке **Object Browser**. Вы заметили как это удобно? Советуем всегда пользоваться этой вкладкой.

Совет

Посмотрите, как выглядит главное окно приложения и как действуют кнопки, расположенные в нем.

5.2.1. Создание управляющих элементов-кнопок и их обработка

В главном окне программного проекта WndBtn_MFC (рис. 5.10) создаются две кнопки, причем размещаются они в динамической памяти. По нажатию кнопки **О кнопке** выдается информационный бокс сообщения, а по кнопке **Выход** приложение завершает свою работу.

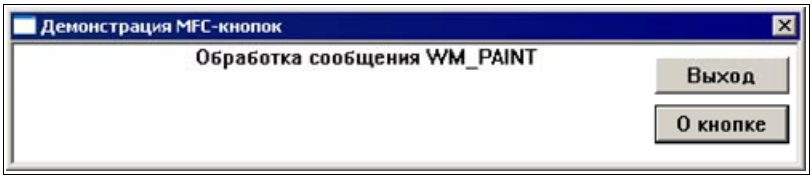


Рис. 5.10. Главное окно проекта WndBtn_MFC

Управляющие элементы-кнопки являются объектами, производными от класса `CObject` библиотеки классов MFC. Рассмотрим создание и обработку кнопок. В данном проекте создание кнопок выполняет перегружаемый виртуальный метод:

```
void FrameWnd :: CreateChildControls( void )
{
    // Размещаем в динамической памяти и создаем кнопку "О кнопке"
    //   (выбрана по умолчанию)
    m_pBtnAbout = new CButton;
    ASSERT_VALID( m_pBtnAbout );    // Обработка ошибки размещения
    BOOL                rc = m_pBtnAbout->Create( "О кнопке",
        WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON,
        CRect( 424, 40, 513, 65 ), this, IDC_BTNABOUT );
    if( !rc )
    {
        TRACE0( "\n Ошибка 2. Кнопка \"О кнопке\" не была"
                " создана \n" );
```

```
        exit( 2 );
    }

    // Размещаем в динамической памяти и создаем кнопку "Выход"
    m_pBtnExit = new CButton;
    ASSERT_VALID( m_pBtnExit );     // Обработка ошибки размещения
    rc = m_pBtnExit->Create( "Выход", WS_VISIBLE | WS_CHILD
        | BS_PUSHBUTTON, CRect( 424, 8, 513, 33 ), this,
        IDC_BTNEXIT );
    if( !rc )
    {
        TRACE0( "\n Ошибка 3. Кнопка \"Выход\" не была"
                " создана \n" );
        exit( 3 );
    }

    return;
}
```

Каждая из кнопок вначале размещается в динамической памяти, а затем создается с помощью метода `Create`. Первый аргумент в вызове этого метода задает текст на кнопке, второй — стиль кнопки, третий — местоположение и размеры кнопки, а последний — ее идентификатор. Обратите внимание на то, что стиль кнопки комбинирует стили окон и кнопок, которые были рассмотрены ранее. Использование стиля `WS_VISIBLE` делает кнопки видимыми сразу после их создания. Для получения более подробной справки об этом методе можно поступить обычным для IDE способом — поместить на название метода `Create` курсор и нажать клавишу <F1>.

Функции, обрабатывающие нажатие кнопок, задаются в проекте с помощью таблицы сообщений (см. листинги 5.3, 5.4):

```
class FrameWnd : public CFrameWnd   // Из FrameWnd.hpp
{
    //...

    protected:

        // Так нужно записывать прототипы функций, обрабатывающих
        //   сообщения
        // Перерисовка окна
        afx_msg void OnPaint( void );
        // Обработчик кнопки "О кнопке"
        afx_msg void OnBtnAboutClick( void );
```

```
        // Обработчик кнопки "Выход"
        afx_msg void OnBtnExitClick( void );

        // Объявление таблицы сообщений
        DECLARE_MESSAGE_MAP( );
    };

// Идентификаторы кнопок — фрагмент файла FrameWnd.cpp
#define IDC_BTNABOUT 100            // Кнопка "О кнопке"
#define IDC_BTNEXIT  101            // Кнопка "Выход"

// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( FrameWnd, CFrameWnd )
    ON_COMMAND( IDC_BTNABOUT, OnBtnAboutClick )
    ON_COMMAND( IDC_BTNEXIT, OnBtnExitClick )
    ON_WM_PAINT( )
END_MESSAGE_MAP( )
```

Командные сообщения Windows от меню, акселераторов и кнопок панелей инструментов обрабатываются макросом `ON_COMMAND( id, memberFn )`. Этот макрос, в качестве параметров, использует идентификатор команды `id` и имя обработчика команды `memberFn`. Прототип обработчика должен быть описан в соответствующем классе (см. файл FrameWnd.hpp). Для команд обновления, извещений (посылаемых дочерними окнами своим родителям), элементов редактирования, списков и комбинированных списков используются другие макросы. Они будут рассмотрены в последующих проектах. В данном приложении обработка кнопок выполняется следующим образом:

```
// Обработчик кнопки "О кнопке" — фрагмент файла FrameWnd.cpp
afx_msg void FrameWnd :: OnBtnAboutClick( void )
{
    // Звуковой сигнал
    BOOL                rc = MessageBeep( -1 );
    /*
    В этой функции можно использовать следующие значения аргумента:
    0xFFFFFFFF — standard beep using the computer speaker
                 (экв. -1),
    MB_ICONASTERISK — SystemAsterisk,
    MB_ICONEXCLAMATION — SystemExclamation,
    MB_ICONHAND — SystemHand,
    MB_ICONQUESTION — SystemQuestion,
    MB_OK — SystemDefault
    */
    if( !rc )
```

```
    {
        TRACE0( "\n Ошибка 4. Неверное завершение MessageBeep \n" );
        exit( 4 );
    }
    AfxMessageBox( "Активизирована кнопка \"О кнопке\"" );

    return afx_msg void( );
}

// Обработчик кнопки "Выход"
afx_msg void FrameWnd :: OnBtnExitClick( void )
{
    BOOL                rc = MessageBeep( -1 );
    if( !rc )
    {
        TRACE0( "\n Ошибка 5. Неверное завершение MessageBeep \n" );
        exit( 5 );
    }
    rc = DestroyWindow( );
    if( !rc )
    {
        TRACE0( "\n Ошибка 6. Окно не было разрушено \n" );
        exit( 6 );
    }

    return afx_msg void( );
}
```

Совет

Рекомендуем создать копию демонстрационного проекта WndBtn_MFC и поэкспериментировать с этим проектом в части, относящейся к созданию кнопок и их обработке (стили кнопок, их размеры и расположение, обработка кнопок, их добавление и удаление и т. п.).

Примечание

Отличительной чертой рассмотренного проекта является обработка ошибок выполнения стандартных функций. Для обработки ошибок используется макрос `ASSERT_VALID`, описанный в листинге 4.2, и макрос `TRACE0`, описанный в листинге 5.2. Используйте эти отладочные средства в ваших проектах.

Совет

Для вывода сообщения на экран используется функция `AfxMessageBox`. Изучите ее, пользуясь встроенной справкой IDE, и используйте ее в ваших проектах, основанных на MFC.

5.3. Низкоуровневое Windows-приложение с дочерними кнопками и окнами

Рассмотрим, в качестве примера, Windows-приложение, главное окно которого содержит некоторый текст и две кнопки. При нажатии одной из них создается дочернее окно с выводом соответствующего текста, а при нажатии другой — приложение завершает свою работу.

Соответствующий программный проект имеется на прилагаемом компакт-диске, в папке \Глава 05\WndWndBtn, а его файловый состав представлен на рис. 5.11.

Рис. 5.11. Файловый состав проекта WndWndBtn

Совет

Проект WndWndBtn создавайте аналогично низкоуровневому проекту FrameWnd из *главы 2*, используя в качестве основы его файлы: StdAfx.h, StdAfx.cpp (эти файлы не изменяются) и FrameWnd.cpp (переименуйте его в файл WndWndBtn.cpp и модифицируйте в соответствии с листингом 5.5). Более подробные сведения о создании проекта см. в *приложении 3*.

Листинг 5.5. Файл WndWndBtn.cpp

```
/*
   Файл                : WndWndBtn.cpp

   Проект              : приложение Win32API с использованием
                         стандартных функций API (кнопки и
                         дочерние окна в главном окне)

   Microsoft Visual Studio C++ .NET 2005
*/
```

```
// Прекомпилируемый заголовочный файл
#include "stdafx.h"

// Прототипы используемых функций
LRESULT CALLBACK WndProc( HWND hwnd, UINT message, WPARAM wParam,
                          LPARAM lParam );
BOOL InitApplication( HINSTANCE hInstance );
BOOL InitInstance( HINSTANCE hInstance, int nCmdShow );
LRESULT CALLBACK WndProc1( HWND hwnd, UINT message, WPARAM wParam,
                           LPARAM lParam );

// Дескриптор (уникальное 32-битное беззнаковое целое),
//   ассоциируемый с текущим приложением
HINSTANCE               hInst;
// Дескриптор главного окна Windows
HWND                    hWnd;
// Дескриптор кнопки "Новое окно"
HWND                    hWnd1;
HWND                    hWnd2;      // Дескриптор кнопки "Выход"
// Идентификатор кнопки "Новое окно"
#define BTN1 1
#define BTN2 2                      // Идентификатор кнопки "Выход"

// ****************************************************************
// Главная функция приложения
int WINAPI WinMain(                 // Возвращает TRUE при успехе
    // Дескриптор данного приложения
    HINSTANCE           hInstance,
    // Дескриптор предыдущей запущенной копии приложения.
    //   В Win32API этот параметр всегда равен NULL и оставлен
    //   исключительно для совместимости с версиями ниже четвертой.
    //   Связано это с тем, что каждое 32-разрядное приложение
    //   запускается в собственном адресном пространстве, в
    //   котором, естественно, нет никаких копий или других
    //   приложений
    HINSTANCE           hPrevInstance,
    // Указатель на командную строку, которую можно в
    //   интегрированной среде разработки задать по команде
    //   "Properties" контекстного меню проекта в элементе "Program
    //   arguments" вкладки "Debugging"
    LPSTR               lpCmdLine,
    // Режим начального отображения главного окна приложения —
    //   после вызова параметр получает значение SW_SHOWNORMAL (1),
    //   что соответствует отображению окна в нормальном виде
```

```
    int                 nCmdShow )
{
    // Передаем дескриптор приложения на глобальный уровень
    hInst = hInstance;

    // Инициализируем приложение — подготавливаем данные классов
    //   окон и регистрируем их
    if ( !InitApplication( hInstance ) )
        return FALSE;

    // Завершаем создание приложения — создаем и отображаем главное
    //   окно приложения с текстом и кнопками
    if ( !InitInstance( hInstance, nCmdShow ) )
        return FALSE;

    MSG                 msg;        // Для очередного сообщения
    // Стандартный цикл обработки сообщений
    while ( GetMessage( &msg, NULL, 0, 0 ) )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }

    return static_cast< int >( msg.wParam );
}

// ****************************************************************
// Регистрация классов окон
BOOL InitApplication(               // Возвращает TRUE при успешном
                                    //   завершении
    // Дескриптор (уникальное число), ассоциируемый с текущим
    //   приложением
    HINSTANCE           hInstance )
{
    // Сведения о регистрируемом классе
    WNDCLASS            wc;

    // Заполняем структуру класса главного окна
    wc.style         = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc   = static_cast< WNDPROC >( WndProc );
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = hInstance;
```

```
    wc.hIcon         = LoadIcon( NULL, IDI_ASTERISK );
    wc.hCursor       = LoadCursor( NULL, IDC_CROSS );
    wc.hbrBackground = reinterpret_cast< HBRUSH >( COLOR_WINDOW+1 );
    wc.lpszMenuName  = NULL;
    wc.lpszClassName = "WndWndBtnAPI";

    // Регистрируем класс главного окна
    if ( !RegisterClass( &wc ) )
        return FALSE;

    // Заполняем структуру класса дочернего окна
    wc.style         = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc   = static_cast< WNDPROC >( WndProc1 );
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = hInstance;
    wc.hIcon         = LoadIcon( NULL, IDI_ASTERISK );
    wc.hCursor       = LoadCursor( NULL, IDC_CROSS );
    wc.hbrBackground = reinterpret_cast< HBRUSH >( COLOR_WINDOW+1 );
    wc.lpszMenuName  = NULL;
    wc.lpszClassName = "WndAPI";

    // Регистрируем класс дочернего окна
    return RegisterClass( &wc );

    // !!! Класс окна для кнопок ("BUTTON") является стандартным и
    //   не требует специальной регистрации !!!
}

// *************************************************************
// Создание главного окна приложения с текстом и кнопками
BOOL InitInstance(                  // Возвращает TRUE при успехе
    // Дескриптор текущего приложения
    HINSTANCE           hInstance,
    // Режим отображения главного окна — определяет, в каком виде
    //   будет отображено окно приложения
    int                 nCmdShow )
{
    // Создание главного окна
    hWnd = CreateWindow(
        // Имя зарегистрированного класса
        "WndWndBtnAPI",
```

```
    // Заголовок окна
    "Приложение WndWndBtn (Win32API) с кнопками и дочерним "
    "окном",
    // Стиль окна
    WS_VISIBLE | WS_OVERLAPPEDWINDOW,
    // Горизонтальная координата левого верхнего угла окна
    //   (используется умалчиваемое значение)
    CW_USEDEFAULT,
    // Вертикальная координата левого верхнего угла окна
    //   (используется умалчиваемое значение)
    CW_USEDEFAULT,
    CW_USEDEFAULT,              // Ширина окна (используется
                                //   умалчиваемое значение)
    CW_USEDEFAULT,              // Высота окна (используется
                                //   умалчиваемое значение)
    NULL,                       // Дескриптор родительского
                                //   окна (его нет)
    NULL,                       // Дескриптор меню окна (его
                                //   нет)
    hInstance,                  // Дескриптор экземпляра
                                //   приложения
    NULL );                     // Указатель на дополнительные
                                //   данные окна (их нет)
if ( !hWnd )
    return FALSE;

// Создание кнопки "Новое окно"
hWnd1 = CreateWindow(
    // Имя зарегистрированного класса (в данном случае это
    //   стандартное имя)
    "BUTTON",
    "Новое окно",               // Текст на кнопке
    // Стиль окна (WS_VISIBLE использовать обязательно), кнопка
    //   является дочерним окном
    WS_VISIBLE | WS_CHILD,
    // Горизонтальная координата левого верхнего угла кнопки
    210,
    // Вертикальная координата левого верхнего угла кнопки
    20,
    80,                         // Ширина кнопки
    20,                         // Высота кнопки
    // Дескриптор родительского окна
    hWnd,
```

```
    // Для кнопки — ее идентификатор (см. определение на
    //   глобальном уровне)
    reinterpret_cast< HMENU >( BTN1 ),
    // Дескриптор экземпляра приложения
    hInstance,
    // Указатель на дополнительные данные окна (они
    //   отсутствуют)
    NULL );
if ( !hWnd1 )
{
    MessageBox( NULL, "Кнопка \"Новое окно\" не создана",
                "Ошибка 1", MB_OK );
    exit( 1 );
}

// Создание кнопки "Выход"
hWnd2 = CreateWindow(
    // Имя зарегистрированного класса (в данном случае это
    //   стандартное имя)
    "BUTTON",
    "Выход",                    // Текст на кнопке
    // Стиль окна (WS_VISIBLE использовать обязательно), кнопка
    //   является дочерним окном
    WS_VISIBLE | WS_CHILD,
    // Горизонтальная координата левого верхнего угла кнопки
    210,
    // Вертикальная координата левого верхнего угла кнопки
    50,
    80,                         // Ширина кнопки
    20,                         // Высота кнопки
    // Дескриптор родительского окна
    hWnd,
    // Для кнопки — ее идентификатор (см. определение на
    //   глобальном уровне)
    reinterpret_cast< HMENU >( BTN2 ),
    // Дескриптор экземпляра приложения
    hInstance,
    // Указатель на дополнительные данные окна (они
    //   отсутствуют)
    NULL );
if ( !hWnd2 )
{
    MessageBox( NULL, "Кнопка \"Выход\"не создана", "Ошибка 2",
                MB_OK );
```

```
        exit( 2 );
    }

    return TRUE;
}

// ****************************************************************
// Оконная процедура главного окна
// Получает очередное сообщение и индивидуально обрабатывает его
LRESULT CALLBACK WndProc(           // Возвращает результат
                                    //   обработки сообщения,
                                    //   зависящий от посланного
                                    //   сообщения: 0 — успех
    HWND                hwnd,       // Дескриптор созданного окна
    UINT                message,    // Номер сообщения
    WPARAM              wParam,     // Дополнительная информация о
    LPARAM              lParam )    //   сообщении, зависящая от
                                    //   типа сообщения (wParam,
                                    //   lParam)
{

    // Обработчик сообщения — в данном случае приложение явно
    //   отвечает только на сообщения WM_DESTROY, WM_PAINT и
    //   WM_COMMAND от нажатия кнопок, все остальные сообщения
    //   передаются в DefWindowProc( ) — функцию, управляющую
    //   поведением окна по умолчанию. Обработка сообщений
    //   WM_DESTROY и WM_PAINT подробно рассмотрена в проекте
    //   FrameWnd
    switch( message )
    {
    case WM_PAINT:

        // Дескриптор контекста устройства
        HDC             hDC;
        // Указатель на структуру с информацией для приложения
        //   о клиентской области окна
        PAINTSTRUCT     ps;
        // Получение контекста устройства
        hDC = BeginPaint( hwnd, &ps );
        if ( !hDC )
        {
            MessageBox( NULL, "Контекст устройства не получен",
                        "Ошибка 3", MB_OK );
```

```
            exit( 3 );
        }
        // Вывод заданного текста
        if ( !TextOut( hDC, 130, 0,
             "Обработка сообщения WM_PAINT", 28 ) )
        {
            MessageBox( NULL, "Неверное использование TextOut",
                        "Ошибка 4", MB_OK );
            exit( 4 );
        }
        // Освобождение контекста устройства
        if ( !EndPaint( hwnd, &ps ) )
        {
            MessageBox( NULL, "Контекст устройства не освобожден",
                        "Ошибка 5", MB_OK );
            exit( 5 );
        }
        break;

    case WM_DESTROY:

        // Указывает системе, что сделан запрос о завершении
        //   приложения: 0 — код завершения. Вызов этой функции
        //   обычно используется в ответ на поступившее
        //   сообщение WM_DESTROY
        PostQuitMessage( 0 );
        break;

    case WM_COMMAND:
        {
            switch( LOWORD( wParam ) )
            {
            // Нажата кнопка "Дочернее окно"
            case BTN1:

                // Звуковой сигнал
                if ( !MessageBeep( MB_OK ) )
                {
                    MessageBox( NULL, "Ошибка MessageBeep",
                                "Ошибка 6", MB_OK );
                    exit( 6 );
                }
                // Дескриптор дочернего окна
                HWND    hWnd3;
```

```
        // Создание дочернего окна
        hWnd3 = CreateWindow(
            // Имя зарегистрированного класса
            "WndAPI",
            // Заголовок окна
            "Дочернее окно",
            // Стиль окна
            WS_VISIBLE | WS_OVERLAPPEDWINDOW,
            // Горизонтальная координата левого верхнего
            //   угла окна
            CW_USEDEFAULT,
            // Вертикальная координата левого верхнего
            //   угла окна
            CW_USEDEFAULT,
            // Ширина окна
            CW_USEDEFAULT,
            // Высота окна
            CW_USEDEFAULT,
            hWnd,           // Дескриптор родительского
                            //   окна
            NULL,           // Дескриптор меню окна
            hInst,          // Дескриптор экземпляра
                            //   приложения
            NULL );         // Указатель на дополнительные
                            //   данные окна
        if ( !hWnd3 )
        {
            MessageBox( NULL, "Дочернее окно не создано",
                        "Ошибка 7", MB_OK );
            exit( 7 );
        }
        break;

    case BTN2:              // Нажата кнопка "Выход"

        // Указывает системе, что сделан запрос о
        //   завершении приложения: 0 — код
        //   завершения. Вызов этой функции обычно
        //   используется в ответ на поступившее
        //   сообщение WM_DESTROY
        PostQuitMessage( 0 );
        break;
    }
```

```
        }
        break;

    default:

        // Обработка сообщения по умолчанию
        return DefWindowProc( hwnd, message, wParam, lParam );

    }

    return 0;
}

// ************************************************************
// Оконная процедура дочернего окна.
// Получает очередное сообщение и индивидуально обрабатывает его
LRESULT CALLBACK WndProc1(          // Возвращает результат
                                    //   обработки сообщения,
                                    //   зависящий от посланного
                                    //   сообщения: 0 — успех
    HWND                hwnd,       // Дескриптор созданного окна
    UINT                message,    // Номер сообщения
    WPARAM              wParam,     // Дополнительная информация о
    LPARAM              lParam )    //   сообщении, зависящая от
                                    //   типа сообщения (wParam,
                                    //   lParam)
{

    // Обработчик сообщений — в данном случае приложение явно
    //   отвечает только на сообщение WM_PAINT, все остальные
    //   сообщения передаются в DefWindowProc( )  — функцию,
    //   управляющую поведением окна по умолчанию
    switch( message )
    {
    case WM_PAINT:

        // Дескриптор контекста устройства
        HDC             hDC;
        // Указатель на структуру с информацией для приложения
        //   о клиентской области окна
        PAINTSTRUCT     ps;
        // Получение контекста устройства
        hDC = BeginPaint( hwnd, &ps );
        if ( !hDC )
```

```
        {
            MessageBox( NULL, "Контекст устройства не получен",
                        "Ошибка 8", MB_OK );
            exit( 8 );
        }
        // Вывод заданного текста
        if ( !TextOut( hDC, 130, 0,
             "Обработка сообщения WM_PAINT в дочернем окне", 44 ) )
        {
            MessageBox( NULL, "Неверное использование TextOut",
                        "Ошибка 9", MB_OK );
            exit( 9 );
        }
        // Освобождение контекста устройства
        if ( !EndPaint( hwnd, &ps ) )
        {
            MessageBox( NULL, "Контекст устройства не освобожден",
                        "Ошибка 10", MB_OK );
            exit( 10 );
        }
        break;

    default:

        // Обработка сообщения по умолчанию
        return DefWindowProc( hwnd, message, wParam, lParam );

    }

    return 0;
}
```

Теперь проанализируем приведенный исходный код.

Совет

Прежде всего, обратите внимание на тщательную *обработку* возможных *ошибок* использования стандартных функций. Никогда в своих проектах не пренебрегайте такой обработкой.

Данное оконное приложение очень похоже на рассмотренное приложение WndBtn, исходный код которого приведен в листинге 5.1. По этой причине приведем только краткое описание отличий.

После запуска приложения на экране возникает его главное окно, показанное на рис. 5.12. Окно содержит две кнопки — два дочерних окна.

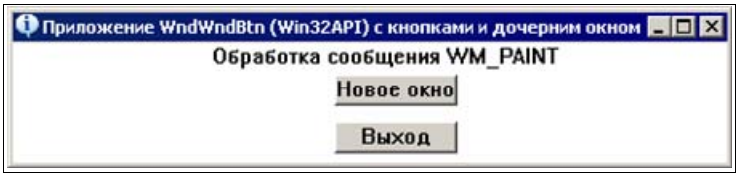


Рис. 5.12. Главное окно приложения WndWndBtn

По нажатию кнопки **Новое окно** создается дочернее окно, вид которого показан на рис. 5.13, а при нажатии кнопки **Выход** приложение завершает свою работу.

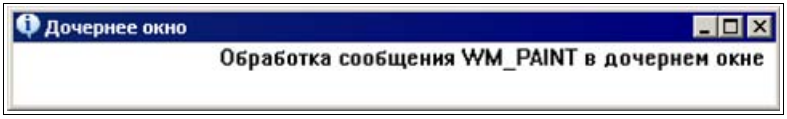


Рис. 5.13. Дочернее окно приложения WndWndBtn

На внешнем уровне в исходном тексте приложения (см. листинг 5.5) изменился состав дескрипторов и идентификаторов кнопок. Функция `WinMain` полностью сохранила свой вид, а в функции `InitApplication` дополнительно определяются свойства и регистрируется класс дочернего окна (рис. 5.13), создаваемого по нажатию кнопки **Новое окно**. В функции `InitInstance`, в дополнение к созданию и отображению главного окна приложения и одной из кнопок, создается и отображается еще одна кнопка — кнопка **Выход**. В данном приложении вместо единственной оконной процедуры используются *две оконные процедуры* — оконная процедура главного окна `WndProc` и оконная процедура `WndProc1` дочернего окна, создаваемого по нажатию кнопки **Новое окно**. В оконной процедуре главного окна `WndProc` обработка нажатия кнопки **Выход** практически такая же, как и в проекте FrameWnd из *главы 2*, и добавилась обработка нажатия кнопки **Новое окно**. В результате обработки этой кнопки создается дочернее окно (см. рис. 5.13). Код, обеспечивающий создание и отображение дочернего окна, с точностью до обозначений совпадает с кодом, обеспечивающим создание и отображение главного окна приложения. В оконной процедуре дочернего окна `WndProc1` обрабатывается только сообщение `WM_PAINT`, причем делается это аналогично обработке такого же сообщения в оконной процедуре главного окна.

Таким образом, в проекте WndWndBtn используются два окна — главное и дочернее со своей, оригинальной обработкой сообщений. Соответственно этому, в проекте определяются и регистрируются два оконных класса, каждый из которых использует свою оконную процедуру.

5.4. Windows-приложение с дочерними кнопками и окнами на базе библиотеки классов MFC

Для сопоставления и лучшего уяснения особенностей приложения (с дочерними окнами) на базе MFC, как и ранее, рассмотрим, в качестве примера, приложение, построенное с использованием библиотеки классов MFC. Соответствующий учебный проект имеется на прилагаемом компакт-диске, в папке \Глава 05\WndWndBtn_MFC, а его файловый состав показан на рис. 5.14.

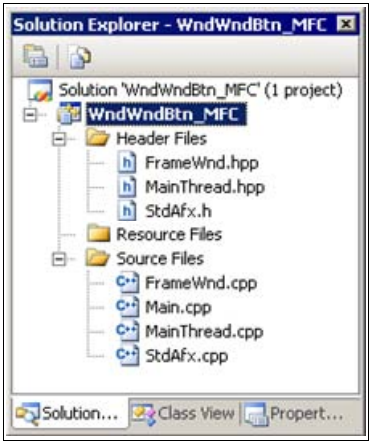


Рис. 5.14. Файловый состав проекта WndWndBtn_MFC

Совет

Проект WndWndBtn_MFC создавайте аналогично проекту WndBtn_MFC из *разд. 5.2*, используя в качестве основы следующие файлы: StdAfx.h, StdAfx.cpp, Main.cpp, MainThread.hpp (не изменяются), а MainThread.cpp, FrameWnd.hpp и FrameWnd.cpp измените, в соответствии с листингами 5.6—5.8. Более подробные сведения о создании проекта см. в *приложении 3*. Для добавления в класс проекта переменной или функции (метода) при модификации файлов, как и ранее, пользуйтесь мастерами (см. рис. 5.5 и 5.6).

Листинг 5.6. Файл MainThread.cpp

```
/*
   Файл              : MainThread.cpp

   Проект            : демонстрация создания Windows-приложения на
                       основе библиотеки классов MFC с кнопками и
                       дочерними окнами и выводом текста
```

```
   Назначение          : реализация класса "MainThread",
                         производного от класса "CWinApp" библиотеки
                         классов MFC (инициализация приложения перед
                         запуском)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAfx.h"                  // Прекомпилируемый файл
#include "MainThread.hpp"            // Объявление класса MainThread
#include "FrameWnd.hpp"              // Объявление класса FrameWnd

// Переопределение виртуальной функции для инициализации приложения
//   перед запуском
BOOL MainThread :: InitInstance( void )
{
    // Указатель на фрейм окна
    FrameWnd            *pFrame;

    // Размещение фрейма окна в динамической памяти с обработкой
    //   ошибки
    pFrame = new FrameWnd;
    ASSERT_VALID( pFrame );

    // Создание окна приложения
    BOOL                rc = pFrame->Create(
        // Используется предопределенное OS WINDOWS имя оконного
        //   класса
        NULL,
        // Заголовок фрейма окна
        "Демонстрация MFC-кнопок и дочерних окон",
        // Стиль окна
        WS_SIZEBOX | WS_POPUPWINDOW | WS_DLGFRAME,
        // Местоположение и размеры окна: 0 — левая сторона,
        //   0 — верх, 530 — правая сторона, 290 — низ окна
        CRect( 0, 0, 530, 290 ) );
    if( !rc )
    {
        TRACE0( "\n Ошибка 1. Фрейм окна не был создан \n" );
        exit( 1 );
    }

    // Создание кнопок
    pFrame->CreateChildControls( );
```

```
    // Отображение окна
    pFrame->ShowWindow( SW_SHOW );

    // Созданный фрейм делаем главным окном приложения
    this->m_pMainWnd = pFrame;

    return TRUE;
}
```

Листинг 5.7. Файл FrameWnd.hpp

```
/*
   Файл                : FrameWnd.hpp

   Проект              : демонстрация создания Windows-приложения на
                         основе библиотеки классов MFC с кнопками и
                         дочерними окнами и выводом текста

   Назначение          : объявление класса "FrameWnd", производного
                         от класса "CFrameWnd" библиотеки классов
                         MFC (обработка сообщений главного окна)

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _FrameWnd_hpp
    #define _FrameWnd_hpp

    // Объявление класса
    class FrameWnd : public CFrameWnd
    {
        // Данные

    private:

        // Указатель на стандартную нажимаемую кнопку (выход из
        //   приложения)
        CButton         *m_pBtnExit;
        // Указатель на стандартную нажимаемую кнопку (дочернее
        //   окно, выбрано по умолчанию)
        CButton         *m_pBtnWnd;
```

```
        // Методы

    public:

        // Конструктор по умолчанию (инициализация указателей на
        //   кнопки)
        FrameWnd( void );
        // Деструктор (освобождает память, занятую кнопками)
        ~FrameWnd( void );

        // Создание кнопок (перегружаемый виртуальный метод)
        virtual void CreateChildControls( void );

    protected:

        // Так нужно записывать прототипы функций, обрабатывающих
        //   сообщения
        // Перерисовка окна
        afx_msg void OnPaint( void );
        // Обработчик кнопки "Окно"
        afx_msg void OnBtnWndClick( void );
        // Обработчик кнопки "Выход"
        afx_msg void OnBtnExitClick( void );

        // Объявление таблицы сообщений
        DECLARE_MESSAGE_MAP( );
    };

#endif
```

Листинг 5.8. Файл FrameWnd.cpp

```
/*
   Файл               : FrameWnd.cpp

   Проект             : демонстрация создания Windows-приложения на
                        основе библиотеки классов MFC с кнопками и
                        дочерними окнами и выводом текста

   Назначение         : реализация класса "FrameWnd", производного
                        от класса "CFrameWnd" библиотеки классов
                        MFC (обработка сообщений главного окна)

   Microsoft Visual Studio C++ .NET 2005
*/
```

```
#include "StdAfx.h"                    // Прекомпилируемый файл
#include "FrameWnd.hpp"                // Объявление класса FrameWnd

// Идентификаторы кнопок
#define IDC_BTNWND 100                 // Кнопка "Окно"
#define IDC_BTNEXIT 101                // Кнопка "Выход"

// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( FrameWnd, CFrameWnd )
    ON_COMMAND( IDC_BTNWND, OnBtnWndClick )
    ON_COMMAND( IDC_BTNEXIT, OnBtnExitClick )
    ON_WM_PAINT( )
END_MESSAGE_MAP( )

// Конструктор по умолчанию (инициализация указателей на кнопки)
FrameWnd :: FrameWnd( void )
{
    m_pBtnWnd = 0;
    m_pBtnExit = 0;
}

// Деструктор (освобождает память, занятую кнопками)
FrameWnd :: ~FrameWnd( void )
{
    if ( m_pBtnWnd )
    {
        delete m_pBtnWnd;
        m_pBtnWnd = 0;
    }
    if ( m_pBtnExit )
    {
        delete m_pBtnExit;
        m_pBtnExit = 0;
    }
}

// Создание кнопок (перегружаемый виртуальный метод)
void FrameWnd :: CreateChildControls( void )
{
    // Размещаем в динамической памяти и создаем кнопку "Окно"
    //   (выбрана по умолчанию)
    m_pBtnWnd = new CButton;
    ASSERT_VALID( m_pBtnWnd );         // Обработка ошибки размещения
```

```
    BOOL                rc = m_pBtnWnd->Create( "Окно",
        WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON,
        CRect( 424, 40, 513, 65 ), this, IDC_BTNWND );
    if( !rc )
    {
        TRACE0( "\n Ошибка 2. Кнопка \"Окно\" не была"
                " создана \n" );
        exit( 2 );
    }

    // Размещаем в динамической памяти и создаем кнопку "Выход"
    m_pBtnExit = new CButton;
    ASSERT_VALID( m_pBtnExit );     // Обработка ошибки размещения
    rc = m_pBtnExit->Create( "Выход", WS_VISIBLE | WS_CHILD
        | BS_PUSHBUTTON, CRect( 424, 8, 513, 33 ), this,
        IDC_BTNEXIT );
    if( !rc )
    {
        TRACE0( "\n Ошибка 3. Кнопка \"Выход\" не была"
                " создана \n" );
        exit( 3 );
    }

    return;
}

// Обработчик кнопки "Окно"
afx_msg void FrameWnd :: OnBtnWndClick( void )
{
    // Звуковой сигнал
    BOOL                rc = MessageBeep( -1 );
    if( !rc )
    {
        TRACE0( "\n Ошибка 4. Неверное завершение MessageBeep \n" );
        exit( 4 );
    }

    // Указатель на фрейм окна
    FrameWnd            *pFrame;

    // Размещение фрейма окна в динамической памяти с обработкой
    //   ошибки
    pFrame = new FrameWnd;
    ASSERT_VALID( pFrame );
```

```
    // Создание окна приложения
    rc = pFrame->Create(
        // Используется предопределенное OS WINDOWS имя оконного
        //   класса
        NULL,
        // Заголовок фрейма окна
        "Это окно было создано по кнопке \"Окно\"",
        // Стиль окна
        WS_SIZEBOX | WS_POPUPWINDOW | WS_DLGFRAME,
        // Местоположение и размеры окна: 20 — левая сторона,
        //   60 — верх, 430 — правая сторона, 190 — низ окна
        CRect( 20, 60, 430, 190 ) );
    if( !rc )
    {
        TRACE0( "\n Ошибка 5. Фрейм дочернего окна не был "
                "создан \n" );
        exit( 5 );
    }

    // Отображение окна
    pFrame->ShowWindow( SW_SHOW );

    return afx_msg void( );
}

// Обработчик кнопки "Выход"
afx_msg void FrameWnd :: OnBtnExitClick( void )
{
    BOOL                rc = MessageBeep( -1 );
    if( !rc )
    {
        TRACE0( "\n Ошибка 6. Неверное завершение MessageBeep \n" );
        exit( 6 );
    }
    rc = DestroyWindow( );
    if( !rc )
    {
        TRACE0( "\n Ошибка 7. Окно не было разрушено \n" );
        exit( 7 );
    }

    return afx_msg void( );
}
```

```
// Обработчик сообщения WM_PAINT
afx_msg void FrameWnd :: OnPaint( void )
{
    // Создаем объект для рисования и вывода
    CPaintDC            PaintDC( this );

    // Вывод заданного текста (третий аргумент) в определенное
    //   место окна (первый и второй аргументы)
    PaintDC.TextOut( 120, 0, "Обработка сообщения WM_PAINT" );

    return afx_msg void( );
}
```

Теперь кратко проанализируем приведенные исходные коды.

После запуска приложения на экран выводится главное окно приложения (рис. 5.15).

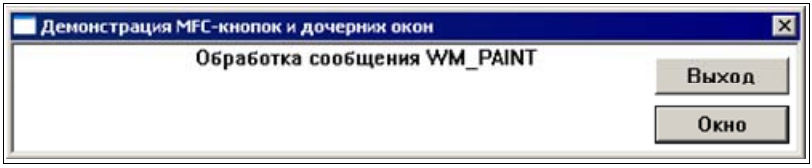


Рис. 5.15. Главное окно проекта WndWndBtn_MFC

В этом проекте новым моментом является лишь другая обработка нажатия кнопки **Окно**, в результате которой создается дочернее окно (рис. 5.16).

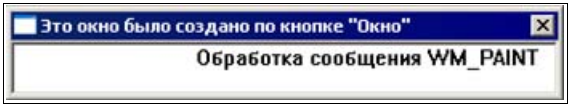


Рис. 5.16. Дочернее окно проекта WndWndBtn_MFC

Замечание

Обратите внимание на то, что сообщения, выдаваемые в главное и дочернее окна, совпадают. Это происходит потому, что в обоих случаях используется один и тот же оконный класс, косвенно определяемый в классе `FrameWnd`, производном от класса `CFrameWnd` библиотеки MFC. Этим данный проект отличается от низкоуровневого проекта-прототипа.

При создании дочернего окна используется стандартная последовательность действий, аналогичная используемой при создании главного окна приложения. Сначала в динамической памяти размещается объект класса `FrameWnd`,

затем с помощью метода `Create` создается окно и посредством метода `ShowWindow` выводится на экран.

5.5. Вопросы и упражнения для самопроверки

Для самопроверки ответьте на следующие вопросы:

1. Следует ли определять в низкоуровневых приложениях свойства и регистрировать класс, используемый при создании кнопок?
2. Опишите и укажите назначение параметров API функции `MessageBox`.
3. Укажите, какие элементарные стили можно использовать при создании кнопок.
4. Как, по сообщению `WM_COMMAND`, определить источник этого сообщения?
5. Укажите назначение макроса `TRACE0`, используемого в приложениях, базирующихся на MFC.
6. Какой метод используется в приложениях на базе MFC для создания кнопок (дочерних управляющих элементов)?
7. Чем отличаются методы `MessageBox` и `AfxMessageBox`, используемые в приложениях на базе MFC?
8. Укажите назначение вкладки **Object Browser**.
9. Укажите порядок создания кнопки.
10. Как в приложениях на базе MFC задаются функции, обрабатывающие нажатия кнопок?
11. Приложение, созданное с помощью функций API, имеет несколько окон. Сколько в нем нужно определить и зарегистрировать классов и сколько использовать оконных процедур?
12. Приложение на базе MFC имеет несколько окон с различной обработкой сообщений. Сколько в нем нужно использовать классов с таблицами сообщений, производных от класса `CFrameWnd` библиотеки MFC?
13. Укажите порядок создания дочернего окна приложения на базе MFC.

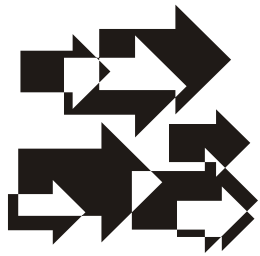
Ответы на вопросы можно проверить — они приведены в *разд. П1.5 приложения 1*.

Совет

Одним из указанных в *главе 2* и *приложении 3* способов создайте копии проектов, рассмотренных ранее в этой главе.

При экспериментах с созданными копиями проектов выполните следующие упражнения:

1. Задавая различные стили кнопок и их комбинации (константы, задающие стили, начинаются с префикса `WS_` или с префикса `BS_`), изучите их влияние на свойства кнопок.
2. Задавая различные варианты начального отображения окна — различные значения аргумента в вызове метода `ShowWindow` (константы, задающие вид начального отображения, начинаются с префикса `SW_`), изучите их влияние на свойства окна.
3. Реализуйте вывод текста и графических объектов в окно приложения.



ГЛАВА 6

Ресурсы: меню, ускорители и таблица строк. "Горячие" клавиши

Новым в этой главе является использование *редактора ресурсов* для создания и настройки меню, акселераторов (ускорителей), таблицы строк и "горячих" клавиш. Обратите на них первоочередное внимание.

6.1. Ресурсы. Редактор ресурсов

Ресурсы являются важной составной частью Windows-приложений и очень удобным инструментом программиста. С ресурсами удобно работать, используя встроенный в IDE редактор ресурсов. Стандартные ресурсы IDE содержат (рис. 6.1):

- ускорители или акселераторы (Accelerator);
- растровые изображения (Bitmap);
- курсоры (Cursor);
- диалоговые окна (Dialog);
- пиктограммы (Icon);
- меню (Menu);
- таблицы строк (String Table);
- панели инструментов (Toolbar);
- версию (Version) и др.

Ресурсы представляют собой *кластеры бинарных данных*, "пристегнутые" во время компоновки к концу выполняемого файла приложения [6]. Когда Windows запускает приложение, она загружает в память только то, что ей необходимо в данный момент. Обычно ресурсы находятся в конце исполняемого exe-файла до тех пор, пока Windows не затребует их. Большинство ресур-

сов — это выгружаемые данные только для чтения, которые операционная система Windows может загрузить в память или выгрузить оттуда в любой момент, если изменятся требования к системной памяти.

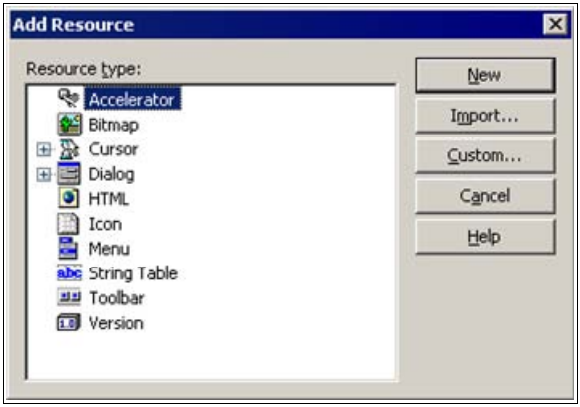


Рис. 6.1. Состав ресурсов ИСР

В этой главе мы рассмотрим использование таких ресурсов, как акселераторы, меню, "горячие" клавиши и таблицы строк. Взаимодействие пользователя с приложением может осуществляться как с помощью команд меню, так и с помощью клавиатурных ускорителей (акселераторов).

Ускорители позволяют пользователю более оперативно активизировать требуемые действия, чем команды меню.

Таблицы строк хранят строковые ресурсы отдельно от основного исходного кода на языке C++. Их легко модифицировать и переводить на другие языки, что обеспечивает легкий способ создания многоязычных приложений, имеющих один и тот же программный код.

Далее, для сравнения, в последний раз рассматриваются варианты реализации демонстрационной программы с меню, ускорителями и таблицей строк с использованием Windows API и библиотеки классов MFC, что, как и ранее, позволит выполнить их сравнительный анализ и лучше понять анатомию приложений на базе MFC. Для обоих вариантов реализации проектов приводится их полный исходный код, а затем проводится их анализ.

6.2. Приложение на базе API с ускорителями и меню

Рассматриваемый далее демонстрационный проект имеется на прилагаемом компакт-диске, в папке \Глава 06\UsgMenu, а его файловый состав представлен на рис. 6.2.

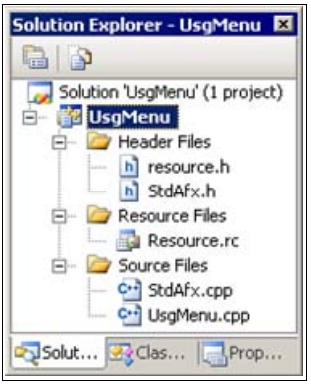


Рис. 6.2. Файловый состав проекта UsgMenu

Совет

Копию проекта UsgMenu создавайте аналогично низкоуровневому проекту FrameWnd из *главы 2*, используя в качестве основы файлы: StdAfx.h, StdAfx.cpp (эти файлы не изменяются) и FrameWnd.cpp (его переименуйте в UsgMenu.cpp и модифицируйте в соответствии с листингом 6.1). Два других файла проекта — resource.h и Resource.rc — создаются и поддерживаются редактором ресурсов, о чем будет сказано далее. Более подробные сведения о создании проекта см. в *приложении 3*.

Листинг 6.1. Файл UsgMenu.cpp

```
/*
   Файл                : UsgMenu.cpp

   Проект              : приложение Win32API с использованием
                         стандартных функций API (акселераторы,
                         меню и "горячие" клавиши)

   Microsoft Visual Studio C++ .NET 2005
*/

// Прекомпилируемый заголовочный файл
#include "stdafx.h"

#include "resource.h"               // Идентификаторы ресурсов

// Прототипы используемых функций
LRESULT CALLBACK WndProc( HWND hwnd, UINT message, WPARAM wParam,
                          LPARAM lParam );
```

```
BOOL InitApplication( HINSTANCE hInstance );
BOOL InitInstance( HINSTANCE hInstance, int nCmdShow );

// Указатель на имя регистрируемого класса
LPCSTR                  szClassName = "UsgMenuAPI";
// Указатель на заголовок окна приложения
LPCSTR                  szTitle = "Win32API: меню, акселераторы и "
                                  "управление стилем карандаша";

// Параметры карандаша
// Толщина
int                     PenWidth = 1;
// RGB-цвет
COLORREF                PenRGB = RGB( 255, 0, 0 );
// Стиль — сплошной
int                     PenStyle = PS_SOLID;

// Дескриптор главного окна
HWND                    hWnd;

// ****************************************************************
// Главная функция приложения
int WINAPI WinMain(                 // Возвращает TRUE при успехе
    // Дескриптор данного приложения
    HINSTANCE           hInstance,
    // Дескриптор предыдущей запущенной копии приложения.
    //   В Win32API этот параметр всегда равен NULL и оставлен
    //   исключительно для совместимости с версиями ниже четвертой.
    //   Связано это с тем, что каждое 32-разрядное приложение
    //   запускается в собственном адресном пространстве, в
    //   котором, естественно, нет никаких копий или других
    //   приложений
    HINSTANCE           hPrevInstance,
    // Указатель на командную строку, которую можно в
    //   интегрированной среде разработки задать по команде
    //   "Properties" контекстного меню проекта в элементе "Program
    //   arguments" вкладки "Debugging"
    LPSTR               lpCmdLine,
    // Режим начального отображения главного окна приложения —
    //   после вызова параметр получает значение SW_SHOWNORMAL (1),
    //   что соответствует отображению окна в нормальном виде
    int                 nCmdShow )
{
```

```
    // Инициализируем приложение — подготавливаем данные класса
    //   окна и регистрируем его
    if ( !InitApplication( hInstance ) )
        return FALSE;

    // Завершаем создание приложения — создаем и отображаем главное
    //   окно приложения с меню и графикой
    if ( !InitInstance( hInstance, nCmdShow ) )
        return FALSE;

    // Загрузка акселераторов
    HACCEL              hAccelTable;
    hAccelTable = LoadAccelerators( hInstance,
        reinterpret_cast< LPCTSTR >( IDR_MAINFRAME ) );
    MSG                 msg;        // Для очередного сообщения
    // Стандартный цикл обработки сообщений
    while ( GetMessage( &msg, NULL, 0, 0 ) )
    {
        // TranslateAccelerator( ) нужна только при использовании
        //   акселераторов
        if ( !TranslateAccelerator( msg.hwnd, hAccelTable, &msg ) )
        {
            TranslateMessage( &msg );
            DispatchMessage( &msg );
        }
    }

    return static_cast< int >( msg.wParam );
}

// ****************************************************************
// Регистрация класса главного окна
BOOL InitApplication(               // Возвращает TRUE при успешном
                                    //   завершении
    // Дескриптор (уникальное число), ассоциируемый с текущим
    //   приложением
    HINSTANCE           hInstance )
{
    // Сведения о регистрируемом классе
    WNDCLASS            wc;

    // Заполняем структуру класса главного окна
    wc.style         = CS_HREDRAW | CS_VREDRAW;
```

```
    wc.lpfnWndProc   = static_cast< WNDPROC >( WndProc );
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = hInstance;
    wc.hIcon         = LoadIcon( NULL, IDI_ASTERISK );
    wc.hCursor       = LoadCursor( NULL, IDC_CROSS );
    wc.hbrBackground =
        reinterpret_cast< HBRUSH >( COLOR_WINDOW+1 );
    wc.lpszMenuName  = MAKEINTRESOURCE( IDR_MAINFRAME );
    wc.lpszClassName = szClassName;

    // Регистрируем класс главного окна
    return RegisterClass( &wc );
}

// ****************************************************************
// Создание главного окна приложения с графикой и меню
BOOL InitInstance(                  // Возвращает TRUE при успехе
    // Дескриптор текущего приложения
    HINSTANCE           hInstance,
    // Режим отображения главного окна — определяет, в каком виде
    //   будет отображено окно приложения
    int                 nCmdShow )
{
    // Создание главного окна
    hWnd = CreateWindow(
        // Указатель на имя зарегистрированного класса
        szClassName,
        // Указатель на заголовок окна
        szTitle,
        WS_OVERLAPPEDWINDOW,        // Стиль окна
        // Горизонтальная координата левого верхнего угла окна
        //   (используется умалчиваемое значение)
        CW_USEDEFAULT,
        // Вертикальная координата левого верхнего угла окна
        //   (используется умалчиваемое значение)
        CW_USEDEFAULT,
        CW_USEDEFAULT,              // Ширина окна (используется
                                    //   умалчиваемое значение)
        CW_USEDEFAULT,              // Высота окна (используется
                                    //   умалчиваемое значение)
        NULL,                       // Дескриптор родительского
                                    //   окна (его нет)
```

```
        NULL,                       // Дескриптор меню окна
        hInstance,                  // Дескриптор экземпляра
                                    //   приложения
        NULL );                     // Указатель на дополнительные
                                    //   данные окна (их нет)
    if ( !hWnd )
        return FALSE;

    // Показать окно: передает в Windows информацию (nCmdShow) о
    //   том, в каком виде необходимо отобразить окно
    if ( ShowWindow( hWnd, nCmdShow ) )
        return FALSE;

    // Перерисовать окно: для перерисовки окна функция предписывает
    //   ОС Windows послать окну сообщение WM_PAINT
    if ( !UpdateWindow( hWnd ) )
        return FALSE;

    return TRUE;
}

// ****************************************************************
// Оконная процедура главного окна
// Получает очередное сообщение и индивидуально обрабатывает его
LRESULT CALLBACK WndProc(           // Возвращает результат
                                    //   обработки сообщения,
                                    //   зависящий от посланного
                                    //   сообщения: 0 — успех
    HWND            hWnd,           // Дескриптор созданного окна
    UINT            message,        // Номер сообщения
    WPARAM          wParam,         // Дополнительная информация о
    LPARAM          lParam )        //   сообщении, зависящая от
                                    //   типа сообщения (wParam,
                                    //   lParam))
{
    // Дескриптор контекста устройства
    HDC             hDC;
    // Указатель на структуру с информацией для приложения о
    //   клиентской области окна
    PAINTSTRUCT     ps;

    // Обработчик сообщения — в данном случае приложение явно
    //   отвечает только на сообщения WM_DESTROY, WM_PAINT и
```

```
//   WM_COMMAND при выборе команд меню, все остальные сообщения
//   передаются в DefWindowProc( ) — функцию, управляющую
//   поведением окна по умолчанию. Обработка сообщений
//   WM_DESTROY и WM_PAINT подробно рассмотрена в проекте
//   FrameWnd
switch( message )
{
// Вывод в окно
case WM_PAINT:

    // Получение контекста устройства
    hDC = BeginPaint( hWnd, &ps );
    if ( !hDC )
    {
        MessageBox( NULL, "Контекст устройства не получен",
                    "Ошибка 1", MB_OK );
        exit( 1 );
    }
    // Дескриптор нового карандаша
    HPEN            hPen;
    // Создание нового карандаша
    hPen = CreatePen( PenStyle, PenWidth, PenRGB );
    if ( !hPen )
    {
        MessageBox( NULL, "Неверное использование "
                    "CreatePen", "Ошибка 2", MB_OK );
        exit( 2 );
    }
    // Дескриптор старого карандаша
    HPEN            hOldPen;
    // Включение его в контекст
    hOldPen = static_cast< HPEN >( SelectObject( hDC, hPen ) );
    if ( !hOldPen )
    {
        MessageBox( NULL, "Неверное использование "
                    "SelectObject", "Ошибка 3", MB_OK );
        exit( 3 );
    }
    // Рисование двух линий
    if ( !MoveToEx( hDC, 30, 30, NULL ) )
    {
        MessageBox( NULL, "Неверное использование MoveToEx",
                    "Ошибка 4", MB_OK );
```

```
            exit( 4 );
        }
        if ( !LineTo( hDC, 30, 300 ) )
        {
            MessageBox( NULL, "Неверное использование LineTo",
                        "Ошибка 5", MB_OK );
            exit( 5 );
        }
        if ( !LineTo( hDC, 450, 300 ) )
        {
            MessageBox( NULL, "Неверное использование LineTo",
                        "Ошибка 6", MB_OK );
            exit( 6 );
        }
        // Удаление нового карандаша
        if ( !DeleteObject( hPen ) )
        {
            MessageBox( NULL, "Неверное использование "
                        "DeleteObject", "Ошибка 7", MB_OK );
            exit( 7 );
        }
        // Восстанавление старого карандаша
        if ( !SelectObject( hDC, hOldPen ) )
        {
            MessageBox( NULL, "Неверное использование "
                        "SelectObject", "Ошибка 8", MB_OK );
            exit( 8 );
        }
        // Освобождение контекста устройства
        if ( !EndPaint( hWnd, &ps ) )
        {
            MessageBox( NULL, "Контекст устройства не освобожден",
                        "Ошибка 8", MB_OK );
            exit( 8 );
        }
        break;

    // Обработка команд меню
    case WM_COMMAND:
        {
            switch( LOWORD( wParam ) )
            {

            case ID_FILE_EXIT:      // Выбрана команда "Файл-Выход"
```

```
            // Указывает системе, что сделан запрос о
            //   завершении приложения: 0 — код завершения.
            // Вызов этой функции обычно используется в ответ
            //   на поступившее сообщение WM_DESTROY
            PostQuitMessage( 0 );
            // Звуковой сигнал
            MessageBeep( MB_OK );
            break;

        // Выбрана команда "Стиль карандаша-Сплошной"
        case ID_PENSTYLE_SOLID:

            PenStyle = PS_SOLID;
            // Перерисовка клиентской области окна
            InvalidateRect( hWnd, 0, 0 );
            break;

        // Выбрана команда "Стиль карандаша-Пунктир"
        case ID_PENSTYLE_DASH:

            PenStyle = PS_DASH;
            // Перерисовка клиентской области окна
            InvalidateRect( hWnd, 0, 0 );
            break;

        // Выбрана команда "Стиль карандаша-Точечный"
        case ID_PENSTYLE_DOT:

            PenStyle = PS_DOT;
            // Перерисовка клиентской области окна
            InvalidateRect( hWnd, 0, 0 );
            break;

        }
    }
    break;

case WM_DESTROY:

    // Указывает системе, что сделан запрос о завершении
    //   приложения: 0 — код завершения. Вызов этой функции
    //   обычно используется в ответ на поступившее
    //   сообщение WM_DESTROY
```

```
        PostQuitMessage( 0 );
        break;

    default:

        // Обработка сообщения по умолчанию
        return DefWindowProc( hWnd, message, wParam, lParam );

    }

    return 0;
}
```

После запуска приложения на экране появляется главное окно приложения (рис. 6.3). Обратите внимание на наличие строки меню в этом окне. В данном случае строка меню содержит два меню — меню **Файл** и меню **Стиль карандаша**.

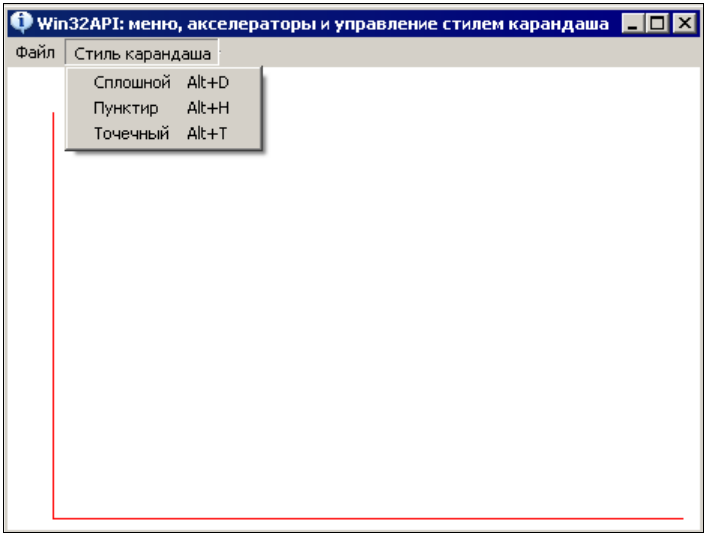


Рис. 6.3. Главное окно приложения UsgMenu

6.2.1. Ресурсы проекта. Работа с редактором ресурсов

Рассмотрение демонстрационной программы начнем с главного — с рассмотрения ресурсов проекта и их программной поддержки. В программе используются два вида ресурсов — акселераторы (ускорители) и меню. Меню

позволяет взаимодействовать с Windows-приложением стандартным способом, с помощью разумно организованного объекта пользовательского интерфейса — строки меню (рис. 6.3). Меню и команды меню являются удобным элементом интерфейса. Общность интерфейса в разных приложениях позволяет быстрее осваивать новые программы.

При выборе команды (пункта) меню (например: меню **Стиль карандаша**, команды **Сплошной**, **Пунктир** или **Точечный**) они посылают Windows соответствующие сообщения. Windows перенаправляет эти сообщения оконной процедуре приложения. В большинстве случаев команды меню формируют сообщения `WM_COMMAND`. Сообщение `WM_COMMAND`>, пришедшее от команды меню, означает, что пользователь либо щелкнул мышью по этому пункту меню, либо с помощью клавиатуры выбрал пункт меню и нажал клавишу <Enter>, либо активизировал связанный с пунктом меню клавиатурный ускоритель. Оконная процедура, получив сообщение `WM_COMMAND`, обрабатывает его желаемым образом. Если же оригинальная обработка сообщения в оконной процедуре не предусмотрена, то выполняется обработка сообщения по умолчанию.

Как же добавить в проект низкоуровневого Windows-приложения требуемые ресурсы?

Прежде всего, при необходимости, следует задать русификацию добавляемых ресурсов. С этой целью во вкладке **Solution Explorer — UsgMenu** правой кнопкой мыши вызовите для проекта UsgMenu контекстное меню, выберите в нем команду **Properties** (Свойства), выберите вкладку **Resourses** (Ресурсы) и в поле **Culture** (Культура) задайте язык **Русский (0x419)**.

Для добавления в проект необходимых ресурсов во вкладке **Solution Explorer — UsgMenu** вызовите для проекта UsgMenu правой кнопкой мыши контекстное меню и выполните в нем команду **Add** (Добавить) | **Resource...** (Ресурс). В появившемся окне **Add Resource** (Добавить ресурс) (см. рис. 6.1) выберите нужные ресурсы (**Accelerator, Menu**), каждый раз, после очередного выбора, нажимая кнопку **New** (Новый).

В появившейся вкладке **Resource View — UsgMenu** выберите ресурсы **Accelerator** и **Menu** и настройте их в соответствии с рис. 6.4—6.5.

Замечание

Вкладки **Resource View — UsgMenu** и **Properties** можно сделать видимыми командами **View** (Вид) | **Resource View** (Просмотр ресурса) и **View** | **Other Windows** (Прочие окна) | **Properties Window** (Окно свойств) соответственно.

Примечание

При необходимости, стандартное расположение окон IDE можно задать командой **Window** (Окно) **| Reset Window Layout** (Восстановить расположение окон).

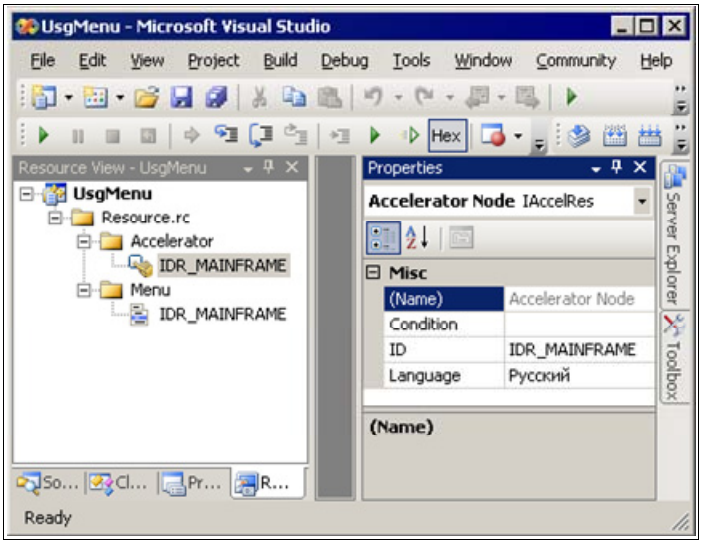


Рис. 6.4. Настройка ресурса Accelerator

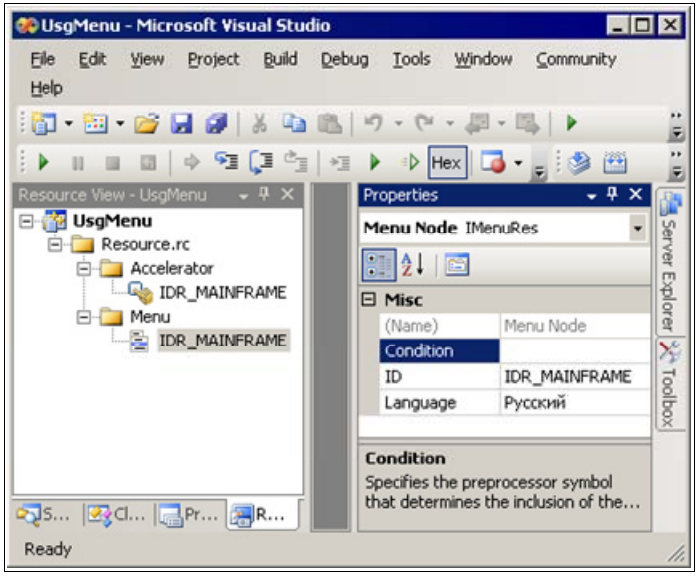


Рис. 6.5. Настройка ресурса Menu

Как же теперь задать состав и настроить меню, команды меню и акселераторы?

Для редактирования меню выберите вкладку **Resource View — UsgMenu**, откройте все ресурсы и щелкните мышью по ресурсу меню, обозначенному

идентификатором IDR_MAINFRAME. В результате этого, меню загрузится в окно редактирования (рис. 6.6) и посредством встроенного редактора его можно модифицировать. С помощью левой кнопки мыши можно выбрать любой пункт меню — он будет отмечен рамкой. Для выбранного пункта меню появится выпадающее подменю, если такое есть. Выбранный пункт меню можно переместить. Создайте копию демонстрационного проекта и попробуйте это проделать. С помощью правой кнопки мыши для любого пункта меню можно вызвать контекстное меню и воспользоваться им. Наконец, дважды щелкнув левой кнопкой мыши по меню, можно посмотреть параметры меню и изменить их. Например, если это проделать для меню **Файл**, то появится окно **Properties**, показанное на рис. 6.7. Оно демонстрирует *типовую настройку* меню, расположенных в строке меню.

Рис. 6.6. Окно редактирования ресурса Menu

Для меню **Файл** (см. рис. 6.7) символ & в свойстве **Caption**, предшествующий букве Ф, делает эту букву в названии меню **Файл** активной (она будет подчеркнута).

Как этим можно воспользоваться, мы покажем далее. Отметим еще несколько важных свойств меню. Свойство **Enabled** должно иметь значение **True**, иначе меню будет выключено. Свойство **Popup** обычно имеет значение **True**. Это означает, что при активизации меню появится всплывающее окно, содержащее команды меню. Если это свойство имеет противоположное значение, то меню настраивается и используется подобно команде (об этом будет сказано далее). Свойство **Help** обычно имеет значение **False**. Это означает, что меню будут располагаться так, как это показано на рис. 6.7. В противном случае соответствующее меню и все меню, находящиеся справа от него, будут прижаты к правой границе.

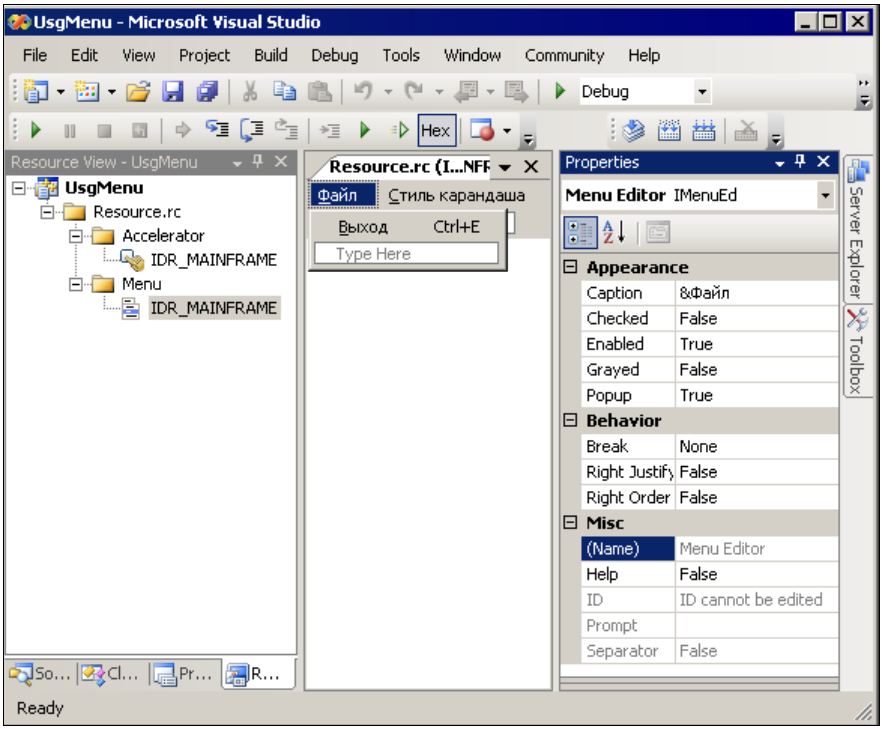


Рис. 6.7. Типовые свойства меню (на примере меню **Файл**)

Совет

В правой части строки заголовка окна **Properties** находятся три кнопки и средняя имеет вид "поплавка" — **Auto Hide** (Скрывать автоматически). При вертикальном расположении "поплавка" окно, при манипуляциях, сохраняется видимым (изменение расположения кнопки происходит при нажатии на нее левой кнопкой мыши). Аналогичные кнопки имеются во многих других окнах IDE. Это очень удобный инструмент — рекомендуем пользоваться им.

Чтобы посмотреть или изменить настройку какой-либо команды меню, например команды **Сплошной** меню **Стиль карандаша**, достаточно выбрать это меню и в появившемся окне дважды щелкнуть левой кнопкой мыши по команде **Сплошной** (рис. 6.8).

Свойства **Caption**, **Enabled, Popup** и **Help** мы уже обсудили. Свойство **ID** задает идентификатор команды, который будет использован при программной обработке действия команды, что будет рассмотрено далее. Свойство **Separator** для пунктов меню имеет значение **False**. В противном случае соответствующая команда будет отображаться в виде разделительной линии. Свойства остальных команд меню аналогичны. Рекомендуем их посмотреть и изучить в процессе упражнений. Для программной поддержки в исходном

коде используются идентификаторы команд. Поэтому перечислим идентификаторы остальных команд. Команда **Выход** использует идентификатор `ID_FILE_EDIT`, команда **Пунктир** — идентификатор `ID_PENSTYLE_DASH`, команда **Точечный** — идентификатор `ID_PENSTYLE_DOT`.

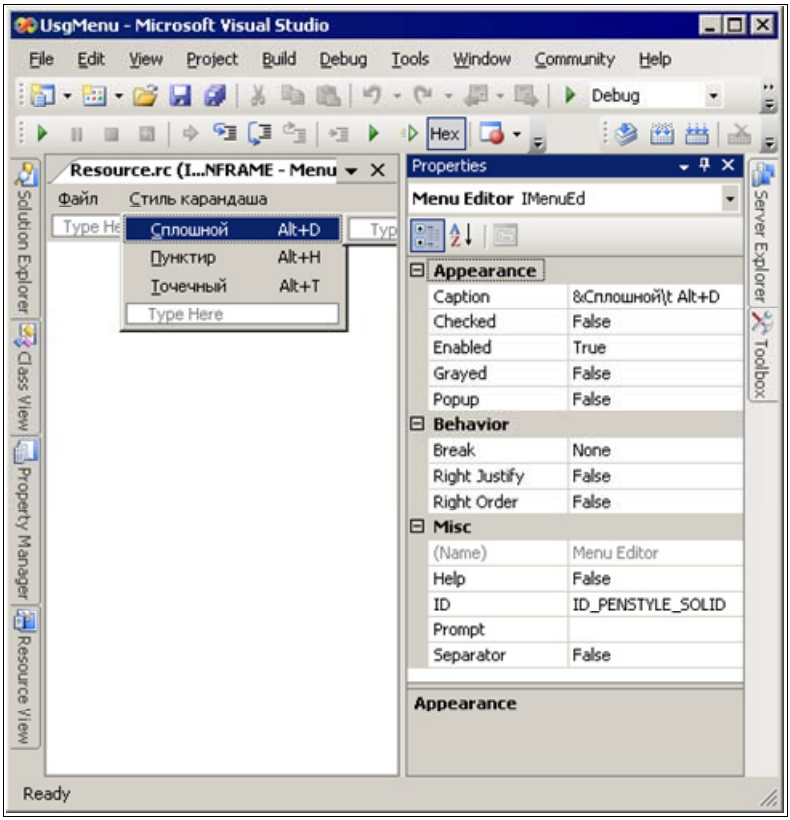


Рис. 6.8. Свойства команды **Сплошной** меню **Стиль карандаша**

Замечание

Обратите внимание на два момента. Во-первых, в конце названия команд меню, для справки, включена информация об акселераторе (об этом пойдет речь далее). Во-вторых, все доступные стандартные и, созданные к этому времени, нестандартные идентификаторы можно посмотреть и выбрать для использования, в качестве значения свойства **ID** (см. рис. 6.8), с помощью кнопки с изображением треугольника, расположенной в правой части. Кнопка обычно не видна, но появляется автоматически при активизации поля значения не только свойства **ID**, но и многих других свойств. Это очень удобно в некоторых случаях.

Для создания нового меню достаточно дважды щелкнуть по пустому меню, расположенному рядом с последним заданным меню. В появившемся окне **Properties** можно задать параметры меню, аналогично тому, как это было описано ранее. Именно так были созданы меню **Файл** и **Стиль карандаша**. Аналогично, для создания новой команды меню достаточно дважды щелкнуть по пустому полю, расположенному внизу. В появившемся окне настройки можно задать параметры команды меню. Так были созданы все команды меню.

Совет

Меню или команду меню можно переместить с помощью левой кнопки мыши.

6.2.2. Акселераторы

Теперь рассмотрим акселераторы программного проекта. С этой целью в окне (вкладке) **Resource View — UsgMenu** щелкнем мышью по ресурсу акселераторов, обозначенному идентификатором `IDR_MAINFRAME`. В результате этого ресурс загрузится в окно редактирования (рис. 6.9), с помощью встроенного редактора ресурсов его можно модифицировать.

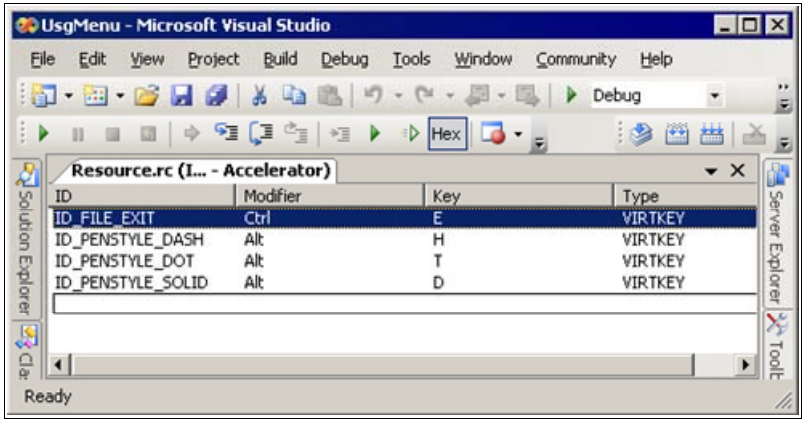


Рис. 6.9. Окно редактирования акселераторов

С помощью левой кнопки мыши можно выбрать любой из акселераторов — он будет подсвечен. Чтобы посмотреть или изменить настройку какого-либо акселератора, достаточно выбрать и дважды щелкнуть по нему левой кнопкой мыши. На рис. 6.10 показана типичная настройка для акселератора. Обратите внимание на то, какой идентификатор использует данный акселератор — это `ID_PENSTYLE_SOLID`, который ранее уже использовался для команды **Стиль карандаша | Сплошной Alt+D**. Это означает, что при вводе клавиа-

турной комбинации <Alt>+<D> будет использована та же обработка, что и при выборе команды **Стиль карандаша | Сплошной Alt+D**. Обратите также внимание, что все доступные стандартные и, созданные к этому времени, нестандартные идентификаторы можно посмотреть и выбрать для использования в поле **ID** с помощью кнопки, расположенной справа (об этом уже говорилось ранее).

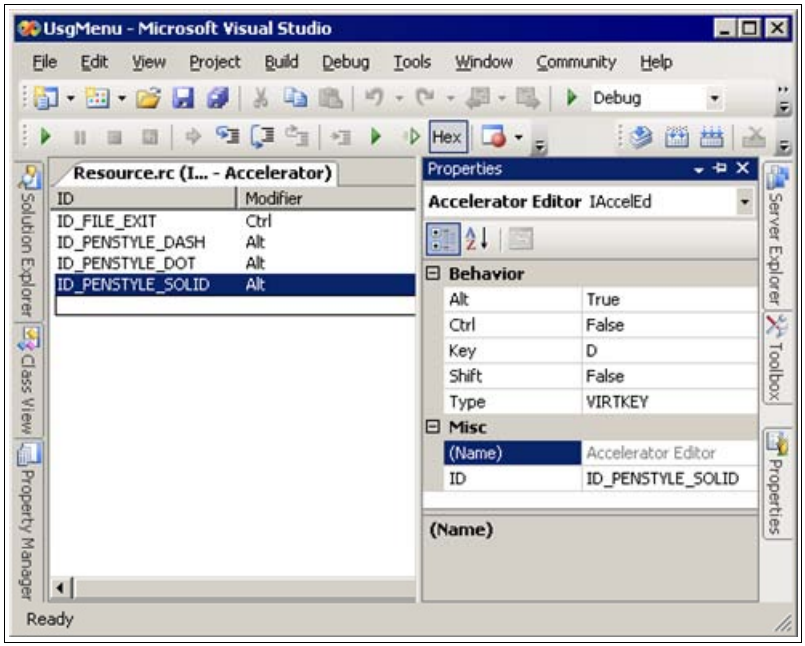


Рис. 6.10. Свойства акселератора

Совет

Задать свойства акселератора можно непосредственно в окне редактирования — в полях строки соответствующего акселератора.

Примечание

Акселераторы могут быть типа VIRTKEY или ASCII. В первом случае можно использовать любые комбинации модификаторов — пустой, <Alt>, <Ctrl>, <Shift>, <Ctrl>+<Alt>, <Ctrl>+<Shift>, <Alt>+<Shift> и <Ctrl>+<Alt>+<Shift>. Во втором случае модификатор может отсутствовать или использоваться единственный модификатор <Alt>.

Для создания нового акселератора достаточно в окне редактирования дважды щелкнуть по пустому нижнему полю. В появившемся окне **Properties** можно задать параметры акселератора, аналогично тому, как это показано на рис. 6.10.

6.2.3. Программная поддержка ресурсов в исходном коде

Теперь настала пора поговорить о программной поддержке ресурсов в исходном коде. Файловый состав рассматриваемого демонстрационного проекта представлен на рис. 6.2. Обращаем ваше внимание на то, что два файла проекта — Resource.rc и resource.h — поддерживаются редактором ресурсов и их непосредственное редактирование не рекомендуется. Они обновляются автоматически при каждой модификации ресурсов проекта, выполняемой с помощью редактора ресурсов.

Модифицируемая на уровне исходного кода часть демонстрационного проекта содержится в файле UsgMenu.cpp (см. листинг 6.1). Рассмотрим те фрагменты файла, которые поддерживают работу с акселераторами и меню.

Прежде всего, для программной поддержки ресурсов акселераторов модифицирован стандартный цикл обработки сообщений в функции `WinMain`:

```
// Загрузка акселераторов
HACCEL                  hAccelTable;
hAccelTable = LoadAccelerators( hInstance,
    reinterpret_cast< LPCTSTR >( IDR_MAINFRAME ) );
MSG                     msg;        // Для очередного сообщения
// Стандартный цикл обработки сообщений
while ( GetMessage( &msg, NULL, 0, 0 ) )
{
    // TranslateAccelerator( ) нужна только при использовании
    //   акселераторов
    if ( !TranslateAccelerator( msg.hwnd, hAccelTable, &msg ) )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
}
```

Непосредственно перед стандартным циклом обработки сообщений вызывается стандартная функция `LoadAccelerators`, которая загружает в приложение таблицу акселераторов, указанную вторым аргументом. Обратите внимание, что в качестве идентификатора ресурса акселераторов используется `IDR_MAINFRAME`, который применялся и редактором ресурсов (см. рис. 6.10). Далее, в теле стандартного цикла обработки сообщений вызывается функция `TranslateAccelerator`, которая формирует сообщение для оконной процедуры.

Для подключения ресурса меню в функции регистрации класса окна `InitApplication` изменена инициализация одного из полей структуры со сведениями о регистрируемом классе:

```
wc.lpszMenuName = MAKEINTRESOURCE( IDR_MAINFRAME );
```

Здесь макрос с параметром `MAKEINTRESOURCE( IDR_MAINFRAME )` используется для корректного приведения типа. Обратите также внимание и на то, что в качестве идентификатора ресурса меню используется `IDR_MAINFRAME`, который применялся и редактором ресурсов.

Наконец, в оконной процедуре `WndProc` выполняется обработка команд меню и акселераторов (см. варианты `ID_FILE_EXIT`, `ID_PENSTYLE_SOLID`, `ID_PENSTYLE_DASH` и `ID_PENSTYLE_DOT` в листинге 6.1). Обратите внимание на то, что в перечисленных вариантах использованы те же идентификаторы, что и в ресурсах команд меню и акселераторов. При обработке команд меню и акселераторов использована стандартная функция `InvalidateRect`, которая обеспечивает перерисовку заданной области окна. Нулевое значение второго аргумента в вызове этой функции задает перерисовку всей клиентской области окна, а нулевое значение последнего аргумента сохраняет прежний фон окна.

6.2.4. Использование "горячих" клавиш при работе с меню

А теперь обещанное — еще об одном способе активизации команд меню без использования мыши и акселераторов. Вспомните, что при настройке меню и команд в их названиях мы использовали символ &, что подчеркивало следующую за ним букву (мы назвали такие символы "горячими", на приведенном рис. 6.3 они подчеркнуты).

Активизировать нужное меню или его команду можно путем соответствующего *последовательного* нажатия "горячих" клавиш. Например, для активизации команды **Точечный Alt+T** меню **Стиль карандаша** достаточно последовательно нажать и отпустить клавиши: <Alt>, <C> и <T>.

6.3. Приложение на базе MFC с акселераторами и меню

Рассматриваемое далее приложение на базе MFC с акселераторами и меню аналогично приведенному в *разд. 6.2*. Соответствующий демонстрационный проект находится на прилагаемом компакт-диске, в папке \Глава 06 \UsgMenu_MFC, а файловый состав проекта показан на рис. 6.11.

Копию проекта UsgMenu_MFC создавайте аналогично проекту WndBtn_MFC из *главы 5*, используя, в качестве основы, его файлы: StdAfx.h, StdAfx.cpp,

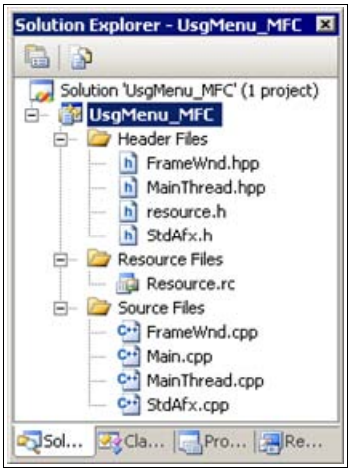


Рис. 6.11. Файловый состав проекта UsgMenu_MFC

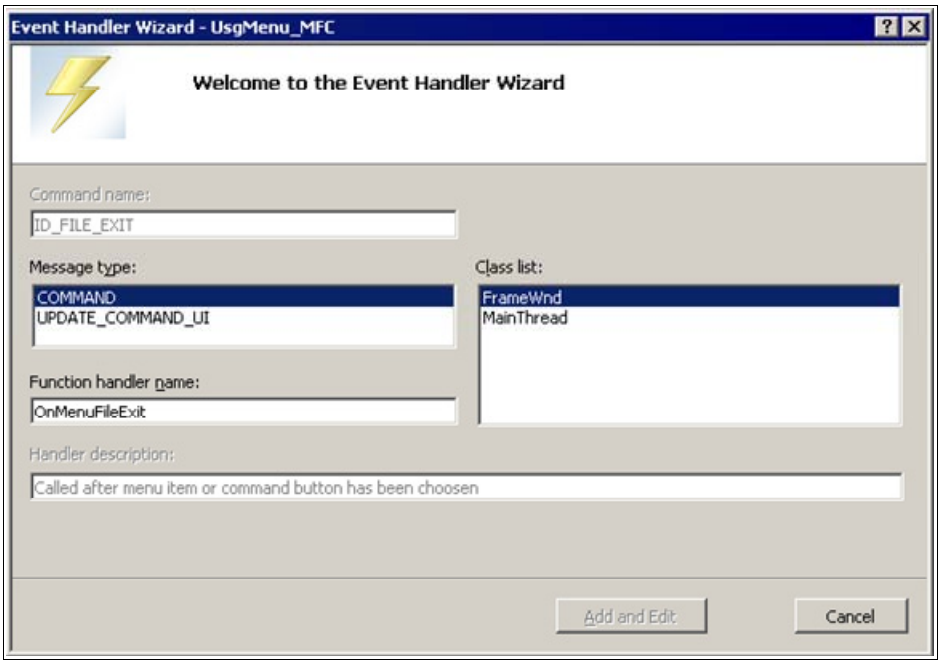


Рис. 6.12. Конфигурация мастера обработки событий для добавления в класс `FrameWnd` таблицы сообщений и обработчика команды **Выход**

Main.cpp, MainThread.hpp (эти файлы не изменяются), а MainThread.cpp, FrameWnd.hpp и FrameWnd.cpp модифицируйте в соответствии с листингами 6.2—6.4. Более подробные сведения о создании проекта см. в *приложении 3*. Для добавления в класс `FrameWnd` таблицы сообщений, прототипов и определений функций, обрабатывающих команды меню, при модификации файлов FrameWnd.hpp и FrameWnd.cpp пользуйтесь мастером обработки событий. Для запуска мастера откройте вкладку **Resource View — UsgMenu**, раскройте ресурс меню и загрузите в окно редактирования файл ресурса Resource.rc. Для команды **Выход** вызовите контекстное меню, выполните
команду **Add Event Handler...** (Добавить обработчик события), добавьте в класс `FrameWnd` таблицу сообщений и обработчик команды **Выход** в соответствии с рис. 6.12. Нажмите кнопку **Add and Edit** (Добавить и отредактировать).

Ресурсы данного проекта и их настройки в точности совпадают с ресурсами предыдущего программного проекта UsgMenu, но дополнены ресурсом таблицы строк, содержащей заголовок главного окна (рис. 6.13).

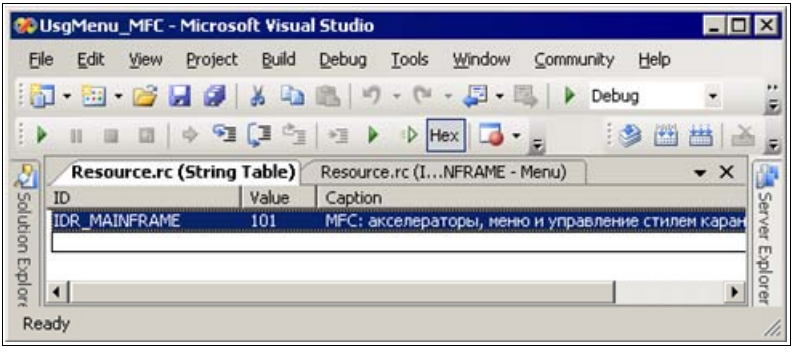


Рис. 6.13. Ресурс таблицы строк проекта UsgMenu_MFC

Листинг 6.2. Файл MainThread.cpp

```
/*
   Файл                 : MainThread.cpp

   Проект               : демонстрация создания Windows-приложения на
                          основе библиотеки классов MFC (ресурсы —
                          меню, акселераторы, строки)

   Назначение           : реализация класса "MainThread",
                          производного от класса "CWinApp" библиотеки
                          классов MFC (инициализация приложения перед
```

```
                        запуском)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAfx.h"                  // Прекомпилируемый файл
#include "MainThread.hpp"            // Объявление класса MainThread
#include "FrameWnd.hpp"              // Объявление класса FrameWnd
// Идентификаторы ресурсов (поддержка редактором ресурсов)
#include "resource.h"

// Переопределение виртуальной функции для инициализации приложения
//   перед запуском
BOOL MainThread :: InitInstance( void )
{
    // Указатель на фрейм окна
    FrameWnd            *pFrame;

    // Размещение фрейма окна в динамической памяти
    pFrame = new FrameWnd;
    ASSERT_VALID( pFrame );          // Обработка ошибки размещения

    // Загрузка ресурсов, регистрация класса окна и создание окна
    BOOL                rc = pFrame->LoadFrame( IDR_MAINFRAME );
    if( !rc )
    {
        TRACE0( "\n Ошибка 1. Метод LoadFrame завершен с "
                "ошибкой \n" );
        exit( 1 );
    }

    // Отображение окна
    pFrame->ShowWindow( SW_SHOW );
    // Обновление окна (генерирует WM_PAINT)
    pFrame->UpdateWindow( );

    // Созданный фрейм делаем главным окном приложения
    this->m_pMainWnd = pFrame;

    return TRUE;
}
```

Листинг 6.3. Файл FrameWnd.hpp

```
/*
   Файл                : FrameWnd.hpp

   Проект              : демонстрация создания Windows-приложения на
                         основе библиотеки классов MFC (ресурсы —
                         меню, акселераторы, строки)

   Назначение          : объявление класса "FrameWnd", производного
                         от класса "CFrameWnd" библиотеки классов
                         MFC (обработка сообщений главного окна)

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _FrameWnd_hpp
    #define _FrameWnd_hpp

    // Объявление класса
    class FrameWnd : public CFrameWnd
    {
        // Данные

    private:

        // Вершины ромба
        CPoint          Vert0;      // Левая вершина
        CPoint          Vert1;      // Верхняя вершина
        CPoint          Vert2;      // Правая вершина
        CPoint          Vert3;      // Нижняя вершина

        // Параметры карандаша
        int             m_PenWidth; // Толщина
        COLORREF        m_PenRGB;   // RGB-цвет
        int             m_PenStyle; // Стиль

        // Методы

    public:

        // Конструктор умолчания (инициализация параметров
        //   карандаша)
        FrameWnd( void );
```

```
    protected:

        // Прототипы функций, обрабатывающих сообщения
        // Перерисовка окна
        afx_msg void OnPaint( void );
        // Обработка команды Файл | Выход
        afx_msg void OnMenuFileExit( void );
        // Обработка команды Стиль карандаша | Сплошной
        afx_msg void OnMenuPenstyleSolid( void );
        // Обработка команды Стиль карандаша | Пунктир
        afx_msg void OnMenuPenstyleDash( void );
        // Обработка команды Стиль карандаша | Точечный
        afx_msg void OnMenuPenstyleDot( void );

    private:

        // Изображение ромба в клиентской области окна
        // Задание вершин ромба
        void SetRhombVertexes( void );
        // Рисование ромба
        void DrawRhomb( CDC *pDC );

        // Объявление таблицы сообщений
        DECLARE_MESSAGE_MAP( );
    };

#endif
```

Листинг 6.4. Файл FrameWnd.cpp

```
/*
   Файл                : FrameWnd.cpp

   Проект              : демонстрация создания Windows-приложения на
                         основе библиотеки классов MFC (ресурсы —
                         меню, акселераторы, строки)

   Назначение          : реализация класса "FrameWnd", производного
                         от класса "CFrameWnd" библиотеки классов
                         MFC (обработка сообщений главного окна)

   Microsoft Visual Studio C++ .NET 2005
*/
```

```
#include "StdAfx.h"                    // Прекомпилируемый файл
#include "FrameWnd.hpp"                // Объявление класса FrameWnd
// Идентификаторы ресурсов (поддержка редактором ресурсов)
#include "resource.h"

// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( FrameWnd, CFrameWnd )
    ON_WM_PAINT( )
    ON_COMMAND( ID_FILE_EXIT, OnMenuFileExit )
    ON_COMMAND( ID_PENSTYLE_SOLID, OnMenuPenstyleSolid )
    ON_COMMAND( ID_PENSTYLE_DASH, OnMenuPenstyleDash )
    ON_COMMAND( ID_PENSTYLE_DOT, OnMenuPenstyleDot )
END_MESSAGE_MAP( )

// Конструктор умолчания (инициализация параметров карандаша)
FrameWnd :: FrameWnd( void )
{
    m_PenWidth = 1;
    m_PenRGB   = RGB( 255, 0, 0 );
    m_PenStyle = PS_SOLID;
}

// Обработка команды Файл | Выход
afx_msg void FrameWnd::OnMenuFileExit( void )
{
    BOOL                rc = MessageBeep( -1 );
    if( !rc )
    {
        TRACE0( "\n Ошибка 2. Неверное завершение MessageBeep \n" );
        exit( 2 );
    }
    rc = DestroyWindow( );
    if( !rc )
    {
        TRACE0( "\n Ошибка 3. Окно не было разрушено \n" );
        exit( 3 );
    }

    return afx_msg void( );
}

// Обработчик сообщения WM_PAINT: рисование ромба
afx_msg void FrameWnd::OnPaint( void )
```

```
{
    // Создаем объект для рисования и вывода
    CPaintDC            PaintDC( this );

    // Рисуем ромб
    SetRhombVertexes( );
    DrawRhomb( &PaintDC );

    return afx_msg void( );
}

// Обработка команды Стиль карандаша | Сплошной
afx_msg void FrameWnd::OnMenuPenstyleSolid( void )
{
    m_PenStyle = PS_SOLID;
    CWnd :: Invalidate( );

    return afx_msg void( );
}

// Обработка команды Стиль карандаша | Пунктир
afx_msg void FrameWnd::OnMenuPenstyleDash( void )
{
    m_PenStyle = PS_DASH;
    CWnd :: Invalidate( );

    return afx_msg void( );
}

// Обработка команды Стиль карандаша | Точечный
afx_msg void FrameWnd::OnMenuPenstyleDot( void )
{
    m_PenStyle = PS_DOT;
    CWnd :: Invalidate( );

    return afx_msg void( );
}

// Задание вершин ромба
void FrameWnd::SetRhombVertexes( void )
{
    CRect               ClientRect; // Клиентская область окна
    // Минимальное значение координаты x в клиентской области окна
    int                 MinX;
```

```
    // Среднее значение координаты x в клиентской области окна
    int                 MidX;
    // Максимальное значение координаты x в клиентской области окна
    int                 MaxX;
    // Минимальное значение координаты y в клиентской области окна
    int                 MinY;
    // Среднее значение координаты y в клиентской области окна
    int                 MidY;
    // Максимальное значение координаты y в клиентской области окна
    int                 MaxY;

    // Получение размеров клиентской области окна
    CWnd :: GetClientRect( &ClientRect );

    MinX = 0;
    MaxX = ClientRect.Width( ) — 1;
    MidX = MaxX / 2;

    MinY = 0;
    MaxY = ClientRect.Height( ) — 1;
    MidY = MaxY / 2;

    // Задание координат вершин
    Vert0 = CPoint( MinX, MidY );
    Vert1 = CPoint( MidX, MinY );
    Vert2 = CPoint( MaxX, MidY );
    Vert3 = CPoint( MidX, MaxY );

    return;
}

// Рисование ромба
void FrameWnd::DrawRhomb(
    CDC                 *pDC )      // Указатель на контекст
                                    //   устройства
{
    CPen                NewPen;     // Новый карандаш
    CPen                *pOldPen;   // Указатель на старый карандаш

    // Создаем новый карандаш
    NewPen.CreatePen( m_PenStyle, m_PenWidth, m_PenRGB );

    // Включаем новый карандаш в контекст устройства и запоминаем
    //   параметры старого карандаша
    pOldPen = pDC->SelectObject( &NewPen );
```

```
    // Позиционируемся в позицию начальной вершины ромба
    pDC->MoveTo( Vert0 );

    // Рисуем стороны ромба
    pDC->LineTo( Vert1 );
    pDC->LineTo( Vert2 );
    pDC->LineTo( Vert3 );
    pDC->LineTo( Vert0 );

    // Восстанавливаем параметры старого карандаша в контексте
    //   устройства
    pDC->SelectObject( pOldPen );

    return;
}
```

Кратко обсудим те фрагменты исходного кода, которые относятся к подключению и обработке ресурсов — остальное было рассмотрено ранее.

Прежде всего, при создании главного окна приложения вместо метода `Create` используется метод `LoadFrame` (см. фрагмент метода `InitInstance` из файла MainThread.cpp, листинг 6.2):

```
// Загрузка ресурсов, регистрация класса окна и создание окна
BOOL                    rc = pFrame->LoadFrame( IDR_MAINFRAME );
if( !rc )
{
    TRACE0( "\n Ошибка 1. Метод LoadFrame завершен с "
            "ошибкой \n" );
    exit( 1 );
}
```

Метод `LoadFrame` регистрирует оконный класс, используя для этого предопределенный шаблон, и извлекает из файла ресурсов, задаваемого своим идентификатором, остальные, необходимые для создания окна Windows, параметры:

```
virtual BOOL CFrameWnd::LoadFrame(
    UINT                nIDResource,
    DWORD               dwDefaultStyle =
        WS_OVERLAPPEDWINDOW | FWS_ADDTOTITLE,
    CWnd                *pParentWnd = NULL,
    CCreateContext      *pContext = NULL );
```

Метод возвращает TRUE, если создание фрейма прошло успешно, и FALSE — в противоположном случае, и имеет следующие параметры:

- nIDResource — идентификатор ресурсов, который определяет меню, таблицу акселераторов и строку, помещаемую в заголовок окна (обратите внимание, что все идентификаторы включаемых ресурсов одинаковы — см. рис. 6.13);
- dwDefaultStyle — стиль фрейма (необязательный параметр), дополнительно к стандартным может быть использован стиль FWS_ADDTOTITLE, который добавит в конец заголовка окна название документа, представленного в окне;
- pParentWnd — указатель на родительское окно данного фрейма (необязательный параметр);
- pContext — указатель на структуру CCreateContext, которая определяет объекты, связанные с данным фреймом (необязательный параметр).

Единственное назначение метода LoadFrame — максимально упростить создание фрейма окна:

- используя идентификатор ресурсов, метод извлекает из ресурсов все необходимые параметры;
- вызывает метод CFrameWnd::Create, который был рассмотрен ранее;
- заполняет структуру CREATESTRUCT, вызывает виртуальный метод CFrameWnd::PreCreateWindow, в котором выполняется регистрация класса окна. Переопределяя PreCreateWindow, можно изменять параметры окна. При переопределении не забывайте вызывать метод базового класса CFrameWnd::PreCreateWindow;
- после того как окно создано и присоединено к объекту фрейма окна, в ответ на сообщение WM_CREATE вызывается виртуальный обработчик сообщения OnCreate. Этот обработчик — одно из мест, где возможно создание и присоединение к окну произвольных элементов пользовательского интерфейса. Для этого достаточно переопределить метод OnCreate в производном классе. Такая ситуация встретится в нашем следующем демонстрационном проекте в *главе 8* с панелью инструментов и строкой статуса. При переопределении не забывайте вызывать метод базового класса CFrameWnd::Create.

Класс FrameWnd, определение которого приведено в файлах FrameWnd.hpp и FrameWnd.cpp, содержит таблицу сообщений и обрабатывающие функции и не требует особых пояснений. Рассмотрим лишь метод Invalidate:

```
void CWnd::Invalidate( BOOL bErase = TRUE );
```

Метод обеспечивает перерисовку клиентской области окна при обработке очередного сообщения `WM_PAINT`. При `bErase = FALSE` фон окна не меняется.

Совет

В заключение рекомендуем поэкспериментировать с данным демонстрационным проектом. Сопоставьте данный проект с предыдущим, аналогичным ему проектом. Это весьма полезно.

6.4. Вопросы и упражнения для самопроверки

Для самопроверки ответьте на следующие вопросы:

1. Что такое ресурсы? Перечислите ресурсы, доступные в IDE.
2. Охарактеризуйте меню, команды, акселераторы и таблицы строк.
3. Как русифицировать добавляемые ресурсы?
4. Как добавить в проект необходимые ресурсы?
5. Как настроить ресурсы?
6. Как сделать видимыми вкладки **Resource View** и **Properties**? Как вернуться к умалчиваемому расположению окон IDE?
7. Как задать состав и настроить меню, команды, акселераторы и "горячие" клавиши?
8. Укажите типовые свойства меню.
9. Как посмотреть или модифицировать свойства команды меню? Укажите типовые свойства команды меню.
10. Как можно создать и настроить новое меню или команду?
11. Как расположить существующие меню или их команды в другом порядке?
12. Как посмотреть или модифицировать свойства акселераторов?
13. Охарактеризуйте типы акселераторов. Укажите, как создать новый акселератор.
14. Какие файлы проекта, использующего ресурсы, поддерживаются редактором ресурсов и непосредственно не редактируются?
15. Как выполняется программная поддержка ресурсов акселераторов?
16. Как в программе подключить ресурс меню?
17. Как в программе обрабатываются команды меню (акселераторы)?

18. Перечислите различные способы активизации команд меню и дайте их краткое описание.
19. Как в приложение на базе MFC добавить с помощью мастера обработки событий таблицу сообщений, прототипы и определения функций, обрабатывающих команды меню и акселераторы?
20. Как к приложению на базе MFC подключить ресурсы?
21. Как в приложении на базе MFC получить размеры клиентской области окна?
22. Какие недостатки имеются в использовании методов классов `CPen` и `CPaintDC` в приложении на базе MFC?

Ответы на вопросы можно проверить — они приведены в *разд. П1.6 приложения 1*.

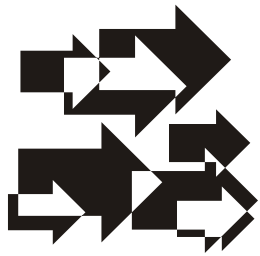
Совет

Одним из указанных в *главе 2* и *приложении 3* способов создайте копии проектов, рассмотренных в этой главе. Не забудьте включить в копии проектов файлы ресурсов — resource.h и Resource.rc.

При экспериментах с созданными копиями проектов выполните следующие упражнения:

1. В одной из копий демонстрационных проектов этой главы поменяйте местами меню.
2. В копии проекта UsgMenu поменяйте местами команды меню **Стиль карандаша**.
3. В копии проекта UsgMenu_MFC поэкспериментируйте со свойствами акселераторов.
4. В копии проекта UsgMenu модифицируйте меню **Файл** таким образом, чтобы приложение завершало работу сразу же после активизации меню.
5. В копии проекта UsgMenu_MFC активизируйте одну из команд меню всеми способами — с помощью мыши, акселератора и "горячих" клавиш.
6. В копии проекта UsgMenu_MFC в меню **Стиль карандаша** вначале удалите команду **Сплошной** и относящийся к ней программный код, а затем с помощью мастера обработки событий добавьте эту же команду.
7. В копии проекта UsgMenu разместите между командами меню **Стиль карандаша** разделители (сепараторы).
8. В копии проекта UsgMenu_MFC прижмите меню к правой границе.

ГЛАВА 7



Ресурсы: панели инструментов и всплывающие подсказки. Строка статуса

Совет

Новым в этой главе является использование редактора ресурсов для создания и настройки *панелей инструментов* с кнопками и *всплывающими подсказками*, а также создание и настройка *строки статуса*. Обратите на них первоочередное внимание.

Любое профессионально написанное оконное приложение (для работы с документами, графикой и т. п.), но не являющееся диалоговым окном, имеет в своем главном окне, как минимум, три дополнительных инструментальных средства, значительно облегчающих работу с приложением — это панели инструментов (Toolbars), строка состояния (Statusbar) и всплывающие подсказки (рис. 7.1).

Панель инструментов — это панель управления с кнопками, которые могут иметь различные стили. С помощью кнопок можно более оперативно, по сравнению с командами меню, выполнять требуемые действия. Например, на рис. 7.1 показана кнопка на панели инструментов, которая, подобно команде **Файл | Выход**, позволяет завершить работу приложения. Панели инструментов обычно выравниваются по верхнему краю окна-рамки. Однако они могут "плавать" как отдельные окна, имеющие свои размеры, но при этом остающиеся на переднем плане или же "пристыковывающиеся" к любому краю окна-рамки. Многочисленные примеры инструментальных панелей можно видеть в IDE. С помощью редактора ресурсов можно легко создать панели инструментов с кнопками и всплывающими подсказками, а подключить их к приложению не представляет никакой сложности.

Всплывающие подсказки — это всплывающие окна, отображающие единственную строку текста (обычно короткую), которая описывает назначение элемента интерфейса пользователя, например, назначение кнопки панели инструментов. Всплывающие подсказки остаются невидимыми до тех пор, пока

пользователь не поместит курсор мыши над каким-либо элементом. И тогда они появляются там, где находится курсор. На рис. 7.1 показана всплывающая подсказка с текстом "Выход", которая появляется, если курсор поместить над кнопкой панели инструментов.

Рис. 7.1. Панель инструментов, всплывающие подсказки и строка статуса рассмотренного далее демонстрационного проекта UsgToolStatusBars

Строка статуса — это полоса (строка), имеющая ряд индикаторов, в которых отображается некоторый текст. Этот текст может содержать пояснение к выбранному пункту меню, выбранной кнопке или данные о текущем состоянии приложения. На рис. 7.1 строка статуса расположена внизу главного окна приложения.

Мы уже неоднократно рассматривали аналогичные примеры демонстрационных программ, построенных как с использованием стандартных функций API, так и с использованием библиотеки классов MFC. Надеемся, что их сравнение позволило вам лучше уяснить анатомию приложений на базе MFC и понять, что создавать такие приложения проще и быстрее. Поэтому в этой главе и далее будем рассматривать демонстрационные программы только приложений на базе MFC.

7.1. Приложение на базе MFC с панелью инструментов, всплывающими подсказками и строкой статуса

Рассматриваемое далее приложение, в части акселераторов, меню и таблицы строк, аналогично рассмотренному в *разд. 6.2*, но дополнительно содержит

панель инструментов со всплывающими подсказками и строку состояния. Соответствующий демонстрационный проект находится на прилагаемом компакт-диске, в папке \Глава 07\,UsgToolStatusBars, а файловый состав проекта показан на рис. 7.2.

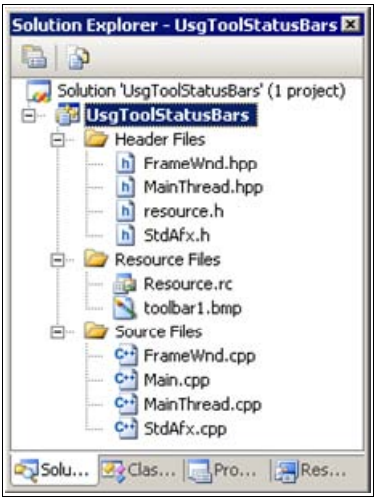


Рис. 7.2. Файловый состав проекта UsgToolStatusBars

Совет

Копию проекта UsgToolStatusBars создавайте самым простым способом — скопируйте папку проекта с компакт-диска. Файлы StdAfx.h, StdAfx.cpp, Main.cpp, MainThread.hpp в данном проекте такие же, как и в проекте UsgMenu_MFC из *главы 6*, а файлы MainThread.cpp, FrameWnd.hpp и FrameWnd.cpp модифицированы в соответствии с листингами 7.1—7.3. Два оставшихся файла — resource.h и Resourse.rc — созданы редактором ресурсов и поддерживаются им. Ресурсы проекта представлены на рис. 7.3, причем ресурсы акселераторов, меню и таблицы строк такие же, как в проекте UsgMenu_MFC из *главы 6*.

Листинг 7.1. Файл MainThread.cpp

```
/*
   Файл                : MainThread.cpp

   Проект              : демонстрация создания Windows-приложения на
                         основе библиотеки классов MFC (панели
                         инструментов, строка статуса)

   Назначение          : реализация класса "MainThread",
                         производного от класса "CWinApp" библиотеки
```

```
                        классов MFC (инициализация приложения перед
                        запуском)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAFX.h"                 // Прекомпилируемый файл
#include "MainThread.hpp"           // Объявление класса MainThread
#include "FrameWnd.hpp"             // Объявление класса FrameWnd
// Идентификаторы ресурсов (поддержка редактором ресурсов)
#include "resource.h"

// Переопределение виртуальной функции для инициализации приложения
//   перед запуском
BOOL MainThread :: InitInstance( void )
{
    // Указатель на фрейм окна
    FrameWnd            *pFrame;

    // Размещение фрейма окна в динамической памяти с обработкой
    //   ошибки
    pFrame = new FrameWnd;
    ASSERT_VALID( pFrame );         // Обработка ошибки размещения

    // Загрузка ресурсов, регистрация класса окна и создание окна
    BOOL                rc = pFrame->LoadFrame( IDR_MAINFRAME );
    if( !rc )
    {
        TRACE0( "\n Ошибка 1. Метод LoadFrame завершен с "
                "ошибкой \n" );
        exit( 1 );
    }

    // Отображение окна
    pFrame->ShowWindow( SW_SHOW );
    // Обновление окна (генерирует WM_PAINT)
    pFrame->UpdateWindow( );

    // Созданный фрейм делаем главным окном приложения
    this->m_pMainWnd = pFrame;

    return TRUE;
}
```

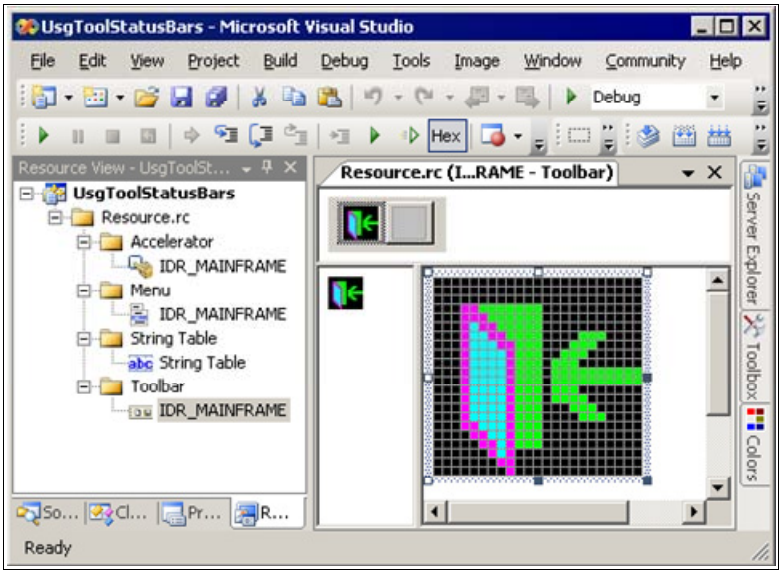


Рис. 7.3. Ресурсы проекта UsgToolStatusBars

Листинг 7.2. Файл FrameWnd.hpp

```
/*
   Файл                 : FrameWnd.hpp

   Проект               : демонстрация создания Windows-приложения на
                          основе библиотеки классов MFC (панели
                          инструментов, строка статуса)

   Назначение           : объявление класса "FrameWnd", производного
                          от класса "CFrameWnd" библиотеки классов
                          MFC (обработка сообщений главного окна)

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _FrameWnd_hpp
    #define _FrameWnd_hpp

    // Объявление класса
    class FrameWnd : public CFrameWnd
```

```
{
    // Данные

private:

    // Вершины ромба
    CPoint          Vert0;      // Левая вершина
    CPoint          Vert1;      // Верхняя вершина
    CPoint          Vert2;      // Правая вершина
    CPoint          Vert3;      // Нижняя вершина

    // Параметры карандаша
    int             m_PenWidth; // Толщина
    COLORREF        m_PenRGB;   // RGB-цвет
    int             m_PenStyle; // Стиль

    // Строка статуса
    CStatusBar      m_wndStatusBar;
    // Панель инструментов
    CToolBar        m_wndToolBar;

    // Методы

public:

    // Конструктор умолчания (инициализация параметров
    //   карандаша)
    FrameWnd( void );

protected:

    // Так нужно записывать прототипы функций, обрабатывающих
    //   сообщения
    // Обработчик сообщения WM_CREATE: создание и визуализация
    //   панелей инструментов и строки статуса
    afx_msg int OnCreate( LPCREATESTRUCT lpCreateStruct );
    // Перерисовка окна
    afx_msg void OnPaint( void );
    // Обработка команды Файл | Выход
    afx_msg void OnMenuFileExit( void );
    // Обработка команды Стиль карандаша | Сплошной
    afx_msg void OnMenuPenstyleSolid( void );
    // Обработка команды Стиль карандаша | Пунктир
    afx_msg void OnMenuPenstyleDash( void );
```

```
        // Обработка команды Стиль карандаша | Точечный
        afx_msg void OnMenuPenstyleDot( void );

    private:

        // Изображение ромба в клиентской области окна
        // Задание вершин ромба
        void SetRhombVertexes( void );
        // Рисование ромба
        void DrawRhomb( CDC *pDC );

        // Объявление таблицы сообщений
        DECLARE_MESSAGE_MAP( );
    };
```

Листинг 7.3. Файл FrameWnd.cpp

```
/*
   Файл                : FrameWnd.cpp

   Проект              : демонстрация создания Windows-приложения на
                         основе библиотеки классов MFC (панели
                         инструментов, строка статуса)

   Назначение          : реализация класса "FrameWnd", производного
                         от класса "CFrameWnd" библиотеки классов
                         MFC (обработка сообщений главного окна)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAFX.h"                  // Прекомпилируемый файл
#include "FrameWnd.hpp"              // Объявление класса FrameWnd
// Идентификаторы ресурсов (поддержка редактором ресурсов)
#include "resource.h"

// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( FrameWnd, CFrameWnd )
    ON_WM_CREATE( )
    ON_WM_PAINT( )
    ON_COMMAND( ID_FILE_EXIT, OnMenuFileExit )
    ON_COMMAND( ID_PENSTYLE_SOLID, OnMenuPenstyleSolid )
```

```
    ON_COMMAND( ID_PENSTYLE_DASH, OnMenuPenstyleDash )
    ON_COMMAND( ID_PENSTYLE_DOT, OnMenuPenstyleDot )
END_MESSAGE_MAP( )

// Индикаторы для строки статуса
static UINT indicators[ ] =
{
    // Размещаем индикаторы в правой части строки статуса
    ID_SEPARATOR,
    ID_INDICATOR_CAPS,              // Индикация Caps Lock
    ID_INDICATOR_NUM,               // Индикация Num Lk
    ID_INDICATOR_SCRL               // Индикация Scr Lk
};

// Конструктор умолчания (инициализация параметров карандаша)
FrameWnd :: FrameWnd( void )
{
    m_PenWidth = 1;
    m_PenRGB   = RGB( 255, 0, 0 );
    m_PenStyle = PS_SOLID;
}

// Обработчик сообщения WM_CREATE: создание и визуализация панелей
//   инструментов и строки статуса
afx_msg int FrameWnd :: OnCreate( LPCREATESTRUCT lpCreateStruct )
{
    // Вначале вызываем метод OnCreate базового класса
    if ( CFrameWnd :: OnCreate( lpCreateStruct ) == -1 )
        return -1;

    // Создаем и визуализируем окно Windows для линейки
    //   инструментов
    if ( !m_wndToolBar.CreateEx( this, TBSTYLE_FLAT, WS_CHILD
         | WS_VISIBLE | CBRS_TOP | CBRS_GRIPPER | CBRS_TOOLTIPS
         | CBRS_FLYBY | CBRS_SIZE_DYNAMIC, CRect( 0, 0, 0, 0 ) ) ||
         !m_wndToolBar.LoadToolBar( IDR_MAINFRAME ) )
    {
        TRACE0( "\n Ошибка 2 создания панели инструментов \n" );
        return 2;
    }

    // Создаем и визуализируем окно Windows для строки статуса
    //   с индикацией состояния Caps Lock, Num Lk и Scr Lk
```

```
    if ( !m_wndStatusBar.Create( this ) ||
         !m_wndStatusBar.SetIndicators( indicators, sizeof( indicators )
         / sizeof( UINT ) ) )
    {
        TRACE0( "\n Ошибка 3 создания строки статуса \n" );
        return 3;
    }

    // Задаем заголовок для будущего окна, если панель инструментов
    //   будет в плавающем режиме
    m_wndToolBar.SetWindowText( "Инструменты" );

    // Разрешаем привязку панели инструментов к любой стороне
    //   фрейма
    m_wndToolBar.EnableDocking( CBRS_ALIGN_ANY );

    // То же самое нужно проделать со стороны фрейма
    EnableDocking( CBRS_ALIGN_ANY );

    // А теперь собственно привязка панели инструментов
    DockControlBar( &m_wndToolBar );

    return afx_msg int( );
}

// Обработчик сообщения WM_PAINT: перерисовывает панель
//   инструментов и изображает симметричный ромб
afx_msg void FrameWnd :: OnPaint( void )
{
    // Создаем объект для рисования и вывода
    CPaintDC            PaintDC( this );

    // !!! Важно — перерисовываем панель инструментов
    m_wndToolBar.LoadToolBar( IDR_MAINFRAME );

    // Рисуем ромб
    SetRhombVertexes( );
    DrawRhomb( &PaintDC );

    return afx_msg void( );
}

// Обработка команды Файл | Выход
afx_msg void FrameWnd :: OnMenuFileExit( void )
```

```
{
    BOOL                rc = MessageBeep( -1 );
    if( !rc )
    {
        TRACE0( "\n Ошибка 4. Неверное завершение MessageBeep \n" );
        exit( 4 );
    }
    rc = DestroyWindow( );
    if( !rc )
    {
        TRACE0( "\n Ошибка 5. Окно не было разрушено \n" );
        exit( 5 );
    }

    return afx_msg void( );
}

// Обработка команды Стиль карандаша | Сплошной
afx_msg void FrameWnd :: OnMenuPenstyleSolid( void )
{
    m_PenStyle = PS_SOLID;
    CWnd :: Invalidate( );

    return afx_msg void( );
}

// Обработка команды Стиль карандаша | Пунктир
afx_msg void FrameWnd :: OnMenuPenstyleDash( void )
{
    m_PenStyle = PS_DASH;
    CWnd :: Invalidate( );

    return afx_msg void( );
}

// Обработка команды Стиль карандаша | Точечный
afx_msg void FrameWnd :: OnMenuPenstyleDot( void )
{
    m_PenStyle = PS_DOT;
    CWnd :: Invalidate( );

    return afx_msg void( );
}
```

```
// Задание вершин ромба
void FrameWnd :: SetRhombVertexes( void )
{
    CRect               ClientRect; // Клиентская область окна
    // Минимальное значение координаты x в клиентской области окна
    int                 MinX;
    // Среднее значение координаты x в клиентской области окна
    int                 MidX;
    // Максимальное значение координаты x в клиентской области окна
    int                 MaxX;
    // Минимальное значение координаты y в клиентской области окна
    int                 MinY;
    // Среднее значение координаты y в клиентской области окна
    int                 MidY;
    // Максимальное значение координаты y в клиентской области окна
    int                 MaxY;

    // Получение размеров клиентской области окна
    CWnd :: GetClientRect( &ClientRect );

    MinX = 0;
    MaxX = ClientRect.Width( ) - 1;
    MidX = MaxX / 2;

    MinY = 0;
    MaxY = ClientRect.Height( ) - 1;
    MidY = MaxY / 2;

    // Задание координат вершин
    Vert0 = CPoint( MinX, MidY );
    Vert1 = CPoint( MidX, MinY );
    Vert2 = CPoint( MaxX, MidY );
    Vert3 = CPoint( MidX, MaxY );

    return;
}

// Рисование ромба
void FrameWnd :: DrawRhomb(
    CDC                 *pDC )      // Указатель на контекст
                                    //   устройства
{
    CPen                NewPen;     // Новый карандаш
    CPen                *pOldPen;   // Указатель на старый карандаш
```

```
    // Создаем новый карандаш
    NewPen.CreatePen( m_PenStyle, m_PenWidth, m_PenRGB );

    // Включаем новый карандаш в контекст устройства и запоминаем
    //   параметры старого карандаша
    pOldPen = pDC->SelectObject( &NewPen );

    // Позиционируемся в позицию начальной вершины ромба
    pDC->MoveTo( Vert0 );

    // Рисуем стороны ромба
    pDC->LineTo( Vert1 );
    pDC->LineTo( Vert2 );
    pDC->LineTo( Vert3 );
    pDC->LineTo( Vert0 );

    // Восстанавливаем параметры старого карандаша в контексте
    //   устройства
    pDC->SelectObject( pOldPen );

    return;
}
```

7.2. Панель инструментов

Для включения панели инструментов в проект проще всего выбрать вкладку **Resource View** и выполнить команду меню **Project | Add Resource...**. В появившемся окне (см. рис. 6.1) выберите тип ресурса **Toolbar** и нажмите кнопку **New**. В результате этого в проект будет включен ресурс панели инструментов.

Для настройки панели инструментов надо выбрать идентификатор панели инструментов, открыть вкладку **Properties** и задать параметры панели инструментов в соответствии с рис. 7.4. Обратите внимание на то, что идентификатор ресурса панели инструментов не обязательно должен совпадать с идентификаторами других ресурсов программного проекта (см. рис. 7.3). Объясняется это тем, что для загрузки ресурса панели инструментов используется отдельный метод `LoadToolBar` (см. листинг 7.3).

Для включения кнопки в созданную панель инструментов достаточно дважды "щелкнуть" левой кнопкой мыши по ресурсу панели `IDR_MAINFRAME` и в окне редактирования появится ресурс панели инструментов. Далее следует дважды "щелкнуть" левой кнопкой мыши по пустой кнопке. В результате появит-

ся окно настройки параметров кнопки (рис. 7.5). После задания параметров кнопки можно "нарисовать" пиктограмму кнопки и ресурс панели инструментов приобретет вид, показанный на рис. 7.3.

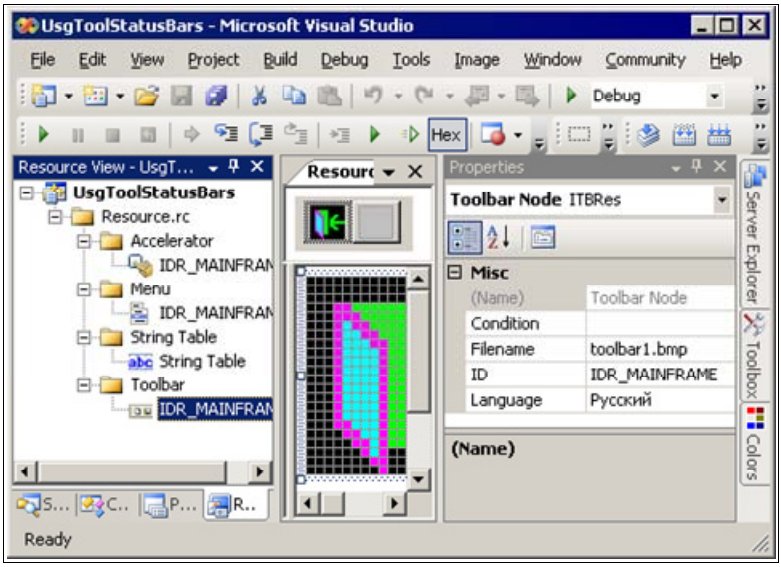


Рис. 7.4. Задание свойств панели инструментов

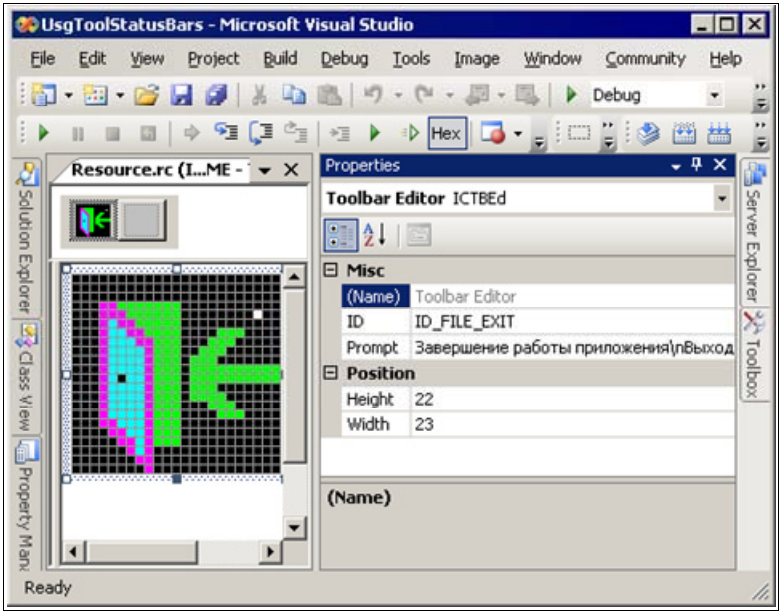


Рис. 7.5. Задание свойств кнопки на панели инструментов

На рис. 7.5 следует обратить внимание на два момента. Во-первых, идентификатор кнопки совпадает с идентификатором команды **Файл | Выход**. Поэтому выполнение этой команды и нажатие кнопки на панели инструментов приведут к одинаковому результату — работа приложения будет завершена. Во-вторых, в поле **Prompt** задается текст, выдаваемый в строку статуса (он предшествует символу \n), и текст всплывающей подсказки (он следует за символом \n).

Примечание

С помощью редактора ресурсов можно легко изменить взаимное расположение *кнопок* панели инструментов. Для этого достаточно, левой кнопкой мыши, "перетащить" соответствующие кнопки на нужное место. Группы взаимосвязанных кнопок панели инструментов можно разделить сепараторами (разделяющими линиями). Для достижения этого нужно разделяемые кнопки, с помощью левой кнопки мыши, сначала надвинуть друг на друга, а потом раздвинуть. И, наконец, существующую кнопку можно удалить с панели инструментов просто переместив ее за пределы панели инструментов.

Теперь рассмотрим *программную поддержку панели инструментов*. Добавить панель инструментов в приложение на базе MFC довольно легко. Для этого достаточно определить, в качестве члена класса, переменную с типом `CToolBar` в классе главного окна-рамки приложения (см. файл FrameWnd.hpp, листинг 7.2):

```
// Панель инструментов
CToolBar                m_wndToolBar;
```

И создать панель инструментов в методе обработки сообщения `WM_CREATE` *(см. разд. 6.3)* окна-рамки с помощью метода `CToolBar :: Create( )` или `CToolBar::CreateEx`:

```
// Создаем и визуализируем окно Windows для линейки
//   инструментов
if ( !m_wndToolBar.CreateEx( this, TBSTYLE_FLAT, WS_CHILD
     | WS_VISIBLE | CBRS_TOP | CBRS_GRIPPER | CBRS_TOOLTIPS
     | CBRS_FLYBY | CBRS_SIZE_DYNAMIC, CRect( 0, 0, 0, 0 ) ) ||
     !m_wndToolBar.LoadToolBar( IDR_MAINFRAME ) )
{
    TRACE0( "\n Ошибка 2 создания панели инструментов \n" );
    return 2;
}
```

Стиль `TBSTYLE_FLAT` в списке аргументов метода `CreateEx` создает панель инструментов и кнопки так, чтобы текст кнопки, если он будет, был виден под точечным рисунком кнопки. Стиль `CBRS_TOP` задает начальное положение инструментальной панели на верхней стороне рамки. Стиль `CBRS_GRIPPER` обес-

печивает перемещаемость панели инструментов. Для добавления всплывающих подсказок и получения возможности изменения размеров панели, когда она находится в плавающем состоянии, используется сочетание стилей `CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC`. В этом фрагменте исходного кода используется метод `CToolBar::LoadToolBar`, которому в качестве аргумента передается идентификатор ресурса `IDR_MAINFRAME`. Этот идентификатор определяет растровое изображение, используемое для отображения кнопок этой панели (в данном проекте используется единственная кнопка).

Метод `CToolBar::CreateEx` имеет следующий прототип:

```
// Возвращает ненулевое значение при успехе, иначе — 0
BOOL CToolBar :: CreateEx(
    // Указатель на родительское окно панели инструментов
    CWnd                *pParentWnd,
    // Дополнительные стили кнопок панели инструментов
    DWORD               dwCtrlStyle = TBSTYLE_FLAT,
    // Стиль панели инструментов
    DWORD               dwStyle = WS_CHILD | WS_VISIBLE |
                                  CBRS_ALIGN_TOP,
    // Задает прямоугольник, определяющий положение и размер
    //   панели инструментов (по умолчанию положение и размер
    //   определяются автоматически)
    CRect               rcBorders = CRect(0, 0, 0, 0),
    // Идентификатор дочернего окна (панель инструментов
    //   является дочерним окном)
    UINT                nID = AFX_IDW_TOOLBAR);
```

Можно использовать следующие дополнительные стили кнопок панелей инструментов:

- `TBSTYLE_BUTTON` — стандартная кнопка;
- `TBSTYLE_SEP` — разделитель между группами кнопок, представляющий собой маленький промежуток между группами кнопок. Кнопка с таким стилем не может взаимодействовать с пользователем;
- `TBSTYLE_CHECK` — кнопка с фиксацией. Эта кнопка "залипает", когда пользователь на нее нажимает. Для возврата в исходное положение надо нажать на нее еще раз;
- `TBSTYLE_GROUP` — отмечает начало группы стандартных кнопок;
- `TBSTYLE_CHECKGROUP` — отмечает начало группы кнопок с фиксацией. Кнопка остается нажатой до тех пор, пока не будет нажата другая кнопка из этой же группы;

- TBSTYLE_TOOLTIPS — панель инструментов может создавать и управлять элементом "всплывающая" подсказка;
- TBSTYLE_WRAPABLE — панель инструментов может располагаться в несколько линий. Кнопки переносятся на следующую строку, если им не хватило места в этой строке.

Для панели инструментов можно комбинировать следующие стили:

- CBRS_TOP — расположить вверху рабочей области родительского окна;
- CBRS_BOTTOM — расположить внизу рабочей области родительского окна;
- CBRS_LEFT — расположить в левой части рабочей области родительского окна;
- CBRS_RIGHT — расположить в правой части рабочей области родительского окна;
- CBRS_TOOLTIPS — выводить краткое описание кнопок (всплывающие подсказки);
- CBRS_SIZE_DYNAMIC — разрешает изменять размеры панели инструментов при помощи мыши, когда панель находится в плавающем состоянии;
- CBRS_SIZE_FIXED — запрещает изменять размеры панели инструментов;
- CBRS_FLOATING — панель инструментов плавающая;
- CBRS_FLYBY — в строке состояния отображать информацию о кнопке;
- CBRS_HIDE_INPLACE — сделать невидимой для пользователя.

Можно задавать также стили окон Windows и стили, общие для всех панелей управления:

- CBRS_ALIGN_TOP — эквивалентно CBRS_TOP;
- CBRS_ALIGN_BOTTOM — эквивалентно CBRS_BOTTOM;
- CBRS_ALIGN_LEFT — эквивалентно CBRS_LEFT;
- CBRS_ALIGN_RIGHT — эквивалентно CBRS_RIGHT;
- CBRS_ALIGN_ANY — панель управления привязывается к какой-либо стороне рабочей области фрейма;
- CBRS_BORDER_TOP — рамка очерчивает верхний край панели управления, когда она видима;
- CBRS_BORDER_BOTTOM — рамка очерчивает нижний край панели управления, когда она видима;
- CBRS_BORDER_LEFT — рамка очерчивает левый край панели управления, когда она видима;

- `CBRS_BORDER_RIGHT` — рамка очерчивает правый край панели управления, когда она видима;
- `CBRS_FLOAT_MULTI` — в единственном окне фрейма могут располагаться несколько панелей управления;
- `CCS_ADJUSTABLE` — пользователь может изменять конфигурацию элемента управления;
- `CCS_TOP` — элемент управления должен быть расположен в верхней части рабочей области окна;
- `CCS_NODIVIDER` — в верхней части элемента управления не надо рисовать разделительную линию шириной 2 пиксела;
- `CCS_NOMOVEY` — в ответ на сообщение `WM_SIZE` элемент управления может изменять свои размеры и перемещаться только горизонтально, а вертикальное положение меняться не будет;
- `CCS_NOPARENTALIGN` — элемент управления не будет автоматически перемещаться в верхнюю или нижнюю часть родительского окна;
- `CCS_NORESIZE` — приложение должно задавать размеры элемента управления явным образом, т. к. размеры, заданные по умолчанию, не будут использоваться.

Чтобы панель инструментов можно было привязать к любой стороне фрейма главного окна, необходимо вызвать методы `CToolBar::EnableDocking` и `CFrameWnd::EnableDocking` следующим образом:

```
// Разрешаем привязку панели инструментов к любой стороне
//   фрейма
m_wndToolBar.EnableDocking( CBRS_ALIGN_ANY );
// То же самое нужно проделать со стороны фрейма
EnableDocking( CBRS_ALIGN_ANY );
```

И, наконец, можно пристыковать панель инструментов к окну-рамке при запуске приложения с помощью вызова `DockControlBar` следующим образом:

```
// А теперь собственно привязка панели инструментов
DockControlBar( &m_wndToolBar );
```

7.3. Строка состояния

Использовать объект с типом `CStatusBar` также очень просто, учитывая, что библиотека классов MFC выполняет за вас всю черновую работу. Для этого необходимо определить, в качестве члена класса, переменную типа `CStatusBar` в классе главного окна-рамки приложения (см. листинг 6.2):

```
// Строка статуса
CStatusBar              m_wndStatusBar;
```

Затем необходимо создать массив строк или идентификаторов строковых ресурсов, которые будут использованы для отображения текста в индикаторах строки состояния. Обычно используются предопределенные значения:

```
static UINT indicators[ ] =
{
    ID_SEPARATOR,
    ID_INDICATOR_CAPS,              // Индикация режима Caps Lock
    ID_INDICATOR_NUM,               // Индикация режима Num Lock
    ID_INDICATOR_SCRL               // Индикация режима Scroll Lock
};
```

Наряду с этими можно использовать следующие нерекомендуемые значения:

- `ID_INDICATOR_EXT` — расширенный индикатор выбора;
- `ID_INDICATOR_OVR` — индикатор режима вставки;
- `ID_INDICATOR_REC` — индикатор режима записи.

Для создания строки состояния, в методе обработки сообщения `WM_CREATE` окна-рамки, вызывается метод `CStatusBar::Create`, а для задания текста индикаторов строки состояния вызывается метод `CStatusBar::SetIndicators`:

```
// Создаем и визуализируем окно Windows для строки статуса
if ( !m_wndStatusBar.Create( this ) ||
     !m_wndStatusBar.SetIndicators( indicators,
     sizeof( indicators ) / sizeof( UINT ) ) )
{
    TRACE0( "\n Ошибка 3 создания строки статуса \n" );
    return 3;
}
```

Метод `CStatusBar::Create` имеет следующий прототип:

```
// Возвращает ненулевое значение при успехе, иначе — 0
BOOL Create(
    // Указатель на родительское окно строки статуса (this -
    //   текущее окно родительсеое)
    CWnd                *pParentWnd,
    // Стиль строки состояния. В дополнение к стандартным стилям
    //   Windows поддерживаются следующие стили:
    //   CBRS_TOP — строка состояния вверху фрейма окна;
    //   CBRS_BOTTOM — строка состояния внизу фрейма окна;
```

```
    //   CBRS_NOALIGN — строка состояния не переразмещается при
    //                  изменении размеров родительского окна;
    //   CBARS_SIZEGRIP — правый нижний угол строки состояния
    //                    приобретает вид, показанный на рис. 7.1
    DWORD               dwStyle = WS_CHILD | WS_VISIBLE |
                                  CBRS_BOTTOM,
    UINT                nID = AFX_IDW_STATUS_BAR );
```

В демонстрационном проекте для второго и третьего аргументов, в вызове метода `CStatusBar::Create`, использованы значения по умолчанию. По умолчанию строка состояния располагается в нижней части родительского окна. Однако, при желании, его можно расположить и в верхней части, использовав стиль `CBRS_TOP`.

Метод `CStatusBar::SetIndicators` имеет следующий прототип:

```
// Возвращает ненулевое значение при успехе, иначе — 0
BOOL SetIndicators(
    // Указатель на массив индикаторов
    const UINT          *lpIDArray,
    // Размер массива
    int                 nIDCount );
```

Вид главного окна демонстрационного проекта UsgToolStatusBars показан на рис. 7.1.

7.4. Вопросы и упражнения для самопроверки

Для самопроверки ответьте на следующие вопросы:

1. Что такое панель инструментов?
2. Что такое строка статуса?
3. Что такое всплывающие подсказки?
4. Как создать панель инструментов?
5. Как настроить панель инструментов?
6. Как включить кнопку в созданную панель инструментов и настроить ее?
7. Как для кнопки панели инструментов задать текст, выдаваемый в строку статуса, и текст всплывающей подсказки?
8. Как можно изменить взаимное расположение кнопок панели инструментов, разделить их сепараторами или удалить кнопки?

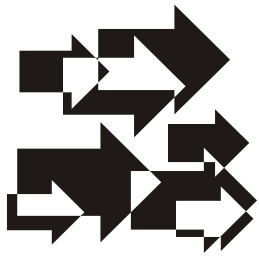
9. В чем заключается программная поддержка панели инструментов?
10. Как создать и настроить строку статуса?

Ответы на вопросы можно проверить — они приведены в *разд. П1.7 приложения 1*.

При экспериментах с созданной копией проекта выполните следующие упражнения:

1. Добавьте в панель инструментов дополнительные кнопки, попробуйте поменять их взаимное расположение, разделять сепараторами и удалять кнопки.
2. Запретите изменение положения панели инструментов.
3. Добавьте еще одну панель инструментов.
4. Измените расположение строки статуса.
5. Измените текст всплывающей подсказки и текст, выдаваемый в строку состояния.

ГЛАВА 8



Ресурсы: диалоговые окна и управляющие элементы

Совет

Новым в этой главе является использование редактора ресурсов для создания и настройки широко применяемых в приложениях диалоговых окон с управляющими элементами. Уделите им должное внимание.

Весьма важной частью Windows-приложений являются диалоговые окна (dialog boxes), обеспечивающие интерактивное взаимодействие с приложением. Как правило, они включают *дочерние* (child) окна, которые называют элементами управления. С их помощью можно задать, получить или отобразить значения соответствующих параметров (переменных) приложения.

Диалоговые окна бывают двух типов — *модальные* (modal) и *немодальные* (modeless). Разница между ними заключается в способе управления потоками сообщений. Модальные диалоговые окна блокируют все остальные окна приложения так, что пользователь не может ничего сделать, пока не закроет модальное окно. Иначе говоря, модальное окно отсекает поток сообщений, идущих от мыши и клавиатуры к родительским или *одноуровневым* (sibling) окнам, делая их недоступными. Если пользователь все же пытается взаимодействовать с недоступным окном, то система предупреждает об этом звуковым сигналом. Однако модальные диалоговые окна обычно не препятствуют переключению с одного приложения на другое.

Немодальные диалоговые окна больше похожи на обычные окна, т. к. они дают возможность доступа к остальным окнам приложения. Это связано с тем, что диалоговые окна этого типа не прерывают потоков сообщений, идущих в любую часть вашего приложения. На первый взгляд немодальные диалоговые окна более привлекательны, т. к. дают пользователю большую свободу выбора. Однако большинство диалоговых окон все-таки модальные. Это связано с тем, что модальное диалоговое окно идеально концентрирует вни-

мание пользователя на некотором ограниченном наборе действий, которые он может в данный момент выполнить.

Оба типа диалоговых окон являются важнейшей частью пользовательского интерфейса ОС Windows и Windows-приложений. Поэтому, наряду с *пользовательскими диалоговыми окнами*, создаваемыми для нужд приложения, уже давно сложился стандартный набор широко используемых диалоговых окон (табл. 8.1). Пояснять, что собой представляют стандартные диалоговые окна и как ими пользоваться, видимо, нет необходимости. Скорее всего, вы часто сталкивались и пользовались ими.

Таблица 8.1. *Стандартные диалоговые окна Windows*

Назначение диалогового окна	**Описание**
Выбор цвета	Позволяет выбирать и создавать цвета. Возвращает выбранный или созданный цвет (это окно реализуется с помощью класса `CColorDialog` библиотеки классов MFC)
Выбор шрифта	Позволяет выбирать шрифт и возвращает выбранный шрифт (класс `CFontDialog` библиотеки классов MFC)
Поиск текста	Позволяет указывать строку, которую нужно найти, и возвращает выбранную строку (класс `CFindReplaceDialog` библиотеки классов MFC)
Поиск и замена текста	Позволяет указывать строки, используемые в операции поиска и замены, и возвращает указанные строки (класс `CFindReplaceDialog` библиотеки классов MFC)
Открыть	Позволяет указать полное имя файла, который нужно открыть, и возвращает указанное имя (класс `CFileDialog` библиотеки классов MFC)
Сохранить как	Позволяет указать полное имя сохраняемого файла и возвращает это имя (класс `CFileDialog` библиотеки классов MFC)
Печать	Позволяет установить параметры задания на печать, включая качество печати, диапазон страниц и число копий, и возвращает установленные параметры (класс `CPrintDialog` библиотеки классов MFC)
Параметры страницы	Позволяет установить параметры страницы, используемые при печати, включая ориентацию страницы, размеры полей, размеры страницы и пр., и возвращает установленные параметры страницы (класс `CPageSetupDialog` библиотеки классов MFC)

На рис. 8.1 показано положение классов, перечисленных в табл. 8.1, в иерархии классов библиотеки MFC.

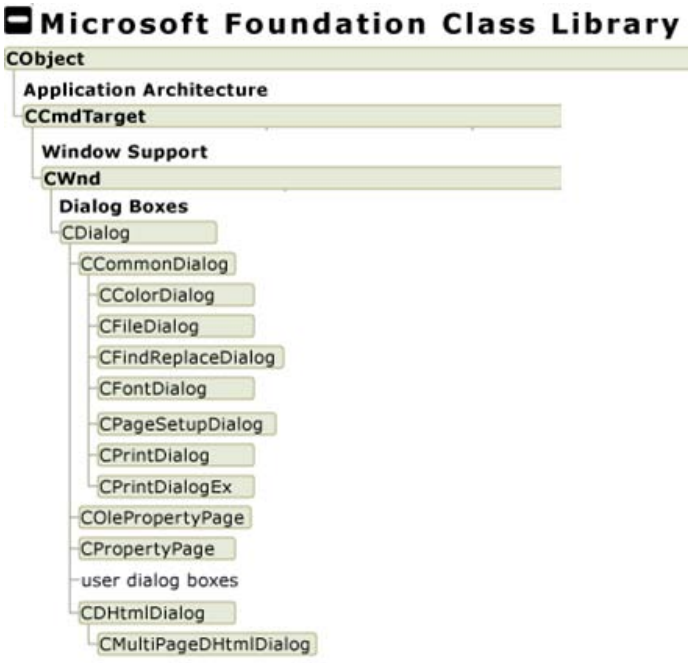


Рис. 8.1. Положение стандартных классов диалоговых окон в иерархии классов библиотеки MFC

Примечание

Напомним, что самые точные и полные сведения о составе и иерархии классов библиотеки MFC можно получить из справочной системы IDE Microsoft Visual Studio C++ .NET 2005. Для этого в IDE достаточно выполнить команду **Help | Index** и в поле **Look for:** появившегося окна **MFC Reference — Microsoft Visual Studio 2005 Documentation — Microsoft Document Explorer** набрать **MFC, hierarhy** и посмотреть вкладки **Hierarhy Chart** и **Hierarhy Chart Categories**.

Стандартные диалоговые окна можно встроить в каркас полученного с помощью мастера MFC Application Wizard Windows-приложения, что и будет рассмотрено далее, в *главе 12*. Поэтому давайте займемся пользовательскими диалоговыми окнами — теми, которые можно создавать самим. Используя библиотеку классов MFC, вы получите значительное преимущество, в части затрат времени и труда, перед теми, кто пользуется функциями API. Библиотека MFC абстрагирует понятия и функциональные возможности диалоговых окон в классе `CDialog` (см. рис. 8.1, user dialog boxes).

8.1. Пользовательские диалоговые окна

Как уже говорилось ранее, диалоговые окна — это важный компонент графического интерфейса пользователя в большинстве Windows-приложений. Они предоставляют возможность взаимодействовать с приложением, а диалоговые окна библиотеки MFC, кроме того, позволяют хранить и проверять данные, вводимые пользователем. Далее мы рассмотрим, как создаются пользовательские диалоговые окна, какие элементы управления они могут использовать и как классы, производные от стандартного класса `CDialog` библиотеки классов MFC, могут абстрагировать шаблоны ресурсов диалоговых окон, чтобы применять в них все возможности библиотеки MFC.

Что отличает диалоговые окна от других окон? Прежде всего, то, что их можно проектировать визуально, с помощью редактора ресурсов. И это является очевидным преимуществом, т. к. нет необходимости задавать положение элементов управления в коде, можно просто визуально разместить их на шаблоне ресурсов диалогового окна в редакторе ресурсов. Визуальный редактор ресурсов запоминает тип, стили, положение, размеры и другие параметры диалогового окна и его управляющих элементов в шаблоне ресурсов диалогового окна. Шаблон ресурсов диалогового окна определяет само окно и хранится в файле ресурсов, подобно меню, акселераторам, таблицам строк, панелям инструментов и другим ресурсам.

Диалоговое окно представляет собой окно особого рода — оно используется для того, чтобы принять ввод, сделанный пользователем в той или иной форме. Диалоговое окно, подобно любому другому, реагирует на сообщения, но, в дополнение к множеству обычных сообщений, диалоговое окно обрабатывает еще два дополнительных сообщения (`WM_INITDIALOG` и `WM_COMMAND`).

Диалоговое окно получает сообщение `WM_INITDIALOG` после того, как оно было создано и все его элементы управления инициализированы, но до вывода его на экран. Поэтому если вы создадите свой перегруженный метод обработки сообщения `WM_INITDIALOG`, то сможете модифицировать элементы управления или данные перед тем, как диалоговое окно появится на экране.

Библиотека классов MFC содержит класс `CDialog` — базовый класс диалогового окна (см. рис. 8.1). Этот класс абстрагирует шаблон ресурсов диалогового окна и включает в себя различные методы, предназначенные для работы с диалоговыми окнами, с которыми вам необходимо ознакомиться.

Для создания диалогового окна следует создать объект с типом класса `CDialog` (или с типом класса, производного от `CDialog`). Как только такой объект создан, появляются две возможности: воспользоваться методом `DoModal`, чтобы создать модальное диалоговое окно, или же вызвать метод `Create`, чтобы создать немодальное диалоговое окно. В обоих случаях на этом этапе создается диалоговое окно со своими элементами управления и оно связывается

с созданным объектом MFC. Метод `Create` имеет два прототипа, из которых чаще всего используется следующий:

```
virtual BOOL Create(
    // Цифровой идентификатор ресурса диалогового окна
    UINT                nIDTemplate,
    // Указатель на окно-хозяин диалогового окна
    CWnd                *pParentWnd = NULL );
Второй прототип имеет следующий вид:
virtual BOOL Create(
    // Строка, заканчивающаяся нулевым символом, которая является
    //   названием ресурса диалогового окна
    LPCTSTR               lpszTemplateName,
    // Указатель на окно-хозяин диалогового окна
    CWnd                *pParentWnd = NULL );
```

Класс `CDialog` включает в себя ряд стандартных (табл. 8.2) и виртуальных методов (табл. 8.3).

Таблица 8.2. *Стандартные методы класса* `CDialog`

Метод	Описание
`DoModal`	Отображает модальное диалоговое окно и не завершается до тех пор, пока диалог не будет закрыт
`EndDialog`	Закрывает модальное диалоговое окно
`GetDefID`	Извлекает идентификатор кнопки по умолчанию, если он есть
`GotoDlgCtrl`	Перемещает фокус ввода на указанный элемент диалогового окна
`MapDialogRect`	Преобразует единицы измерения прямоугольника диалогового окна в экранные единицы измерения (пикселы)
`NextDlgCtrl`	Перемещает фокус ввода на следующий элемент диалогового окна
`PrevDlgCtrl`	Перемещает фокус ввода на предыдущий элемент диалогового окна
`SetDefID`	Устанавливает в диалоговом окне управляющий элемент по умолчанию
`SetHelpID`	Задает идентификатор контекстной справки по умолчанию

В шаблонах ресурсов диалоговых окон могут быть использованы различные элементы управления (рис. 8.2). В их число входят: рисунок (Picture Control), статический текст (Static Text), элемент редактирования (Edit Control), группа (Group Box), нажимаемая кнопка (Button), отмечаемая кнопка (Check Box),

Таблица 8.3. *Виртуальные методы класса* `CDialog`

Метод	Описание
`OnCancel`	Закрывает диалоговое окно (при нажатии кнопки **Отмена** или клавиши <Esc>). Базовая реализация класса `CDialog` предусматривает, что в этом случае функция `DoModal` возвращает значение `IDCANCEL`
`OnInitDialog`	Управляет инициализацией диалогового окна (вы можете перегрузить этот метод и добавить ваш собственный код инициализации)
`OnOk`	Закрывает диалоговое окно при нажатии кнопки **OK**. Базовая реализация класса `CDialog` предусматривает, что в этом случае функция `DoModal` возвращает значение `IDOK`
`OnSetFont`	Задает шрифт, используемый в диалоговом окне

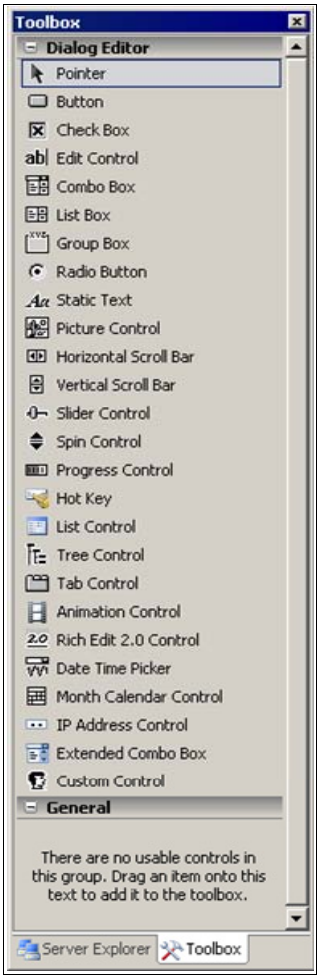


Рис. 8.2. Управляющие элементы шаблонов ресурсов диалоговых окон

радиокнопка (Radio Button), комбинированный список (Combo Box), список (List Box), горизонтальная (Horizontal Scroll Bar) и вертикальная (Vertical Scroll Bar) полосы прокрутки, "спин" (Spin Control), индикатор прогресса (Progress Control), бегунок (Slider Control), аниматор (Animation Control) и ряд других элементов.

Диалоговое окно всегда содержит две кнопки — **Отмена** (Cancel) и **OK**. Это окно автоматически закрывается, когда нажимается либо кнопка **Отмена** с помощью метода `CDialog::OnCancel`, либо кнопка **OK** с помощью метода `CDialog::OnOK`, либо когда вы в своем коде вызываете метод `CDialog::EndDialog`. Закрыть диалоговое окно можно с помощью кнопки **Закрыть**, расположенной в строке заголовка справа, или посредством клавиши <Esc>.

8.2. Демонстрационный пример с модальным диалогом и управляющими элементами — радиокнопками

Рассматриваемое далее приложение с акселераторами, меню, таблицей строк аналогично рассмотренному в *разд. 6.2 главы 6*, но дополнительно содержит меню для работы с модальным диалоговым окном и модальное диалоговое окно с управляющими элементами — радиокнопками. Соответствующий демонстрационный проект находится на прилагаемом компакт-диске, в папке \Глава 08\UsgModalDlg, а файловый состав проекта показан на рис. 8.3.

Копию проекта UsgModalDlg, для экспериментов, создавайте самым простым способом — скопируйте папку проекта с компакт-диска. Файлы StdAfx.h, StdAfx.cpp, Main.cpp, MainThread.hpp в данном проекте такие же, как и в проекте UsgMenu_MFC из *главы 6*, а файлы MainThread.cpp, FrameWnd.hpp, FrameWnd.cpp, Dialog.hpp и Dialog.cpp модифицированы в соответствии с листингами 8.1—8.5. Два оставшихся файла — resource.h и Resourse.rc — созданы редактором ресурсов и поддерживаются им. Ресурсы проекта представлены на рис. 8.4. Файлы Dialog.hpp и Dialog.cpp с определением класса `Dialog`, производного от `CDialog` библиотеки MFC, следует, в аналогичных приложениях, включать в проект с помощью мастера, вызываемого командой **Project | Add Class...** после выбора проекта во вкладке **Solution Explorer**. Добавление в класс `Dialog` таблицы сообщений, прототипов и определений функций, обрабатывающих сообщения от управляющих элементов диалогового окна, выполняйте с помощью мастера обработки событий, вызываемого командой **Add Event Handler...** контекстного меню соответствующего управляющего элемента *(см. разд. 6.3)*.

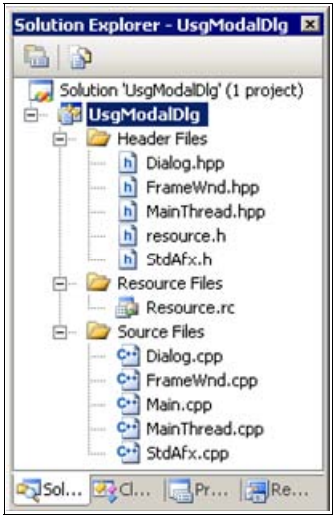


Рис. 8.3. Файловый состав проекта UsgModalDlg

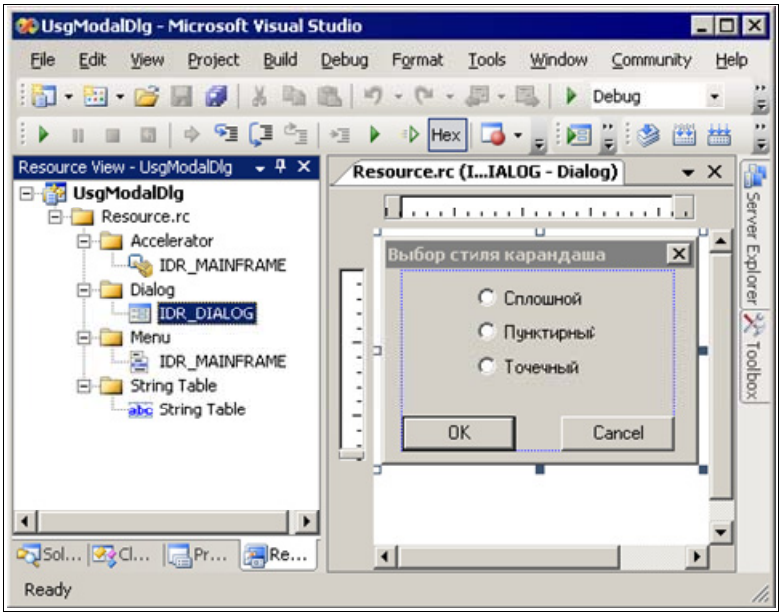


Рис. 8.4. Ресурсы проекта UsgModalDlg

Листинг 8.1. Файл MainThread.cpp

```
/*
   Файл                : MainThread.cpp

   Проект              : демонстрация создания Windows-приложения на
                         основе библиотеки классов MFC (ресурс —
                         модальное диалоговое окно с радиокнопками)

   Назначение          : реализация класса "MainThread",
                         производного от класса "CWinApp" библиотеки
                         классов MFC (инициализация приложения перед
                         запуском)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAfx.h"                  // Прекомпилируемый файл
#include "MainThread.hpp"            // Объявление класса MainThread
#include "FrameWnd.hpp"              // Объявление класса FrameWnd
// Идентификаторы ресурсов (поддержка редактором ресурсов)
#include "resource.h"

// Переопределение виртуальной функции для инициализации приложения
//   перед запуском
BOOL MainThread :: InitInstance( void )
{
    // Указатель на фрейм окна
    FrameWnd            *pFrame;

    // Размещение фрейма окна в динамической памяти
    pFrame = new FrameWnd;
    ASSERT_VALID( pFrame );          // Обработка ошибки размещения

    // Загрузка ресурсов, регистрация класса окна и создание окна
    BOOL                rc = pFrame->LoadFrame( IDR_MAINFRAME );
    if( !rc )
    {
        TRACE0( "\n Ошибка 1. Метод LoadFrame завершен с "
                "ошибкой \n" );
        exit( 1 );
    }
```

```
    // Отображение окна
    pFrame->ShowWindow( SW_SHOW );
    // Обновление окна (генерирует WM_PAINT)
    pFrame->UpdateWindow( );

    // Созданный фрейм делаем главным окном приложения
    this->m_pMainWnd = pFrame;

    return TRUE;
}
```

Листинг 8.2. Файл FrameWnd.hpp

```
/*
   Файл                : FrameWnd.hpp

   Проект              : демонстрация создания Windows-приложения на
                         основе библиотеки классов MFC (ресурс —
                         модальное диалоговое окно с радиокнопками)

   Назначение          : объявление класса "FrameWnd", производного
                         от класса "CFrameWnd" библиотеки классов
                         MFC (обработка сообщений главного окна)

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _FrameWnd_hpp
    #define _FrameWnd_hpp

    // Объявление класса
    class FrameWnd : public CFrameWnd
    {
        // Данные

    private:

        // Вершины ромба
        CPoint          Vert0;      // Левая вершина
        CPoint          Vert1;      // Верхняя вершина
        CPoint          Vert2;      // Правая вершина
        CPoint          Vert3;      // Нижняя вершина
```

```
        // Параметры карандаша
        int             m_PenWidth; // Толщина
        COLORREF        m_PenRGB;   // RGB-цвет
        int             m_PenStyle; // Стиль

        // Методы

    public:

        // Конструктор умолчания (инициализация параметров
        //   карандаша)
        FrameWnd( void );

    protected:

        // Прототипы функций, обрабатывающих сообщения
        // Перерисовка окна
        afx_msg void OnPaint( void );
        // Обработка команды Файл | Выход
        afx_msg void OnMenuFileExit( void );
        // Обработка команды Стиль карандаша | Сплошной
        afx_msg void OnMenuPenstyleSolid( void );
        // Обработка команды Стиль карандаша | Пунктирный
        afx_msg void OnMenuPenstyleDash( void );
        // Обработка команды Стиль карандаша | Точечный
        afx_msg void OnMenuPenstyleDot( void );
        // Обработчик команды Радиокнопки | Диалог
        afx_msg void OnMenuRadioButtonDialog(void);

    private:

        // Изображение ромба в клиентской области окна
        // Задание вершин ромба
        void SetRhombVertexes( void );
        // Рисование ромба
        void DrawRhomb( CDC *pDC );

        // Объявление таблицы сообщений
        DECLARE_MESSAGE_MAP( );
    };

#endif
```

Листинг 8.3. Файл FrameWnd.cpp

```
/*
   Файл                : FrameWnd.cpp

   Проект              : демонстрация создания Windows-приложения на
                         основе библиотеки классов MFC (ресурс —
                         модальное диалоговое окно с радиокнопками)

   Назначение          : реализация класса "FrameWnd", производного
                         от класса "CFrameWnd" библиотеки классов
                         MFC (обработка сообщений главного окна)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAfx.h"                  // Прекомпилируемый файл
#include "FrameWnd.hpp"              // Объявление класса FrameWnd
#include "Dialog.hpp"                // Объявление класса Dialog
// Идентификаторы ресурсов (поддержка редактором ресурсов)
#include "resource.h"

// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( FrameWnd, CFrameWnd )
    ON_WM_PAINT( )
    ON_COMMAND( ID_FILE_EXIT, OnMenuFileExit )
    ON_COMMAND( ID_PENSTYLE_SOLID, OnMenuPenstyleSolid )
    ON_COMMAND( ID_PENSTYLE_DASH, OnMenuPenstyleDash )
    ON_COMMAND( ID_PENSTYLE_DOT, OnMenuPenstyleDot )
    ON_COMMAND( ID_DIALOG, OnMenuRadioButtonDialog )
END_MESSAGE_MAP( )

// Конструктор умолчания (инициализация параметров карандаша)
FrameWnd :: FrameWnd( void )
{
    m_PenWidth = 1;
    m_PenRGB   = RGB( 255, 0, 0 );
    m_PenStyle = PS_SOLID;
}

// Обработка команды Файл | Выход
afx_msg void FrameWnd :: OnMenuFileExit( void )
{
    BOOL                rc = MessageBeep( -1 );
```

```
    if( !rc )
    {
        TRACE0( "\n Ошибка 2. Неверное завершение MessageBeep \n" );
        exit( 2 );
    }
    rc = DestroyWindow( );
    if( !rc )
    {
        TRACE0( "\n Ошибка 3. Окно не было разрушено \n" );
        exit( 3 );
    }

    return afx_msg void( );
}

// Обработчик сообщения WM_PAINT: рисование ромба
afx_msg void FrameWnd :: OnPaint( void )
{
    // Создаем объект для рисования и вывода
    CPaintDC            PaintDC( this );

    // Рисуем ромб
    SetRhombVertexes( );
    DrawRhomb( &PaintDC );

    return afx_msg void( );
}

// Обработка команды Стиль карандаша | Сплошной
afx_msg void FrameWnd :: OnMenuPenstyleSolid( void )
{
    m_PenStyle = PS_SOLID;
    CWnd :: Invalidate( );

    return afx_msg void( );
}

// Обработка команды Стиль карандаша | Пунктирный
afx_msg void FrameWnd :: OnMenuPenstyleDash( void )
{
    m_PenStyle = PS_DASH;
    CWnd :: Invalidate( );

    return afx_msg void( );
}
```

```
// Обработка команды Стиль карандаша | Точечный
afx_msg void FrameWnd :: OnMenuPenstyleDot( void )
{
    m_PenStyle = PS_DOT;
    CWnd :: Invalidate( );

    return afx_msg void( );
}

// Обработчик команды Радиокнопки | Диалог
//   !!! Будьте внимательнее — новая информация
afx_msg void FrameWnd :: OnMenuRadioButtonDialog( void )
{
    // Создаем объект диалога — автоматически вызывается
    //   конструктор
    Dialog         Dlg;

    // Передаем стиль карандаша из класса FrameWnd в класс
    //   Dialog
    Dlg.m_PenStyle = m_PenStyle;

    // Создается модальное диалоговое окно — оно будет
    //   автоматически закрыто после нажатия OK или Cancel
    int                DlgResult = static_cast<int>
                                   ( Dlg.DoModal( ) );
    if (DlgResult == IDOK)
    {                              // Выход из диалога по OK

        // Передаем стиль карандаша из класса Dialog в класс
        //   FrameWnd
        m_PenStyle = Dlg.m_PenStyle;
    }

    CWnd::Invalidate( );           // Перерисовываем окно

    return afx_msg void( );
}

// Задание вершин ромба
void FrameWnd :: SetRhombVertexes( void )
{
    CRect              ClientRect; // Клиентская область окна
    // Минимальное значение координаты x в клиентской области окна
    int                MinX;
```

```
    // Среднее значение координаты x в клиентской области окна
    int                 MidX;
    // Максимальное значение координаты x в клиентской области окна
    int                 MaxX;
    // Минимальное значение координаты y в клиентской области окна
    int                 MinY;
    // Среднее значение координаты y в клиентской области окна
    int                 MidY;
    // Максимальное значение координаты y в клиентской области окна
    int                 MaxY;

    // Получение размеров клиентской области окна
    CWnd :: GetClientRect( &ClientRect );

    MinX = 0;
    MaxX = ClientRect.Width( ) — 1;
    MidX = MaxX / 2;

    MinY = 0;
    MaxY = ClientRect.Height( ) — 1;
    MidY = MaxY / 2;

    // Задание координат вершин
    Vert0 = CPoint( MinX, MidY );
    Vert1 = CPoint( MidX, MinY );
    Vert2 = CPoint( MaxX, MidY );
    Vert3 = CPoint( MidX, MaxY );

    return;
}

// Рисование ромба
void FrameWnd :: DrawRhomb(
    CDC                 *pDC )      // Указатель на контекст
                                    //   устройства
{
    CPen                NewPen;     // Новый карандаш
    CPen                *pOldPen;   // Указатель на старый карандаш

    // Создаем новый карандаш
    NewPen.CreatePen( m_PenStyle, m_PenWidth, m_PenRGB );

    // Включаем новый карандаш в контекст устройства и запоминаем
    //   параметры старого карандаша
    pOldPen = pDC->SelectObject( &NewPen );
```

```
    // Позиционируемся в позицию начальной вершины ромба
    pDC->MoveTo( Vert0 );

    // Рисуем стороны ромба
    pDC->LineTo( Vert1 );
    pDC->LineTo( Vert2 );
    pDC->LineTo( Vert3 );
    pDC->LineTo( Vert0 );

    // Восстанавливаем параметры старого карандаша в контексте
    //   устройства
    pDC->SelectObject( pOldPen );

    return;
}
```

Листинг 8.4. Файл Dialog.hpp

```
/*
   Файл                : Dialog.hpp

   Проект              : демонстрация создания Windows-приложения на
                         основе библиотеки классов MFC (ресурс —
                         модальное диалоговое окно с радиокнопками)

   Назначение          : объявление класса "Dialog", производного
                         от класса "CDialod" библиотеки классов
                         MFC (обработка сообщений)

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _Dialog_hpp
    #define _Dialog_hpp

    // Объявление класса
    class Dialog : public CDialog
    {
        // Данные

    private:
```

```
        // Для карандаша
        // Указатель на радиокнопку задания стиля "Сплошной"
        CButton         *pRadioSolid;
        // Указатель на радиокнопку задания стиля "Штриховой"
        CButton         *pRadioDash;
        // Указатель на радиокнопку задания стиля "Точечный"
        CButton         *pRadioDot;

    public:

        int             m_PenStyle; // Стиль карандаша

    public:

        // Конструктор — передает идентификатор окна в базовый
        //   класс
        Dialog( CWnd *pParent = 0 );

    protected:

        // Перегружаем виртуальный метод для инициализации
        virtual BOOL OnInitDialog( void );
        // Перегружаем виртуальный метод для закрытия окна по
        //   кнопке OK
        virtual void OnOK( void );

        // Обработчики радиокнопок (в данном случае они избыточны)
        /*afx_msg void OnRadioSolidClick( void );
        afx_msg void OnRadioDashClick( void );
        afx_msg void OnRadioDotClick( void );*/

        // Объявление таблицы сообщений
        DECLARE_MESSAGE_MAP( );
    };

#endif
```

Листинг 8.5. Файл Dialog.cpp

```
/*
   Файл              : Dialog.cpp

   Проект            : демонстрация создания Windows-приложения на
                       основе библиотеки классов MFC (ресурс —
                       модальное диалоговое окно с радиокнопками)
```

```
   Назначение          : реализация класса "Dialog", производного
                         от класса "CDialod" библиотеки классов
                         MFC (обработка сообщений окна)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAfx.h"                  // Прекомпилируемый файл
// Объявление класса Dialog
#include "Dialog.hpp"
// Идентификаторы ресурсов (поддержка редактором ресурсов)
#include "resource.h"

// Определение таблицы сообщений (в данном случае она избыточна)
BEGIN_MESSAGE_MAP( Dialog, CDialog )
    /*ON_BN_CLICKED( IDC_RADIO_SOLID, OnRadioSolidClick )
    ON_BN_CLICKED( IDC_RADIO_DASH, OnRadioDashClick )
    ON_BN_CLICKED( IDC_RADIO_DOT, OnRadioDotClick )*/
END_MESSAGE_MAP( )

// Конструктор
Dialog :: Dialog(
    CWnd                *pParent )  // Указатель на родительское
                                    //   окно (NULL — родительским
                                    //   является главное окно)
    : CDialog( IDR_DIALOG, pParent )
    { }

// Инициализация окна диалога
BOOL Dialog :: OnInitDialog( void )
{
    // Вызываем метод базового класса
    CDialog :: OnInitDialog( );

    // Получаем указатели на радиокнопки
    pRadioSolid = static_cast< CButton * >
                            ( GetDlgItem( IDC_RADIO_SOLID ) );
    pRadioDash  = static_cast< CButton * >
                            ( GetDlgItem( IDC_RADIO_DASH ) );
    pRadioDot   = static_cast< CButton * >
                            ( GetDlgItem( IDC_RADIO_DOT ) );

    // Устанавливаем активную радиокнопку
    switch ( m_PenStyle )
```

```
    {

    case PS_SOLID:
        pRadioSolid->SetCheck( 1 );
        break;

    case PS_DASH:
        pRadioDash->SetCheck( 1 );
        break;

    case PS_DOT:
        pRadioDot->SetCheck( 1 );
        break;
    }

    return TRUE;
}

// Обрабатываем закрытие по кнопке OK
void Dialog :: OnOK( void )
{
    // Вызываем метод базового класса
    CDialog :: OnOK( );

    // Избыточный программный код
    /*pRadioSolid = static_cast< CButton * >
                              ( GetDlgItem( IDC_RADIO_SOLID ) );
    pRadioDash  = static_cast< CButton * >
                              ( GetDlgItem( IDC_RADIO_DASH ) );
    pRadioDot   = static_cast< CButton * >
                              ( GetDlgItem( IDC_RADIO_DOT ) );*/

    if ( pRadioSolid->GetCheck( ) == 1 )
    {
        m_PenStyle = PS_SOLID;
    }
    else if ( pRadioDash->GetCheck( ) == 1 )
    {
        m_PenStyle = PS_DASH;
    }
    else
    {
        m_PenStyle = PS_DOT;
    }
```

```
    return;
}

// Обработчики радиокнопок — в данном случае они избыточны
и закомментированы
// Обработчик радиокнопки "Сплошной"
/*void Dialog :: OnRadioSolidClick( void )
{
    m_PenStyle = PS_SOLID;

    return;
}

// Обработчик радиокнопки "Пунктирный"
void Dialog :: OnRadioDashClick( void )
{
    m_PenStyle = PS_DASH;

    return;
}

// Обработчик радиокнопки "Точечный"
void Dialog :: OnRadioDotClick( void )
{
    m_PenStyle = PS_DOT;

    return;
}*/
```

Главное окно приложения представлено на рис. 8.5, а модальное диалоговое окно с радиокнопками показано на рис. 8.6.

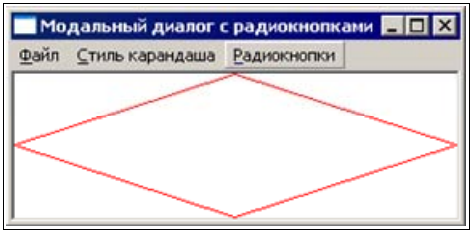


Рис. 8.5. Главное окно проекта UsgModalDlg

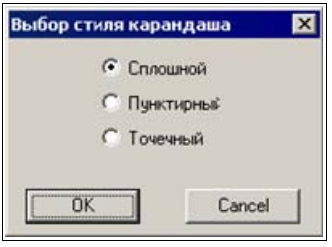


Рис. 8.6. Модальный диалог с радиокнопками проекта UsgModalDlg

8.2.1. Создание шаблона ресурсов диалогового окна

Шаблон ресурсов диалогового окна проще всего создать с помощью редактора ресурсов. Для этого достаточно на вкладке **Resource View** выбрать ваш проект (в нашем случае это UsgModalDlg) и выполнить команду **Project | Add Resource...**, а в появившемся окне **Add Resource** выбрать ресурс **Dialog** и нажать кнопку **New**. В результате создается шаблон с двумя кнопками — **OK** и **Cancel**. Размеры окна, местоположение и размеры указанных кнопок можно легко изменить с помощью мыши, сделав их, например, такими, как на рис. 8.6.

В результате, во вкладке **Resource View**, появится ресурс **Dialog** с некоторым идентификатором, заданным по умолчанию редактором ресурсов. Если для данного идентификатора двойным щелчком левой кнопки мыши загрузить ресурс диалогового окна в окно редактирования, а затем, с помощью правой кнопки мыши, вызвать контекстное меню и выполнить команду **Properties**, то можно задать требуемые свойства шаблона ресурсов диалогового окна. Для демонстрационного проекта указанные свойства показаны на рис. 8.7.

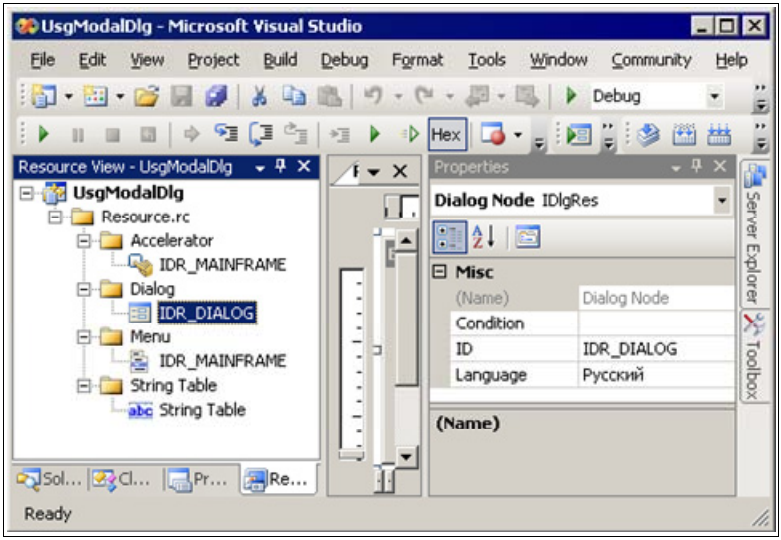


Рис. 8.7. Свойства ресурса диалогового окна проекта UsgModalDlg

Для размещения в диалоговом окне нужных управляющих элементов можно воспользоваться панелью **Toolbox** (Панель инструментов) (см. рис. 8.2). В демонстрационном примере такими управляющими элементами являются три радиокнопки, с помощью которых задается стиль линии — сплошной, штриховой или пунктирный.

Примечание

Отличительным признаком радиокнопок является то, что активной является только одна кнопка из группы.

Для добавления радиокнопки в диалоговое окно выберите на панели **Toolbox** соответствующую иконку — радиокнопку (Radio Button) и перетащите ее в нужное место диалогового окна. Размеры и текст радиокнопок можно легко изменить. Результат размещения радиокнопок в нашем проекте был показан на рис. 8.4.

Для настройки свойств радиокнопки достаточно вызвать ее контекстное меню и выбрать пункт **Properties**. В появившейся одноименной вкладке задать свойства радиокнопки. Свойства радиокнопок для демонстрационного примера показаны на рис. 8.8.

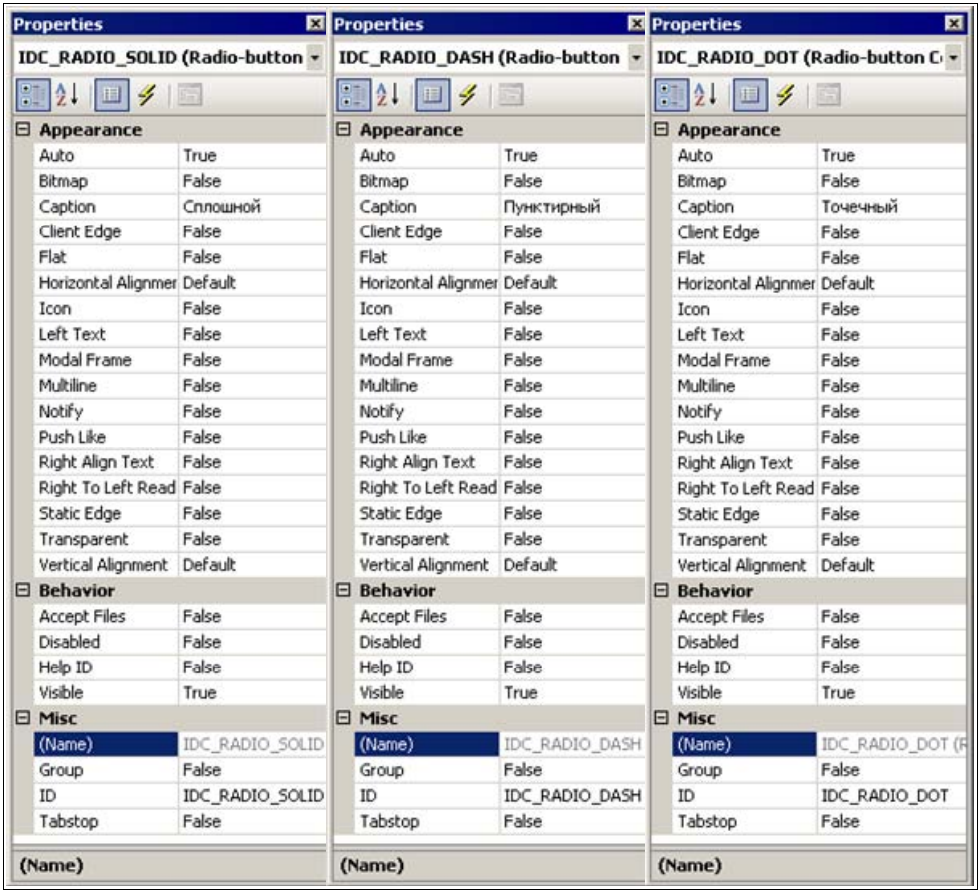


Рис. 8.8. Свойства радиокнопок шаблона ресурсов диалогового окна проекта UsgModalDlg

Полученный шаблон ресурсов диалогового окна хранится как составная часть в файле проекта Resource.rc, а идентификаторы диалогового окна и его управляющих элементов — в файле resource.h. Напоминаем, что эти файлы не следует редактировать "вручную" — они создаются и модифицируются редактором ресурсов.

8.2.2. Программная поддержка модального диалогового окна с радиокнопками

Демонстрационный проект содержит три класса (рис. 8.9—8.11): класс `MainThread`, производный от класса `CWinApp` библиотеки MFC, класс `FrameWnd`, производный от класса `CFrameWnd`, и класс `Dialog`, производный от класса `CDialog`.

Класс `FrameWnd` содержит *таблицу сообщений*, которая предусматривает создание диалогового окна для задания стиля линии (см. рис. 8.6):

```
// Фрагмент файла FrmWnd.hpp
    ...
    protected:

        // Прототипы функций, обрабатывающих сообщения
        // Перерисовка окна
        afx_msg void OnPaint( void );
        // Обработка команды Файл | Выход
        afx_msg void OnMenuFileExit( void );
        // Обработка команды Стиль карандаша | Сплошной
        afx_msg void OnMenuPenstyleSolid( void );
        // Обработка команды Стиль карандаша | Пунктирный
        afx_msg void OnMenuPenstyleDash( void );
        // Обработка команды Стиль карандаша | Точечный
        afx_msg void OnMenuPenstyleDot( void );
        // Обработка команды Радиокнопки | Диалог
        afx_msg void OnMenuRadioButtonDialog(void);
    ...
        // Объявление таблицы сообщений
        DECLARE_MESSAGE_MAP( );

// Фрагмент файла FrameWnd.cpp
    ...
// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( FrameWnd, CFrameWnd )
    ON_WM_PAINT( )
    ON_COMMAND( ID_FILE_EXIT, OnMenuFileExit )
```

```
    ON_COMMAND( ID_PENSTYLE_SOLID, OnMenuPenstyleSolid )
    ON_COMMAND( ID_PENSTYLE_DASH, OnMenuPenstyleDash )
    ON_COMMAND( ID_PENSTYLE_DOT, OnMenuPenstyleDot )
    ON_COMMAND( ID_DIALOG, OnMenuRadioButtonDialog )
END_MESSAGE_MAP( )
    ...
// Обработка команды Радиокнопки | Диалог
//   !!! Будьте внимательнее — новая информация
afx_msg void FrameWnd :: OnMenuRadioButtonDialog( void )
{
    // Создаем объект диалога — автоматически вызывается
    //   конструктор
    Dialog         Dlg;

    // Передаем стиль карандаша из класса FrameWnd в класс
    //   Dialog
    Dlg.m_PenStyle = m_PenStyle;

    // Создается модальное диалоговое окно — оно будет
    //   автоматически закрыто после нажатия OK или Cancel
    int                DlgResult = static_cast<int>
                                   ( Dlg.DoModal( ) );
    if (DlgResult == IDOK)
    {                              // Выход из диалога по OK

        // Передаем стиль карандаша из класса Dialog в класс
        //   FrameWnd
        m_PenStyle = Dlg.m_PenStyle;
    }

    CWnd::Invalidate( );           // Перерисовываем окно

    return afx_msg void( );
}
```

При выполнении команды **Радиокнопки | Диалог** вызывается метод `FrameWnd::OnMenuRadioButtonDialog`, который и создает диалоговое окно, показанное ранее на рис. 8.6. Для создания модального окна диалога в функции `OnMenuRadioButtonDialog` используется виртуальный метод `DoModal`.

Класс `Dialog`, производный от класса `CDialog` библиотеки MFC, обеспечивает работу с диалоговым окном. Рассмотрим этот класс подробнее. В объявлении класса (см. листинг 8.4) определены следующие члены класса: переменная

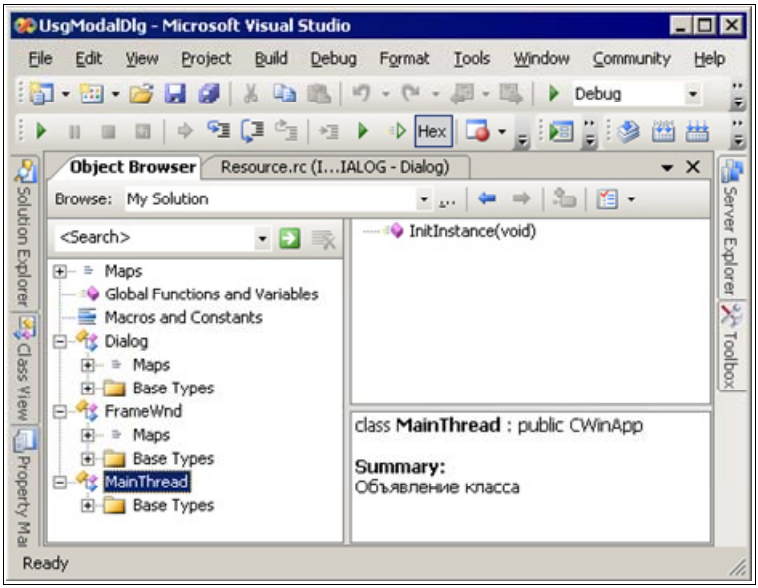


Рис. 8.9. Класс `MainThread` проекта UsgModalDlg

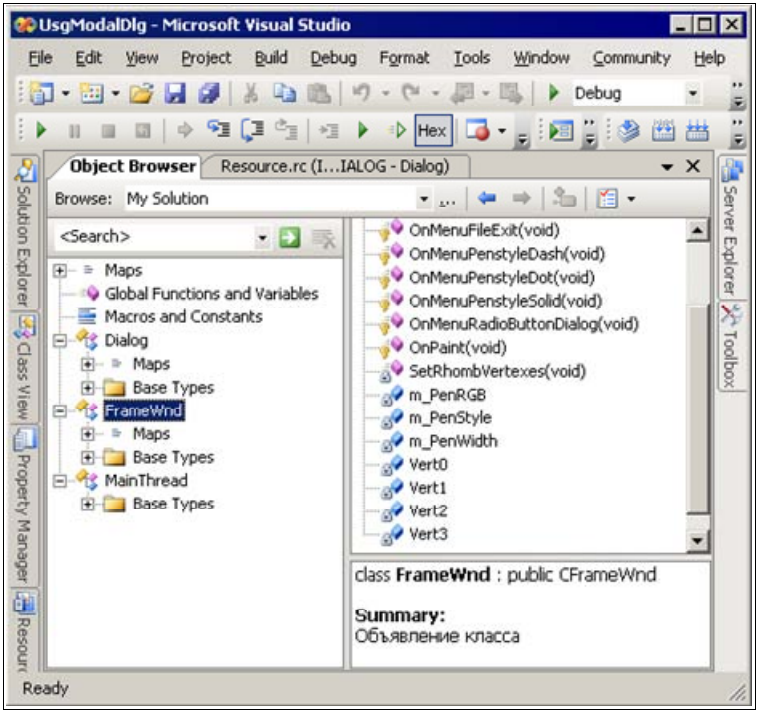


Рис. 8.10. Класс `FrameWnd` проекта UsgModalDlg

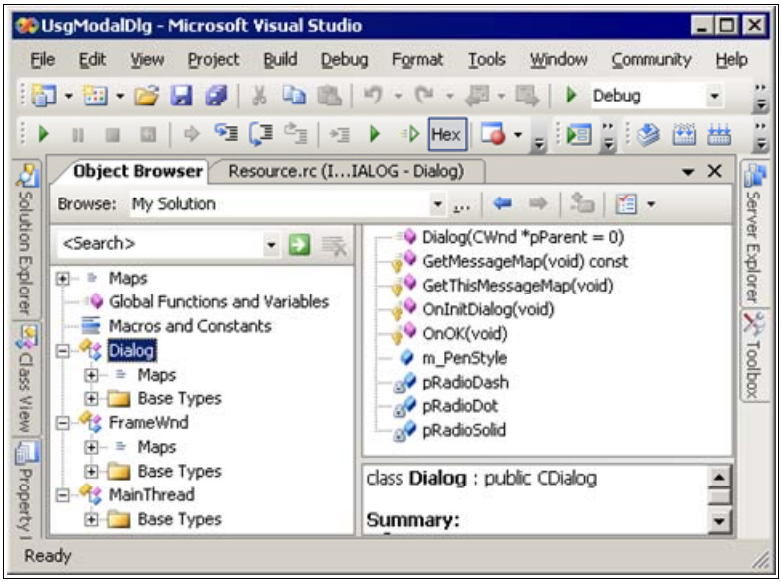


Рис. 8.11. Класс `Dialog` проекта UsgModalDlg

`m_PenStyle` — стиль карандаша, и три указателя на управляющие элементы-радиокнопки — `pRadioSolid`, `pRadioDash` и `pRadioDot`, которые доступны только в методах класса `Dialog`. В классе имеются также несколько методов, прототипы которых находятся в объявлении класса. К их числу относятся:

- конструктор класса;
- прототип переопределяемого виртуального метода `OnInitDialog`, выполняющего требуемую инициализацию окна диалога;
- прототип переопределяемого виртуального метода `OnOK`, обеспечивающего получение данных из окна диалога и их использование;
- прототипы методов `OnRadioSolidClick`, `OnRadioDashClick` и `OnRadioDotClick`, обеспечивающих работу с радиокнопками, задающими стиль линии пера (карандаша), и макрос `DECLARE_MESSAGE_MAP`, объявляющий таблицу сообщений.

В реализации класса `Dialog` (см. листинг 8.5), прежде всего, содержится таблица сообщений:

```
// Определение таблицы сообщений (в данном случае она избыточна)
BEGIN_MESSAGE_MAP( Dialog, CDialog )
    /*ON_BN_CLICKED( IDC_RADIO_SOLID, OnRadioSolidClick )
    ON_BN_CLICKED( IDC_RADIO_DASH, OnRadioDashClick )
    ON_BN_CLICKED( IDC_RADIO_DOT, OnRadioDotClick )*/
END_MESSAGE_MAP( )
```

При выборе радиокнопки формируется сообщение `BN_CLICKED`, которому в таблице сообщений соответствует макрос `ON_BN_CLICKED`. Таким образом, таблица сообщений ставит в соответствие радиокнопкам **Сплошной, Пунктирный** и **Точечный** (см. рис. 8.4) обрабатывающие функции `OnRadioSolidClick`, `OnRadioDashClick` и `OnRadioDotClick`.

Определение конструктора в файле реализации класса имеет вид:

```
// Конструктор
Dialog :: Dialog(
    CWnd                *pParent )  // Указатель на родительское
                                    //   окно (NULL — родительским
                                    //   является главное окно)
    : CDialog( IDR_DIALOG, pParent )
    { }
```

Обратите внимание на то, что в конструктор базового класса `CDialog` необходимо, в качестве первого аргумента, передать идентификатор ресурса окна диалога `IDR_DIALOG`.

Примечание

В данном демонстрационном проекте в классе `Dialog` таблица сообщений и все, что к ней относится, являются избыточными (в исходном коде они закомментированы). Если указанные комментарии удалить, то проект сохранит работоспособность. Такое оформление принято, чтобы показать общую схему программной поддержки диалогового окна.

Совет

Постарайтесь уяснить, почему такая избыточность имеет место.

Реализация класса `Dialog` содержит переопределение виртуального метода `OnInitDialog`, выполняющего требуемую инициализацию модального окна диалога. В этом методе следует обязательно вызвать соответствующий метод базового класса, а затем выполнить необходимую инициализацию окна. В методе `OnInitDialog` для определения значений указателей радиокнопок используется метод `GetDlgItem`, наследуемый из класса `CWnd` библиотеки MFC. Далее, в соответствии со значением члена класса `Dialog::m_PenStyle` устанавливается начальное состояние радиокнопок. Для этой цели используется метод `CButton::SetCheck`.

Замечание

Метод `GetDlgItem`, наследуемый из класса `CWnd` библиотеки MFC, можно использовать не только для получения значений указателей на радиокнопки, но и

для получения значений указателей на любые другие управляющие элементы диалогового окна.

Файл реализации класса `Dialog` также содержит переопределение виртуального метода `OnOK`, обеспечивающего получение данных из диалогового окна и их использование. С этой целью, с помощью метода `CButton::GetCheck`, определяется состояние активной радиокнопки и задаваемый ею стиль присваивается члену класса `Dialog::m_PenStyle`. Теперь удвойте ваше внимание — важная информация! Для получения информации из модального диалогового окна следует закрыть его нажав на кнопку **OK**. При этом метод `DoModal` возвращает значение `IDOK` и значение стиля карандаша, заданное в диалоговом окне, присваивается переменной `m_PenStyle` класса `FrameWnd`:

```
// Передаем стиль карандаша из класса FrameWnd в класс
//   Dialog
Dlg.m_PenStyle = m_PenStyle;

// Создается модальное диалоговое окно — оно будет
//   автоматически закрыто после нажатия OK или Cancel
int                 DlgResult = static_cast<int>
                                ( Dlg.DoModal( ) );
if (DlgResult == IDOK)
{                               // Выход из диалога по OK

    // Передаем стиль карандаша из класса Dialog в класс
    //   FrameWnd
    m_PenStyle = Dlg.m_PenStyle;
}
```

Обратите внимание также и на то, что в перегруженном методе `OnOK` следует вызвать соответствующий метод базового класса:

```
// Вызываем метод базового класса
CDialog :: OnOK( );
```

И, наконец, файл реализации класса `Dialog` содержит определение методов `OnRadioSolidClick`, `OnRadioDashClick` и `OnRadioDotClick`, выполняемых при выборе соответствующих радиокнопок. Каждый из этих методов тривиален и состоит в присваивании члену класса `FrameWnd::m_PenStyle` соответствующего стиля карандаша (пера).

8.3. Демонстрационный пример с немодальным диалоговым окном и управляющими элементами — однострочным редактором и статическим текстом

Прежде чем перейти к рассмотрению демонстрационного примера, кратко рассмотрим класс `CEdit` библиотеки MFC, поддерживающий работу окон редактирования.

8.3.1. Окна редактирования: класс *CEdit* [5]

Окно редактирования — это прямоугольное окно с большими возможностями. Окна редактирования выполняют множество полезных функций, значительно облегчающих обработку текста. Например, приложение Блокнот, поставляемое вместе с Windows, представляет собой большое окно редактирования с меню. Большая часть функциональных возможностей Блокнота определяется как раз возможностями окна редактирования.

Существуют однострочные и многострочные окна редактирования. Однострочное окно редактирования выводит весь текст в одной строке, независимо от наличия в нем символов возврата каретки. Многострочное окно выводит текст в несколько строк, разделенных символами возврата каретки.

Библиотека MFC поддерживает функциональность окна редактирования с помощью класса `CEdit`.

Совет

Посмотрите и запомните место класса `CEdit` в иерархии классов библиотеки MFC, пользуясь справочной системой, встроенной в IDE Microsoft Visual Studio C++ .NET 2005. Для этого в IDE достаточно выполнить команду **Help | Index** и в поле **Look for:** появившегося окна **MFC Reference — Microsoft Visual Studio 2005 Documentation — Microsoft Document Explorer** набрать **MFC, hierarhy** и посмотреть вкладки **Hierarhy Chart** и **Hierarhy Chart Categories**.

Класс `CEdit` является непосредственным потомком класса `CWnd`, а значит, сам является окном и наследует все функциональные возможности класса `CWnd`.

Окно редактирования может быть создано в программе как *подчиненное* любого окна, но обычно оно определяется в шаблоне ресурсов диалогового окна (так проще и удобнее). Именно такой вариант мы представим здесь, а в *разд. 8.4*, для сопоставления, рассмотрим и другой вариант, когда окно редактирования будет подчинено непосредственно главному окну приложения.

Имеются следующие стили окна редактирования (объекта CEdit), представленные в табл. 8.4.

Таблица 8.4. *Стили окна редактирования*

Макроопределение стиля	Значение
ES_AUTOHSCROLL	Автоматическое прокручивание текста вправо на 10 символов при достижении правой границы окна (при нажатии клавиши <Enter> текст прокручивается обратно в позицию 0, т. е. в начало)
ES_AUTOVSCROLL	Автоматическое прокручивание текста на одну страницу вверх в многострочном окне редактирования, когда пользователь нажимает <Enter> в конце последней строки
ES_CENTER	Выравнивает текст по центру в многострочном окне редактирования
ES_LEFT	Выравнивает текст по левому краю
ES_LOWERCASE	По мере ввода автоматически трансформирует символы в нижний регистр
ES_MULTILINE	Задает многострочное окно редактирования (однострочное по умолчанию)
ES_OEMCONVERT	Поддерживает правильное преобразование символов в кодировку DOS
ES_PASSWORD	Маскирует введенные в окно символы символом *
ES_READONLY	Режим "только для чтения"
ES_RIGHT	Выравнивает текст по правому краю
ES_UPPERCASE	По мере ввода автоматически переводит символы в верхний регистр

Перечисленные в таблице особые стили редактирования можно сочетать с общими стилями окон, доступными объектам класса CWnd.

Класс CEdit имеет большой набор методов для обработки текста в окне редактирования. Среди них имеются *методы общего назначения*, применимые к любым окнам редактирования (табл. 8.5), и *специфические методы* для многострочных окон редактирования.

Примечание

Для извлечения текста из окна редактирования можно использовать метод CWnd::GetWindowText. Чтобы поместить текст в окно редактирования, пользуйтесь методом CWnd::SetWindowText.

Таблица 8.5. *Общие методы класса* `CEdit`

Метод	Описание
`Create`	Создает окно редактирования (с параметрами)
`CanUndo`	Определяет, может ли быть отменена операция редактирования
`Clear`	Удаляет выделенный текст, если он есть
`Copy`	Копирует выделенный текст, если он есть, в буфер обмена
`Cut`	Вырезает выделенный текст, если он есть, и помещает его в буфер обмена
`EmtyUndoBuffer`	Сбрасывает флаг отмены окна редактирования
`DefFirstVisibleLine`	Определяет верхнюю видимую строку в окне редактирования
`GetModify`	Определяет, было ли изменено содержимое окна редактирования
`GetPasswodChar`	Возвращает символ-заменитель при вводе пароля
`GetRect`	Возвращает прямоугольник окна редактирования
`GetSel`	Возвращает позиции первого и последнего символов выделенного текста
`LimitText`	Ограничивает длину текста в окне редактирования
`LineFromChar`	Возвращает номер строки, содержащей символ с указанным индексом
`Linelength`	Возвращает длину строки в окне редактирования
`LineScroll`	Прокручивает текст в многострочном окне редактирования
`Paste`	Вставляет данные из буфера обмена в текущую позицию курсора
`ReplaceSel`	Заменяет выделенный в окне редактирования текст указанным текстом
`SetModify`	Устанавливает или сбрасывает флаг изменения
`SetPasswordChar`	Назначает или удаляет символ-заменитель при вводе текста
`SetReadOnly`	Устанавливает режим "только для чтения"
`SetSel`	Выбирает указанный диапазон символов в окне редактирования
`Undo`	Отменяет последнюю операцию редактирования

Далее рассмотрим демонстрационный пример, в котором обратим внимание на работу с немодальным диалоговым окном, использующим однострочный редактор.

8.3.2. Демонстрационный пример NoModalDlgUsgEdit

Рассматриваемое далее приложение на базе MFC содержит немодальное диалоговое окно с управляющими элементами — однострочным редактором и статическим текстом, и меню. Соответствующий демонстрационный проект находится на прилагаемом компакт-диске, в папке \Глава 08 \NoModalDlgUsgEdit, а файловый состав проекта показан на рис. 8.12.

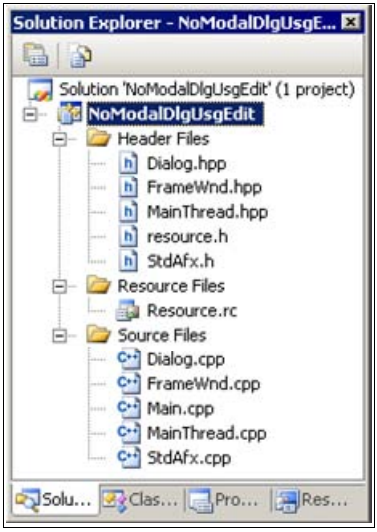


Рис. 8.12. Файловый состав проекта NoModalDlgUsgEdit

Копию проекта NoModalDlgUsgEdit для экспериментов создавайте самым простым способом — скопируйте папку проекта с компакт-диска. Файлы StdAfx.h, StdAfx.cpp, Main.cpp, MainThread.hpp, MainThread.cpp в данном проекте такие же, как в рассмотренном проекте UsgModalDlg_MFC из данной главы, а файлы FrameWnd.hpp, FrameWnd.cpp, Dialog.hpp и Dialog.cpp модифицированы в соответствии с листингами 8.6—8.9. Два оставшихся файла проекта — resource.h и Resourse.rc — созданы редактором ресурсов и поддерживаются им. Ресурсы проекта представлены на рис. 8.13. Файлы Dialog.hpp и Dialog.cpp с определением класса `Dialog`, производного от класса `CDialog` библиотеки MFC, следует, в аналогичных приложениях, включать в

проект с помощью мастера, вызываемого командой **Project | Add Class...** после выбора проекта во вкладке **Solution Explorer**. Добавление в класс `Dialog` таблицы сообщений, прототипов и определений функций, обрабатывающих сообщения от управляющих элементов диалога, выполняйте с помощью мастера обработки событий, вызываемого командой **Add Event Handler...** контекстного меню соответствующего управляющего элемента диалога.

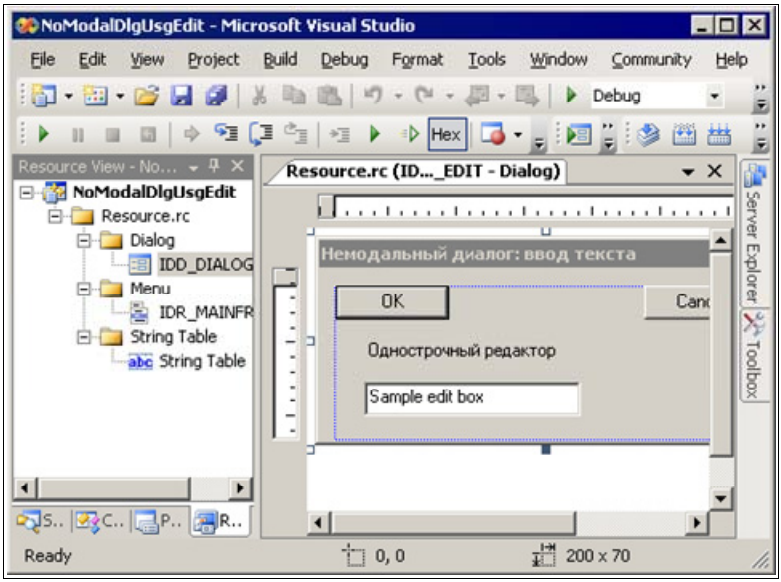


Рис. 8.13. Ресурсы проекта NoModalDlgUsgEdit

Листинг 8.6. Файл FrameWnd.hpp

```
/*
   Файл              : FrameWnd.hpp

   Проект            : демонстрация создания Windows-приложения на
                       основе библиотеки классов MFC (ресурсы —
                       диалоговое окно + однострочный редактор)

   Назначение        : объявление класса "FrameWnd", производного
                       от класса "CFrameWnd" библиотеки классов
                       MFC (обработка сообщений главного окна)

   Microsoft Visual Studio C++ .NET 2005
*/
```

```
// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _FrameWnd_hpp
    #define _FrameWnd_hpp

    // Объявление класса
    class FrameWnd : public CFrameWnd
    {
        // Данные

    private:

        // Для текста однострочного редактора
        CString         e1;

        // Методы

    public:

        // Конструктор умолчания (инициализация текста
        //   однострочного редактора)
        FrameWnd( void );

    protected:

        // Прототипы функций, обрабатывающих сообщения
        // Обработка команды "Выход"
        afx_msg void OnMenuExit( void );
        // Обработка команды "Немодальный диалог"
        afx_msg void OnMenuDialog( void );

        // Объявление таблицы сообщений
        DECLARE_MESSAGE_MAP( );
    };

#endif
```

Листинг 8.7. Файл FrameWnd.cpp

```
/*
   Файл                : FrameWnd.cpp

   Проект              : демонстрация создания Windows-приложения на
                         основе библиотеки классов MFC (ресурсы —
                         диалоговое окно + однострочный редактор)
```

```
   Назначение         : реализация класса "FrameWnd", производного
                        от класса "CFrameWnd" библиотеки классов
                        MFC (обработка сообщений главного окна)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAfx.h"                  // Прекомпилируемый файл
#include "FrameWnd.hpp"              // Объявление класса FrameWnd
#include "Dialog.hpp"                // Объявление класса Dialog
// Идентификаторы ресурсов (поддержка редактором ресурсов)
#include "resource.h"

// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( FrameWnd, CFrameWnd )
    ON_COMMAND( ID_EXIT, OnMenuExit )
    ON_COMMAND( ID_DIALOG_EDIT, OnMenuDialog )
END_MESSAGE_MAP( )

// Конструктор умолчания (инициализация окна редактирования)
FrameWnd :: FrameWnd( void )
{
    // Для текста однострочного редактора
    e1 = "Заставка";
}

// Обработка команды "Выход"
afx_msg void FrameWnd :: OnMenuExit( void )
{
    BOOL               rc = MessageBeep( -1 );
    if( !rc )
    {
        TRACE0( "\n Ошибка 2. Неверное завершение MessageBeep \n" );
        exit( 2 );
    }
    rc = DestroyWindow( );
    if( !rc )
    {
        TRACE0( "\n Ошибка 3. Окно не было разрушено \n" );
        exit( 3 );
    }
```

```
    return afx_msg void( );
}

// Обработка команды "Немодальный_диалог"
afx_msg void FrameWnd :: OnMenuDialog( void )
{
    // Размещаем объект немодального диалогового окна в динамической
    //   памяти
    Dialog              *pDlg = new Dialog;
    ASSERT_VALID( pDlg );           // Обработка ошибки размещения

    // Передаем текст для элемента редактирования
    pDlg->e1 = e1;

    // Создаем немодальное диалоговое окно
    if( !pDlg->Create( ) )
    {
        TRACE0( "\n Ошибка 4 — немодальное окно не создано \n" );
        exit( 4 );
    }

    // Перерисовка окна (формирует сообщение WM_PAINT)
    CWnd :: Invalidate( );

    return afx_msg void( );
}
```

Листинг 8.8. Файл Dialog.hpp

```
/*
   Файл               : Dialog.hpp

   Проект             : демонстрация создания Windows-приложения на
                        основе библиотеки классов MFC (ресурсы
                        диалоговое окно + однострочный редактор)

   Назначение         : объявление класса "Dialog", производного
                        от класса "CDialog" библиотеки классов
                        MFC (обработка сообщений окна)

   Microsoft Visual Studio C++ .NET 2005
*/
```

```
// Предотвращение возможности многократного подключения данного файла
#ifndef _Dialog_hpp
    #define _Dialog_hpp

    // Идентификаторы ресурсов
    #include "resource.h"

    class Dialog : public CDialog
    {
        // Данные

    private:

        // Указатель однострочного элемента редактирования
        CEdit           *pEdit;

    public:

        // Для текста элемента редактирования
        CString         e1;

        // Методы

    public:

        BOOL Create( void );

    private:

        // Инициализация диалогового окна (перегрузка виртуального
        //   метода)
        virtual BOOL OnInitDialog( void );
        // Получение информации из диалогового окна (перегрузка
        //   виртуального метода)
        virtual void OnOK( void );
        // Получение данных из диалогового окна (метод вызывается
        //   автоматически из метода UpdateData( ))
        void DoDataExchange( CDataExchange* pDX );

    };

#endif
```

Листинг 8.9. Файл Dialog.cpp

```
/*
   Файл                : Dialog.cpp

   Проект              : демонстрация создания Windows-приложения на
                         основе библиотеки классов MFC (ресурсы —
                         диалоговое окно + однострочный редактор)

   Назначение          : реализация класса "Dialog",
                         производного от класса "CDialog" библиотеки
                         классов MFC (обработка сообщений окна)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAfx.h"                  // Прекомпилируемый файл
#include "FrameWnd.hpp"              // Объявление класса FrameWnd
#include "Dialog.hpp"                // Объявление класса Dialog

// Инициализация диалога (перегрузка виртуального метода)
BOOL Dialog :: OnInitDialog( void )
{
    // Вызываем метод базового класса
    CDialog :: OnInitDialog( );

    // Получаем указатель на однострочный редактор
    pEdit = static_cast< CEdit * >( GetDlgItem( IDC_EDIT ) );
    if( !pEdit )
    {
        TRACE0( "\n Ошибка 5 — однострочный редактор не "
                "создан \n" );
        exit( 5 );
    }

    // Устанавливаем длину текста (13) для элемента редактирования
    //   и вывод начального текста
    pEdit->SetLimitText( 13 );
    pEdit->SetWindowText( e1 );

    return TRUE;
}
```

```
// ON OK: обработка закрытия кнопкой "OK"
void Dialog :: OnOK( void )
{
    // Вызываем метод базового класса
    CDialog :: OnOK( );

    // Из этого метода автоматически вызывается DoDataExchange( ),
    //   обновляющий данные из диалогового окна
    UpdateData( TRUE );

    // Визуализация обновленных данных
    AfxMessageBox( static_cast< LPCTSTR >( e1 ) );

    return;
}

// Получение данных из диалогового окна (метод вызывается автоматически
//   из метода UpdateData( ))
void Dialog :: DoDataExchange( CDataExchange *pDX )
{
    // Вызываем метод базового класса
    CDialog :: DoDataExchange( pDX );

    // Копируем в e1 текст из однострочного редактора
    DDX_Text( pDX, IDC_EDIT, e1 );

    return;
}

// Создание немодального диалога !!! Внимание, важно
BOOL Dialog :: Create( void )
{
    return CDialog :: Create( IDD_DIALOG_EDIT );
}
```

Проанализируем приведенный пример в части, относящейся к немодальному диалоговому окну и однострочному элементу редактирования.

8.3.3. Создание шаблона ресурсов немодального диалогового окна

Создание шаблона ресурсов немодального диалогового окна выполняется так же, как для модального диалогового окна, а выбор разновидности диалогового окна определяется соответствующей программной поддержкой.

Шаблон ресурсов диалогового окна можно создать так же, как это было указано в *разд. 8.2.1*. В результате создается шаблон с двумя кнопками — **OK** и **Cancel**. Размеры окна, расположение и размеры указанных кнопок можно изменить, сделав их, например, такими, как на рис. 8.13.

В результате, на вкладке **Resource View** появится ресурс **Dialog** с некоторым идентификатором, заданным редактором ресурсов по умолчанию. Если для данного идентификатора двойным щелчком левой кнопки мыши загрузить ресурс в окно редактирования, а затем с помощью правой кнопки мыши вызвать контекстное меню и выполнить команду **Properties**, то можно задать требуемые свойства шаблона ресурсов диалогового окна. Для демонстрационного проекта указанные свойства показаны на рис. 8.14.

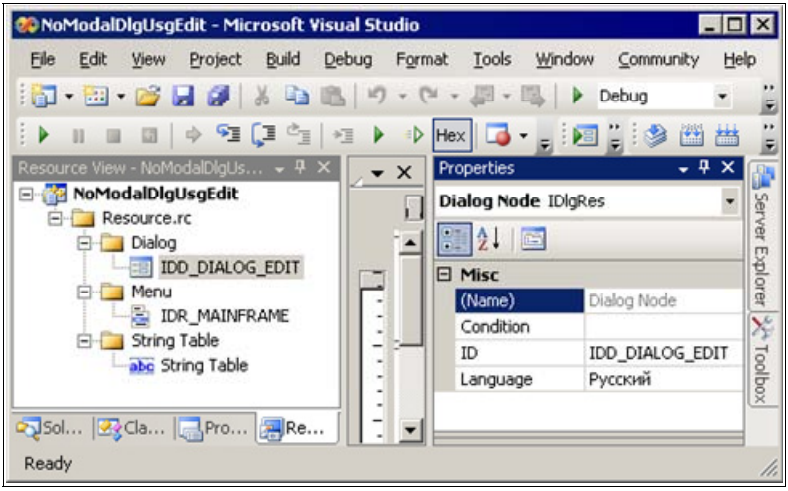


Рис. 8.14. Свойства шаблона ресурсов диалогового окна проекта NoModalDlgUsgEdit

Примечание

Обратите внимание на идентификатор шаблона ресурсов диалогового окна (`IDD_DIALOG_EDIT`) — он будет использован при программной поддержке шаблона ресурсов диалогового окна.

Для размещения в диалоговом окне нужных управляющих элементов следует воспользоваться панелью управляющих элементов **Toolbox** — см. рис. 8.2. В демонстрационном примере, в отличие от предыдущего, такими управляющими элементами являются однострочное окно редактирования и статический текст. Для их добавления в диалоговое окно достаточно последовательно выбрать их на панели **Toolbox** (Edit Control и Static Text) и перетащить в нужное место диалогового окна. Размеры и названия однострочного элемента

редактирования и статического текста можно изменить. Для демонстрационного примера их размеры и названия имеют вид, показанный на рис. 8.13.

Для настройки свойств окна редактирования или статического текста достаточно вызвать последовательно их контекстное меню, выполнить команду **Properties** и в появившейся одноименной вкладке задать требуемые свойства. Для демонстрационного примера они показаны на рис. 8.15.

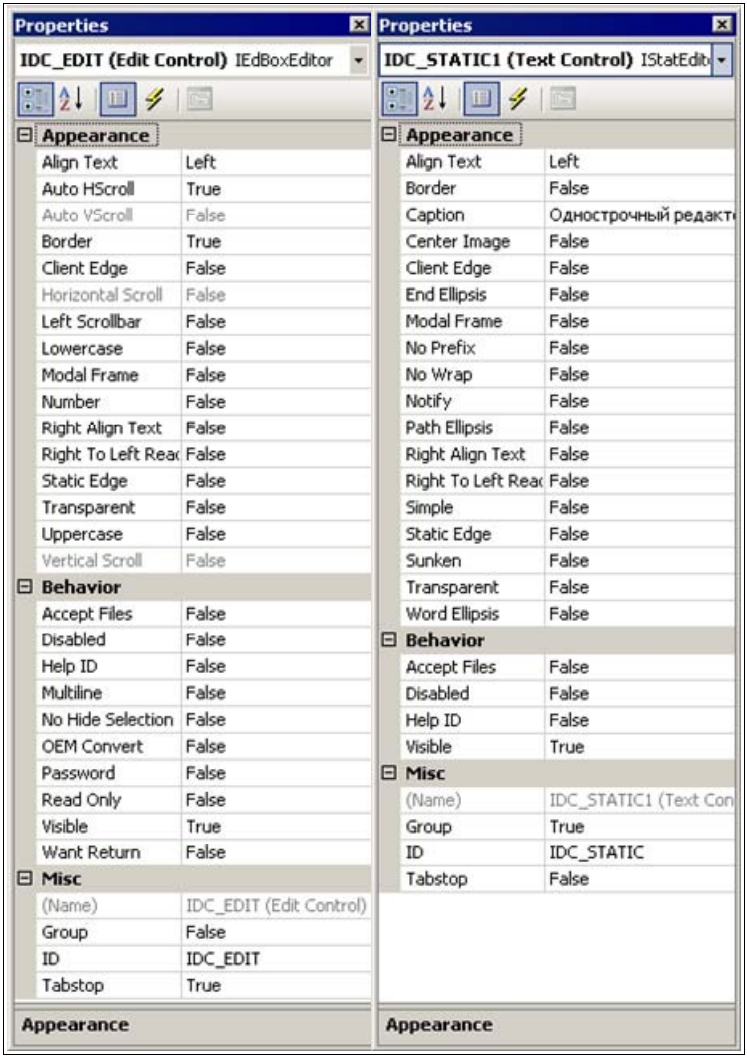


Рис. 8.15. Свойства окна редактирования и статического текста шаблона ресурсов диалогового окна проекта NoModalDlgUsgEdit

Примечание

Обратите внимание на идентификатор окна редактирования (`IDC_EDIT`) — он также будет использован при программной поддержке шаблона ресурсов диалогового окна.

Полученный шаблон ресурсов диалогового окна хранится как составная часть в файле проекта Resource.rc, а идентификаторы окна диалога и его управляющих элементов — в файле resource.h. Еще раз напоминаем, что эти файлы не следует редактировать — они создаются и модифицируются редактором ресурсов.

8.3.4. Программная поддержка немодального диалогового окна

Демонстрационный проект так же, как и в предыдущем примере, содержит три класса (рис. 8.16—8.18): класс `MainThread`, производный от класса `CWinApp` библиотеки MFC, класс `FrameWnd`, производный от класса `CFrameWnd` библиотеки MFC, и класс `Dialog`, производный от класса `CDialog` библиотеки MFC.

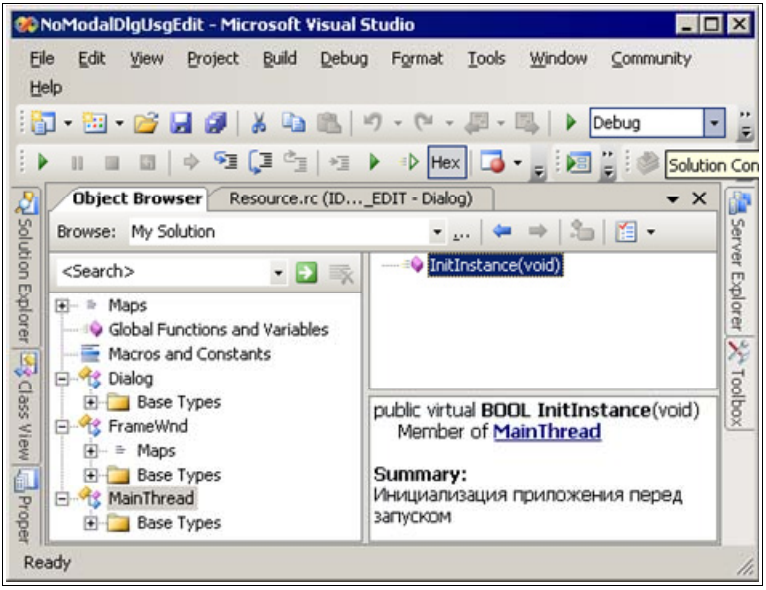


Рис. 8.16. Класс `MainThread` проекта NoModalDlgUsgEdit

Класс `FrameWnd` содержит таблицу сообщений, которая предусматривает создание диалогового окна для работы с элементом редактирования (см. приведенный ранее рис. 8.13).

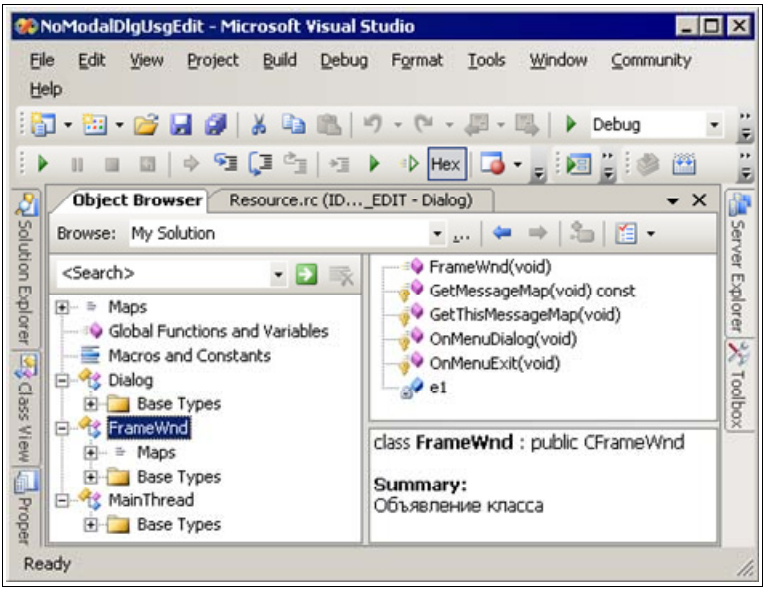


Рис. 8.17. Класс `FrameWnd` проекта NoModalDlgUsgEdit

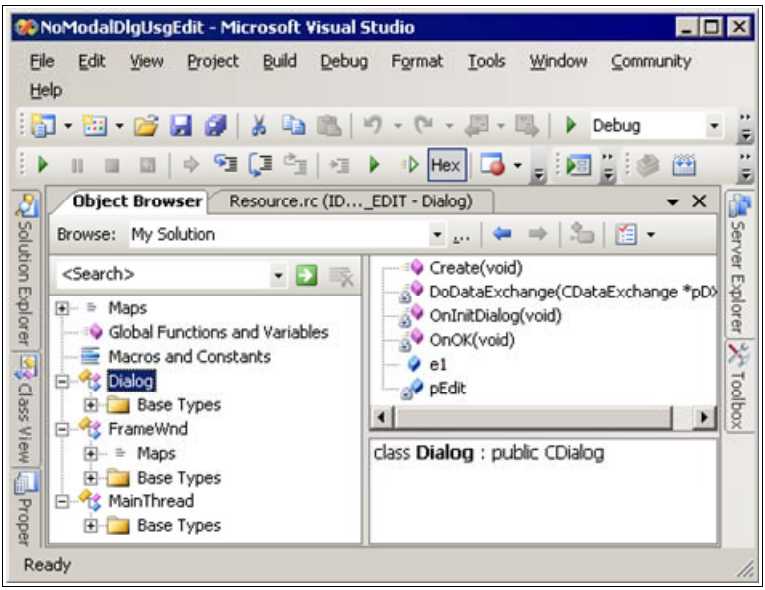


Рис. 8.18. Класс `Dialog` проекта NoModalDlgUsgEdit

```
// Фрагмент файла FrmWnd.hpp
    ...
    protected:

        // Прототипы функций, обрабатывающих сообщения
        // Обработка команды "Выход"
        afx_msg void OnMenuExit( void );
        // Обработка команды "Немодальный_диалог"
        afx_msg void OnMenuDialog( void );

        // Объявление таблицы сообщений
        DECLARE_MESSAGE_MAP( );

// Фрагмент файла FrameWnd.cpp
    ...
// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( FrameWnd, CFrameWnd )
    ON_COMMAND( ID_EXIT, OnMenuExit )
    ON_COMMAND( ID_DIALOG_EDIT, OnMenuDialog )
END_MESSAGE_MAP( )
// Конструктор умолчания (инициализация окна редактирования)
FrameWnd :: FrameWnd( void )
{
    // Для текста однострочного редактора
    e1 = "Заставка";
}
    ...
// Обработка команды "Немодальный_диалог"
afx_msg void FrameWnd :: OnMenuDialog( void )
{
    // Размещаем объект немодального диалогового окна в динамической
    //   памяти
    Dialog              *pDlg = new Dialog;
    ASSERT_VALID( pDlg );           // Обработка ошибки размещения

    // Передаем текст для элемента редактирования
    pDlg->e1 = e1;

    // Создаем немодальное окно диалога
    if( !pDlg->Create( ) )
    {
        TRACE0( "\n Ошибка 4 — немодальное диалоговое окно не "
                "создано \n" );
```

```
        exit( 4 );
    }

    // Перерисовка окна (формирует сообщение WM_PAINT)
    CWnd :: Invalidate( );

    return afx_msg void( );
}
```

При выполнении команды **Немодальный_диалог** вызывается метод `FrameWnd::OnMenuDialog`, который и создает диалоговое окно, показанное на рис. 8.13. Для создания немодального окна диалога в функции `OnMenuDialog` используется метод `Create` класса `CDialog`.

Класс `Dialog`, производный от класса `CDialog` библиотеки MFC, обеспечивает работу с диалоговым окном. В объявлении класса (см. листинг 8.8) определены члены класса: переменная `pEdit` — указатель на однострочный элемент редактирования, который доступен только в методах класса `Dialog`, и строковая переменная `e1` для текста элемента редактирования. В классе также имеются несколько методов, прототипы которых находятся в объявлении класса. К их числу относятся:

- прототип метода для создания немодального диалогового окна (`Create`);
- прототип переопределяемого виртуального метода `OnInitDialog`, выполняющего требуемую инициализацию окна диалога;
- прототип переопределяемого виртуального метода `OnOK`, обеспечивающего получение данных из диалогового окна;
- прототип метода `DoDataExchange`, обеспечивающего получение данных из окна диалога. Этот метод вызывается из метода `UpdateData`.

В реализации класса `Dialog` (см. листинг 8.9), прежде всего, содержится виртуальный метод для создания немодального диалогового окна:

```
// Создание немодального диалога !!! Внимание, важно
BOOL Dialog :: Create( void )
{
    return CDialog :: Create( IDD_DIALOG_EDIT );
}
```

Файл реализации класса `Dialog` содержит далее переопределение виртуального метода `OnInitDialog`, выполняющего требуемую инициализацию немодального окна диалога. В нем следует вызвать соответствующий метод базового класса, а затем выполнить необходимую инициализацию диалогового окна. В методе `OnInitDialog` для определения значений указателя на окно ре-

дактирования используется метод GetDlgItem, наследуемый из класса CWnd библиотеки MFC. Далее, используя метод SetLimitText, устанавливаем длину текста элемента редактирования и, в соответствии со значением члена класса — строковой переменной Dialog::e1, устанавливаем текст, отображаемый в элементе редактирования. Для этой цели используется метод SetWindowText.

Файл реализации класса Dialog содержит также переопределение виртуального метода OnOK, обеспечивающего получение данных из диалогового окна и их использование. С этой целью сначала вызывается одноименный метод базового класса, а затем, с помощью метода UpdateData, автоматически вызывается метод DoDataExchange. Теперь удвойте ваше внимание — важная информация! Именно метод DoDataExchange как раз и обеспечивает получение информации из однострочного окна редактирования при закрытии диалогового окна кнопкой **OK**:

```
// Получение данных из диалогового окна (метод вызывается автоматически
//   из метода UpdateData( ))
void Dialog :: DoDataExchange( CDataExchange *pDX )
{
    // Вызываем метод базового класса
    CDialog :: DoDataExchange( pDX );

    // Копируем в e1 текст из однострочного редактора
    DDX_Text( pDX, IDC_EDIT, e1 );

    return;
}
```

Примечание

Для получения информации из окна редактирования можно использовать удобную функцию DDX_Text. Первый аргумент в вызове этой функции представляет собой указатель на объект класса CDataExchange, с помощью которого устанавливается контекст обмена данными между объектами, задаваемыми вторым и третьим аргументами. Второй аргумент является идентификатором окна редактирования. Третий указывает приемник информации, получаемой из окна редактирования (в демонстрационном проекте это строковая переменная типа CString, являющаяся членом класса Dialog). В качестве третьего аргумента в вызове функции можно использовать также аргументы с типами BYTE, short, int, UINT, long, DWORD, float, double и др.

Совет

Для получения более подробной справки о функции DDX_Text запустите проект и поместите курсор в исходном коде на идентификатор функции DDX_Text (файл Dialog.cpp, метод DoDataExchange) и нажмите клавишу <F1>.

Вид главного окна демонстрационного проекта после задания текста в элементе редактирования показан на рис. 8.19, а окно сообщения, после закрытия диалогового окна кнопкой **OK** — на рис. 8.20.

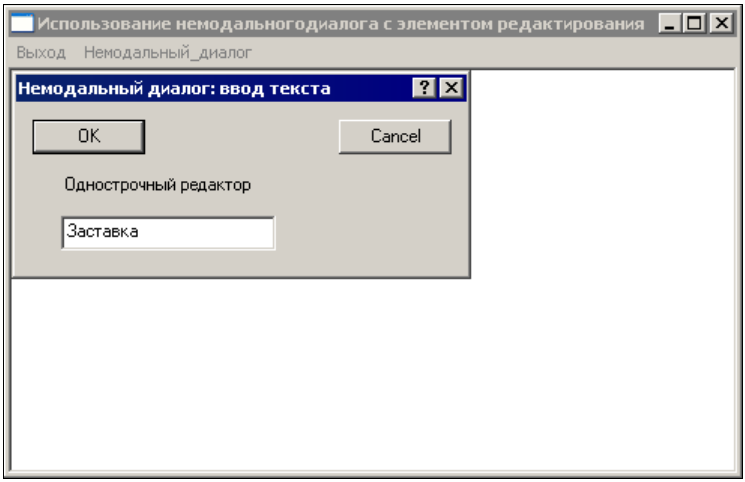


Рис. 8.19. Главное окно проекта NoModalDlgUsgEdit после задания текста в элементе редактирования

Рис. 8.20. Отображение информации, полученной из элемента редактирования

Замечание

Как указывалось ранее, управляющие элементы проще и удобнее применять в диалоговом окне, но их можно использовать и в любых других окнах, например, в главном окне приложения. Чтобы уяснить особенности применения управляющих элементов вне диалоговых окон и выполнить сравнительный анализ указанных вариантов, далее рассмотрим еще один демонстрационный проект. Заметим, что пример использования управляющего элемента (кнопки) в главном окне приложения уже рассмотрен (демонстрационный проект WndBtn_MFC из *главы 5*).

8.4. Демонстрационный пример с управляющими элементами — однострочным редактором и статическим текстом в главном окне

Главное окно рассматриваемого далее приложения на базе MFC содержит однострочное окно редактирования с поясняющим текстом и две кнопки — для считывания информации из окна редактирования и для завершения работы приложения (рис. 8.21).

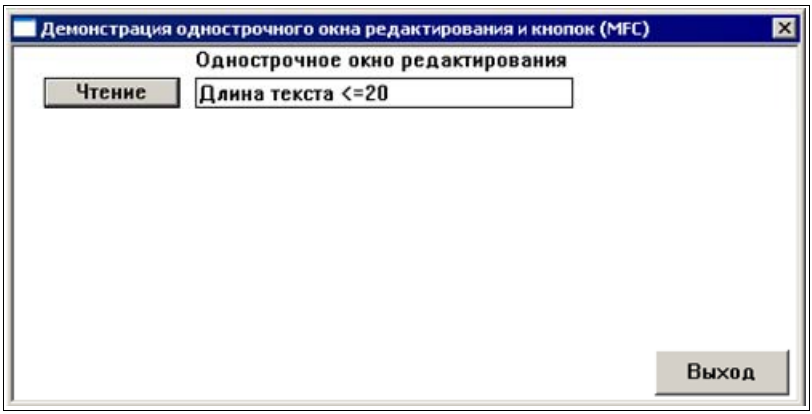


Рис. 8.21. Главное окно демонстрационного проекта WndEditBtn

Отличается это приложение тем, что для поддержки окна редактирования редактор ресурсов не может быть использован и работа однострочного редактора реализуется программно. Соответствующий демонстрационный проект находится на прилагаемом компакт-диске, в папке \Глава 08\WndEditBtn, а файловый состав проекта показан на рис. 8.22.

Совет

Копию проекта WndEditBtn для экспериментов создавайте самым простым способом — скопируйте папку проекта с компакт-диска. Файлы StdAfx.h, StdAfx.cpp, Main.cpp, MainThread.hpp в данном проекте являются типовыми и их текст приводился ранее, а файлы MainThread.cpp, FrameWnd.hpp и FrameWnd.cpp модифицированы в соответствии с листингами 8.10—8.12.

Рис. 8.22. Файловый состав демонстрационного проекта WndEditBtn

Листинг 8.10. Файл MainThread.cpp

```
/*
   Файл                : MainThread.cpp

   Проект              : демонстрация создания Windows-приложения на
                         основе библиотеки классов MFC с кнопками в
                         главном окне, однострочным редактором и
                         выводом текста

   Назначение          : реализация класса "MainThread",
                         производного от класса "CWinApp" библиотеки
                         классов MFC (инициализация приложения перед
                         запуском)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAFX.h"                  // Прекомпилируемый файл
#include "MainThread.hpp"            // Объявление класса MainThread
#include "FrameWnd.hpp"              // Объявление класса FrameWnd

// Переопределение виртуальной функции для инициализации приложения
//   перед запуском
BOOL MainThread :: InitInstance( void )
{
    // Указатель на фрейм окна
    FrameWnd            *pFrame;
```

```
    // Размещение фрейма окна в динамической памяти
    pFrame = new FrameWnd;
    ASSERT_VALID( pFrame );          // Обработка ошибки размещения

    // Создание окна приложения
    BOOL                rc = pFrame->Create(
        // Используется предопределенное OS WINDOWS имя оконного
        //   класса
        NULL,
        // Заголовок фрейма окна
        "Демонстрация однострочного окна редактирования и кнопок "
        "(MFC)",
        // Стиль окна
        WS_SIZEBOX | WS_POPUPWINDOW | WS_DLGFRAME,
        // Местоположение и размеры окна: 0 — левая сторона,
        //   0 — верх, 530 — правая сторона, 290 — низ окна
        CRect( 0, 0, 530, 290 ) );
    if( !rc )
    {
        TRACE0( "\n Ошибка 1. Фрейм окна не был создан \n" );
        exit( 1 );
    }

    // Создание кнопок
    pFrame->CreateChildControls( );
    // Отображение окна
    pFrame->ShowWindow( SW_SHOW );

    // Созданный фрейм делаем главным окном приложения
    this->m_pMainWnd = pFrame;

    return TRUE;
}
```

Листинг 8.11. Файл FrameWnd.hpp

```
/*
   Файл               : FrameWnd.hpp

   Проект             : демонстрация создания Windows-приложения на
                        основе библиотеки классов MFC с кнопками в
                        главном окне, однострочным редактором и
                        выводом текста
```

```
   Назначение          : объявление класса "FrameWnd", производного
                         от класса "CFrameWnd" библиотеки классов
                         MFC (обработка сообщений главного окна)

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _FrameWnd_hpp
    #define _FrameWnd_hpp

    // Объявление класса
    class FrameWnd : public CFrameWnd
    {
        // Данные

    private:

        // Указатель на умалчиваемую кнопку считывания из окна
        //   редактирования
        CButton         *m_pBtnRead;
        // Указатель на кнопку завершения работы
        CButton         *m_pBtnExit;
        // Указатель на однострочный редактор
        CEdit           *m_pEdit;

        // Методы

    public:

        // Конструктор умолчания (инициализация указателей на
        //   кнопки и редактор)
        FrameWnd( void );
        // Деструктор (освобождает память, занятую кнопками и
        //   однострочным редактором)
        ~FrameWnd( void );

        // Создание дочерних управляющих элементов (перегружаемый
        //   виртуальный метод)
        virtual void CreateChildControls( void );

    protected:
```

```
        // Так нужно записывать прототипы функций, обрабатывающих
        //   сообщения
        // Перерисовка окна
        afx_msg void OnPaint( void );
        // Обработчик кнопки "Чтение"
        afx_msg void OnBtnReadClick( void );
        // Обработчик кнопки "Выход"
        afx_msg void OnBtnExitClick( void );

        // Объявление таблицы сообщений
        DECLARE_MESSAGE_MAP( );
    };

#endif
```

Листинг 8.12. Файл FrameWnd.cpp

```
/*
   Файл                : FrameWnd.cpp

   Проект              : демонстрация создания Windows-приложения на
                         основе библиотеки классов MFC с кнопками в
                         главном окне, однострочным редактором и
                         выводом текста

   Назначение          : реализация класса "FrameWnd", производного
                         от класса "CFrameWnd" библиотеки классов
                         MFC (обработка сообщений главного окна)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAFX.h"                 // Прекомпилируемый файл
#include "FrameWnd.hpp"             // Объявление класса FrameWnd

// Идентификаторы управляющих элементов
#define IDC_BTNREAD 100             // Умалчиваемая кнопка "Чтение"
#define IDC_BTNEXIT 101             // Кнопка "Выход"
#define IDC_ECONE   102             // Однострочный редактор

// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( FrameWnd, CFrameWnd )
    ON_COMMAND( IDC_BTNREAD, OnBtnReadClick )
```

```
    ON_COMMAND( IDC_BTNEXIT, OnBtnExitClick )
    ON_WM_PAINT( )
END_MESSAGE_MAP( )

// Конструктор умолчания (инициализация указателей на кнопки и
//   редактор)
FrameWnd :: FrameWnd( void )
{
    m_pBtnRead = 0;
    m_pBtnExit = 0;
    m_pEdit    = 0;
}

// Деструктор (освобождает память, занятую кнопками и однострочным
//   редактором)
FrameWnd :: ~FrameWnd( void )
{
    if ( m_pBtnRead )
    {
        delete m_pBtnRead;
        m_pBtnRead = 0;
    }
    if ( m_pBtnExit )
    {
        delete m_pBtnExit;
        m_pBtnExit = 0;
    }
    if ( m_pEdit )
    {
        delete m_pEdit;
        m_pEdit = 0;
    }
}

// Создание дочерних управляющих элементов (перегружаемый
//   виртуальный метод)
void FrameWnd :: CreateChildControls( void )
{
    // Размещаем в динамической памяти и создаем однострочный
    //   элемент редактирования
    m_pEdit = new CEdit;
    ASSERT_VALID( m_pEdit );        // Обработка ошибки размещения
    BOOL                rc = m_pEdit->Create( WS_VISIBLE |
```

```
    WS_CHILD | ES_LEFT | WS_BORDER, CRect( 120, 20, 370, 40 ),
    this, IDC_ECONE );
if( !rc )
{
    TRACE0( "\n Ошибка 2. Элемент редактирования не был"
            " создан \n" );
    exit( 2 );
}
/*
   Первый аргумент в вызове метода Create( ) задает стиль окна
редактирования. Особые стили редактирования можно сочетать с
общими стилями окон, доступными объектам класса CWnd.
   Второй аргумент задает местоположение и размеры окна
редактирования, третий указывает на родительское окна, а
четвертый — является идентификатором окна редактирования
*/
m_pEdit->LimitText( 20 );       // Ограничение длины текста
// Начальная инициализация окна редактирования
m_pEdit->SetWindowText( "Длина текста <=20" );

// Размещаем в динамической памяти и создаем кнопку "Чтение"
//   (выбрана по умолчанию)
m_pBtnRead = new CButton;
ASSERT_VALID( m_pBtnRead );
rc = m_pBtnRead->Create( "Чтение", WS_VISIBLE | WS_CHILD |
    BS_DEFPUSHBUTTON, CRect( 20, 20, 110, 40 ), this,
    IDC_BTNREAD );
if( !rc )
{
    TRACE0( "\n Ошибка 3. Кнопка \"Чтение\" не была"
            " создана \n" );
    exit( 3 );
}

// Размещаем в динамической памяти и создаем кнопку "Выход"
m_pBtnExit = new CButton;
ASSERT_VALID( m_pBtnExit );     // Обработка ошибки размещения
rc = m_pBtnExit->Create( "Выход", WS_VISIBLE | WS_CHILD
    | BS_PUSHBUTTON, CRect( 424, 200, 513, 230 ), this,
    IDC_BTNEXIT );
if( !rc )
{
    TRACE0( "\n Ошибка 4. Кнопка \"Выход\" не была"
            " создана \n" );
```

```
        exit( 4 );
    }

    return;
}

// Обработчик кнопки "Чтение"
afx_msg void FrameWnd :: OnBtnReadClick( void )
{
    // Звуковой сигнал
    BOOL                rc = MessageBeep( -1 );
    if( !rc )
    {
        TRACE0( "\n Ошибка 5. Неверное завершение MessageBeep \n" );
        exit( 5 );
    }

    // Чтение и визуализация текста из однострочного редактора
    CString             string;        // Для прочитанного текста
    m_pEdit->GetWindowText( string );
    AfxMessageBox( string );

    return afx_msg void( );
}

// Обработчик кнопки "Выход"
afx_msg void FrameWnd :: OnBtnExitClick( void )
{
    BOOL                rc = MessageBeep( -1 );
    if( !rc )
    {
        TRACE0( "\n Ошибка 6. Неверное завершение MessageBeep \n" );
        exit( 6 );
    }
    rc = DestroyWindow( );
    if( !rc )
    {
        TRACE0( "\n Ошибка 7. Окно не было разрушено \n" );
        exit( 7 );
    }

    return afx_msg void( );
}
```

```
// Обработчик сообщения WM_PAINT: демонстрирует вывод поясняющего
//   текста
afx_msg void FrameWnd :: OnPaint( void )
{
    // Создаем объект для рисования и вывода
    CPaintDC            PaintDC( this );

    // Вывод заданного текста (третий аргумент) в определенное
    //   место окна (первый и второй аргументы)
    PaintDC.TextOut( 120, 0, "Однострочное окно редактирования" );

    return afx_msg void( );
}
```

Замечание

Надеемся, что изучение исходного кода приведенного демонстрационного проекта не вызовет у вас затруднений. Код хорошо прокомментирован.

Совет

Сравните два последних демонстрационных проекта. Это очень важно для уяснения особенностей использования управляющих элементов вне диалоговых окон.

8.5. Вопросы и упражнения для самопроверки

Для самопроверки ответьте на следующие вопросы:

1. На что требуется обратить первостепенное внимание в этой главе?
2. Что представляют собой диалоговые окна и управляющие элементы?
3. Укажите типы диалоговых окон и дайте им характеристику.
4. Укажите разновидности диалоговых окон и дайте им характеристику.
5. Перечислите типы стандартных диалоговых окон Windows.
6. Как получить самые точные и полные сведения о составе и иерархии классов библиотеки MFC? Каково место стандартных классов диалоговых окон в иерархии классов библиотеки MFC?
7. Что отличает диалоговые окна от других?
8. Укажите назначение диалоговых окон.
9. Какие сообщения обрабатывает диалоговое окно?

10. Когда нужно обрабатывать сообщение `WM_INITDIALOG`?
11. Укажите назначение класса `CDialog` библиотеки MFC.
12. Как создать диалоговое окно?
13. Перечислите стандартные и виртуальные методы класса `CDialog` библиотеки классов MFC и укажите их назначение.
14. Какие элементы управления можно использовать в диалоговом окне?
15. Назовите обязательные кнопки диалогового окна, укажите способы закрытия окна.
16. Укажите особенности работы с окнами немодального диалогового окна.
17. Как следует добавлять в проект класс диалогового окна, его таблицу сообщений и обработчики сообщений управляющих элементов?
18. Как создать и настроить шаблон ресурсов диалогового окна?
19. Как в диалоговом окне разместить управляющие элементы? Укажите особенность радиокнопок. Как задать свойства радиокнопок?
20. Где хранится шаблон ресурсов диалогового окна?
21. В чем состоит программная поддержка модального диалогового окна в классе `FrameWnd` демонстрационного проекта UsgModalDlg?
22. Какой класс библиотеки MFC поддерживает работу окон редактирования?
23. Что представляет собой окно редактирования?
24. Чем отличаются однострочные и многострочные окна редактирования?
25. Перечислите стили окна редактирования. Можно ли их комбинировать с другими стилями?
26. Охарактеризуйте основные методы класса `CEdit`.
27. Как поместить или извлечь текст из окна редактирования?

Ответы на вопросы можно проверить — они приведены в *разд. П1.8 приложения 1*.

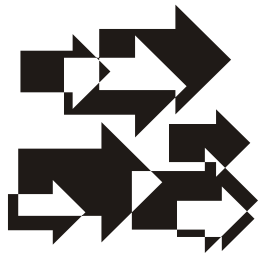
При экспериментах с созданной копией демонстрационного проекта UsgModalDlg выполните следующие упражнения:

1. Уберите комментарии в определении класса `Dialog` и проверьте работоспособность проекта.
2. Попробуйте менять свойства радиокнопок и посмотрите, на что это влияет.
3. Попробуйте менять размеры диалогового окна, размеры и местоположение его управляющих элементов.

При экспериментах с созданными копиями демонстрационных проектов NoModalDlgUsgEdit и WndEditBtn выполните следующие упражнения:

1. Попробуйте менять свойства окна редактирования и статического текста и посмотрите, на что это влияет.
2. Попробуйте менять размеры диалогового окна, размеры и местоположение его управляющих элементов.
3. Выполните сопоставление демонстрационных проектов NoModalDlgUsgEdit и WndEditBtn, обоснуйте, какой из них проще и удобнее.
4. Попробуйте ввести длинный текст в поле однострочного элемента редактирования.

ГЛАВА 9



Элементы управления: кнопки, элемент группировки, спин и элементы прокрутки [5]

Новым в этой главе является использование редактора ресурсов для создания и настройки в диалоговых окнах таких широко применяемых элементов управления, как элементы группировки, кнопки (отмечаемые кнопки, кнопки с растровым изображением), спин и элементы прокрутки (ползунки и полосы прокрутки). Уделите им первоочередное внимание.

9.1. Кнопки. Классы *CButton* и *CBitMapButton* библиотеки MFC

Кнопка — это прямоугольное окно обычно небольшого размера и имеющее название, в котором указано ее назначение. Кнопки являются широко используемыми элементами управления Windows-приложений, что обусловлено их разнообразием. Существует пять основных типов кнопок, определенных в классе `CButton` библиотеки MFC:

- обыкновенные кнопки (см. примеры в *главах 5 и 8*);
- специализированные кнопки **OK** и **Cancel** (Отмена), используемые в диалоговых окнах *(см. главу 8)*;
- радиокнопки *(см. главу 8)*;
- отмечаемые кнопки.

Примечание

Кнопки, как впрочем и другие элементы управления, можно визуально объединять с помощью блока группировки (Group Box). Блок группировки представляет собой видимую рамку для других элементов управления (не обязательно кнопок).

Библиотека MFC обеспечивает все функциональные возможности кнопок с помощью классов CButton и CBitMapButton. Их положение в иерархии классов библиотеки MFC можно, как уже указывалось ранее, получить в справочной системе MSDN, интегрирующейся в Microsoft Visual Studio. В соответствии с иерархией классов библиотеки MFC класс CButton является производным от класса CWnd, а это означает, что кнопка является окном, которое наследует возможности объекта CWnd.

В библиотеке MFC также есть класс CBitMapButton для кнопок с растровыми изображениями, производный от класса CButton и наследующий все его возможности. Кнопка с растровым изображением — это стандартная кнопка, которая вместо поясняющего текста содержит растровый рисунок (пиктограмму).

Кнопки могут применяться несколькими способами.

Для ввода команды пользователя используется *нажимаемая* кнопка.

Если существует взаимоисключающий набор опций, то применяется сгруппированный набор радиокнопок.

Если несколько опций не являются взаимоисключающими (пользователь одновременно может выбрать несколько возможностей), то используется набор отмечаемых кнопок (флажков).

Чтобы создать видимую рамку вокруг взаимосвязанной группы элементов управления (например, группы радиокнопок или отмечаемых кнопок), используется блок группировки.

Два класса кнопок — CButton и CBitMapButton — имеют простой набор методов, что упрощает работу с ними. Как было показано ранее, кнопки могут быть созданы как подчиненные элементы управления любого окна непосредственно в программном коде, но, как правило, они определяются с помощью редактора ресурсов в шаблоне ресурсов диалогового окна.

Подобно другим окнам, кнопки могут сочетать в себе ряд элементарных стилей окон CWnd (см. листинг 2.3 в *главе 2*). Кроме общих стилей окон, для кнопок можно сочетать элементарные стили кнопок, перечисленные в листинге 5.1 *главы 5*.

Сообщения от кнопок (объектов CButton и CBitMapButton) представлены в табл. 9.1.

Таблица 9.1. *Элементы таблицы сообщений от кнопок*

Элемент таблицы сообщений	Значение
ON_BN_CLICKED или ON_COMMAND	При нажатии на кнопку

Таблица 9.1 *(окончание)*

Элемент таблицы сообщений	Значение
ON_BN_DBLCLICKED	При двойном щелчке по кнопке

9.1.1. Группировка управляющих элементов

Замечание

В рассматриваемом далее демонстрационном проекте UsgGroupBox, в модальном диалоговом окне, выполняется группировка радиокнопок. Особенности работы с радиокнопками рассмотрены в *главе 8* (демонстрационный проект UsgModalDlg).

Демонстрационный проект, иллюстрирующий особенности группировки управляющих элементов, находится на прилагаемом компакт-диске, в папке \Глава 09\UsgGroupBox, а файловый состав проекта показан на рис. 9.1.

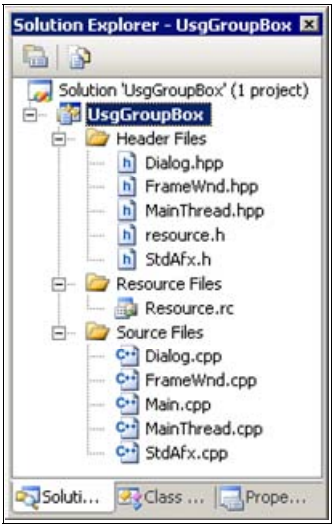


Рис. 9.1. Файловый состав проекта UsgGroupBox

Совет

Копию проекта UsgGroupBox для экспериментов создавайте копированием папки проекта с компакт-диска. Файлы StdAfx.h, StdAfx.cpp, Main.cpp, MainThread.hpp, MainThread.cpp в данном проекте являются стандартными, а файлы FrameWnd.hpp, FrameWnd.cpp, Dialog.hpp и Dialog.cpp модифицированы в соответствии с листингами 9.1—9.4. Два оставшихся файла — resource.h и Resourse.rc — созданы редактором ресурсов и поддерживаются им. Ресурсы проекта представлены на рис. 9.2.

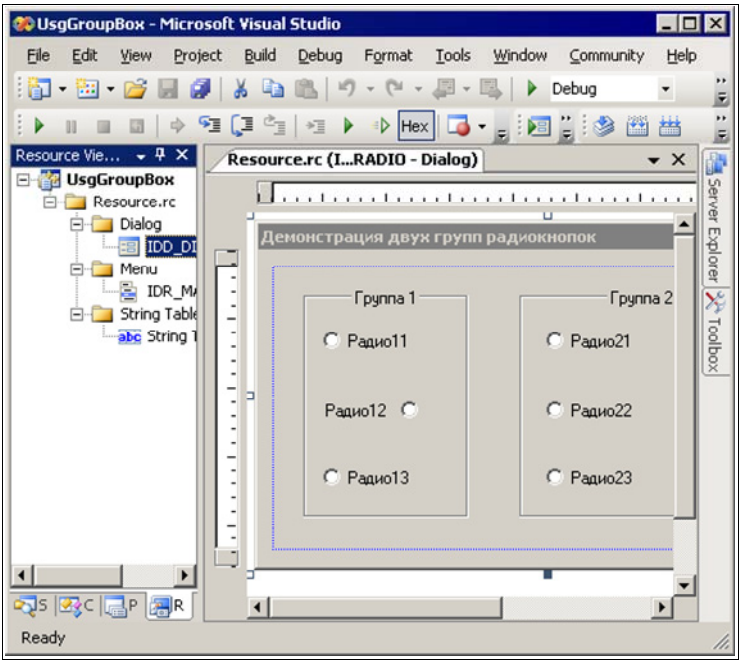


Рис. 9.2. Ресурсы проекта UsgGroupBox

Листинг 9.1. Файл FrameWnd.hpp

```
/*
   Файл                 : FrameWnd.hpp

   Проект               : демонстрация Windows-приложения на основе
                          библиотеки классов MFC с модальным
                          диалоговым окном, использующим GroupBox

   Назначение           : объявление класса "FrameWnd", производного
                          от класса "CFrameWnd" библиотеки классов
                          MFC (обработка сообщений главного окна)

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _FrameWnd_hpp
    #define _FrameWnd_hpp
```

```
    // Объявление класса
    class FrameWnd : public CFrameWnd
    {
        // Данные

    private:

        // Переменные, связанные с GroupBox
        int             m_Radio1;   // Радиокнопка 1
        int             m_Radio2;   // Радиокнопка 2

        // Методы

    public:

        // Конструктор умолчания (инициализация радиокнопок)
        FrameWnd( void );

    protected:

        // Прототипы функций, обрабатывающих сообщения
        // Обработка команды Файл | Выход
        afx_msg void OnMenuFileExit( void );
        // Обработка команды Группы_радио_кнопок | Диалог
        afx_msg void OnMenuRadioButtons( void );

        // Объявление таблицы сообщений
        DECLARE_MESSAGE_MAP( );
    };

#endif
```

Листинг 9.2. Файл FrameWnd.cpp

```
/*
   Файл               : FrameWnd.cpp

   Проект             : демонстрация Windows-приложения на основе
                        библиотеки классов MFC с модальным
                        диалоговым окном, использующим GroupBox

   Назначение         : реализация класса "FrameWnd", производного
                        от класса "CFrameWnd" библиотеки классов
                        MFC (обработка сообщений главного окна)
```

```
   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAfx.h"                  // Прекомпилируемый файл
#include "FrameWnd.hpp"              // Объявление класса FrameWnd
#include "Dialog.hpp"                // Объявление класса Dialog
// Идентификаторы ресурсов (поддержка редактором ресурсов)
#include "resource.h"

// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( FrameWnd, CFrameWnd )
    ON_COMMAND( ID_FILE_EXIT, OnMenuFileExit )
    ON_COMMAND( ID_RADIO_BUTTONS, OnMenuRadioButtons )
END_MESSAGE_MAP( )

// Конструктор умолчания (инициализация радиокнопок)
FrameWnd :: FrameWnd( void )
{
    m_Radio1 = 1;
    m_Radio2 = 5;
}

// Обработка команды Файл | Выход
afx_msg void FrameWnd :: OnMenuFileExit( void )
{
    BOOL               rc = MessageBeep( -1 );
    if( !rc )
    {
        TRACE0( "\n Ошибка 2. Неверное завершение MessageBeep \n" );
        exit( 2 );
    }
    rc = DestroyWindow( );
    if( !rc )
    {
        TRACE0( "\n Ошибка 3. Окно не было разрушено \n" );
        exit( 3 );
    }

    return afx_msg void( );
}

// Обработка команды Группы_радио_кнопок | Диалог
afx_msg void FrameWnd :: OnMenuRadioButtons( void )
```

```
{
    // Создаем объект диалога — автоматически вызывается
    //   конструктор
    Dialog              Dlg( this );

    // Передаем номера радиокнопок из класса FrameWnd в класс
    //   Dialog
    Dlg.m_Radio1 = m_Radio1;
    Dlg.m_Radio2 = m_Radio2;

    // Создается модальное диалоговое окно — оно будет
    //   автоматически закрыто после нажатия OK или Cancel
    int                 DlgResult = static_cast<int>
                                    ( Dlg.DoModal( ) );
    if ( DlgResult == IDOK )
    {                               // Выход из диалога по OK
        // Передаем номера радиокнопок из класса Dialog в класс
        //   FrameWnd
        m_Radio1 = Dlg.m_Radio1;
        m_Radio2 = Dlg.m_Radio2;
    }

    return afx_msg void( );
}
```

Листинг 9.3. Файл Dialog.hpp

```
/*
   Файл               : Dialog.hpp

   Проект             : демонстрация Windows-приложения на основе
                        библиотеки классов MFC с модальным
                        диалоговым окном, использующим GroupBox

   Назначение         : объявление класса "Dialog", производного
                        от класса "CDialod" библиотеки классов
                        MFC (обработка сообщений окна диалога)

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
```

```
#ifndef _Dialog_hpp
    #define _Dialog_hpp

    // Объявление класса
    class Dialog : public CDialog
    {
        // Данные

    private:

        // Указатель на радиокнопку 11 (группа 1, кнопка 1)
        CButton         *m_pRadio11;
        // Указатель на радиокнопку 12 (группа 1, кнопка 2)
        CButton         *m_pRadio12;
        // Указатель на радиокнопку 13 (группа 1, кнопка 3)
        CButton         *m_pRadio13;
        // Указатель на радиокнопку 21 (группа 2, кнопка 1)
        CButton         *m_pRadio21;
        // Указатель на радиокнопку 22 (группа 2, кнопка 2)
        CButton         *m_pRadio22;
        // Указатель на радиокнопку 23 (группа 2, кнопка 3)
        CButton         *m_pRadio23;

    public:

        int             m_Radio1;   // Радиокнопка 1
        int             m_Radio2;   // Радиокнопка 2

    public:

        // Конструктор — передает идентификатор диалогового окна в
        //   базовый класс
        Dialog( CWnd *pParent = 0 );

    protected:

        // Перегружаем виртуальный метод для инициализации
        virtual BOOL OnInitDialog( void );

        // Обработчики радиокнопок
        afx_msg void OnRadio11Click( void );
        afx_msg void OnRadio12Click( void );
```

```
        afx_msg void OnRadio13Click( void );
        afx_msg void OnRadio21Click( void );
        afx_msg void OnRadio22Click( void );
        afx_msg void OnRadio23Click( void );

        // Объявление таблицы сообщений
        DECLARE_MESSAGE_MAP( );
    };

#endif
```

Листинг 9.4. Файл Dialog.cpp

```
/*
   Файл                : Dialog.cpp

   Проект              : демонстрация Windows-приложения на основе
                         библиотеки классов MFC с модальным
                         диалоговым окном, использующим GroupBox

   Назначение          : реализация класса "Dialog", производного
                         от класса "CDialod" библиотеки классов
                         MFC (обработка сообщений)

   Microsoft Visual Studio C++ .NET 2005
*/


#include "StdAfx.h"                  // Прекомпилируемый файл
#include "Dialog.hpp"                // Объявление класса Dialog
// Идентификаторы ресурсов (поддержка редактором ресурсов)
#include "resource.h"

// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( Dialog, CDialog )
    ON_BN_CLICKED( IDC_RADIO11, OnRadio11Click )
    ON_BN_CLICKED( IDC_RADIO12, OnRadio12Click )
    ON_BN_CLICKED( IDC_RADIO13, OnRadio13Click )
    ON_BN_CLICKED( IDC_RADIO21, OnRadio21Click )
    ON_BN_CLICKED( IDC_RADIO22, OnRadio22Click )
    ON_BN_CLICKED( IDC_RADIO23, OnRadio23Click )
END_MESSAGE_MAP( )
```

```
// Конструктор
Dialog :: Dialog(
    CWnd                *pParent )  // Указатель на родительское
                                    //   окно (NULL — родительским
                                    //   является главное окно)
    : CDialog( IDD_DIALOG_RADIO, pParent )
    { }

// Инициализация диалогового окна
BOOL Dialog :: OnInitDialog( void )
{
    // Вызываем метод базового класса
    CDialog :: OnInitDialog( );

    // Получаем указатели на радиокнопки
    m_pRadio11 = static_cast< CButton * >
        ( GetDlgItem( IDC_RADIO11 ) );
    m_pRadio12 = static_cast< CButton * >
        ( GetDlgItem( IDC_RADIO12 ) );
    m_pRadio13 = static_cast< CButton * >
        ( GetDlgItem( IDC_RADIO13 ) );
    m_pRadio21 = static_cast< CButton * >
        ( GetDlgItem( IDC_RADIO21 ) );
    m_pRadio22 = static_cast< CButton * >
        ( GetDlgItem( IDC_RADIO22 ) );
    m_pRadio23 = static_cast< CButton * >
        ( GetDlgItem( IDC_RADIO23 ) );

    // Устанавливаем активную радиокнопку в группе 1
    switch ( m_Radio1 )
    {
    case 1:
        m_pRadio11->SetCheck( 1 );
        break;

    case 2:
        m_pRadio12->SetCheck( 1 );
        break;

    case 3:
        m_pRadio13->SetCheck( 1 );
        break;
    }
```

```
    // Устанавливаем активную радиокнопку в группе 2
    switch ( m_Radio2 )
    {
    case 4:
        m_pRadio21->SetCheck( 1 );
        break;

    case 5:
        m_pRadio22->SetCheck( 1 );
        break;

    case 6:
        m_pRadio23->SetCheck( 1 );
        break;
    }

    return TRUE;
}

// Обработчики радиокнопок
// Обработчик радиокнопки 11
void Dialog :: OnRadio11Click( void )
{
    m_Radio1 = 1;

    return;
}

// Обработчик радиокнопки 12
void Dialog :: OnRadio12Click( void )
{
    m_Radio1 = 2;

    return;
}

// Обработчик радиокнопки 13
void Dialog :: OnRadio13Click( void )
{
    m_Radio1 = 3;

    return;
}
```

```
// Обработчик радиокнопки 21
void Dialog :: OnRadio21Click( void )
{
    m_Radio2 = 4;

    return;
}

// Обработчик радиокнопки 22
void Dialog :: OnRadio22Click( void )
{
    m_Radio2 = 5;

    return;
}

// Обработчик радиокнопки 23
void Dialog :: OnRadio23Click( void )
{
    m_Radio2 = 6;

    return;
}
```

Главное окно приложения представлено на рис. 9.3, а модальное диалоговое окно с группами радиокнопок показано на рис. 9.4.

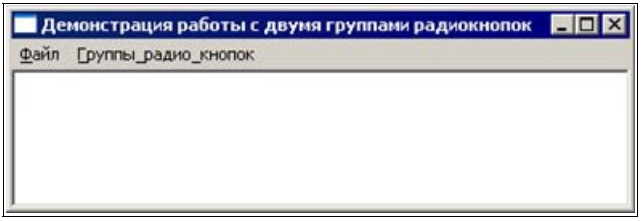


Рис. 9.3. Главное окно проекта UsgGroupBox

Приведенный исходный код достаточно понятен после изучения предшествующих демонстрационных проектов. Поэтому далее ограничимся лишь рассмотрением шаблона ресурсов диалогового окна в части создания и конфигурирования элементов группировки.

Для размещения в диалоговом окне элементов группировки можно воспользоваться панелью управляющих элементов **Toolbox** (см. рис. 8.2). Выберите

элемент группировки (Group Box) на панели **Toolbox** и перетащите его в нужное место диалогового окна. Размеры элемента группировки можно легко изменить с помощью мыши. Для демонстрационного примера размеры элементов группировки имеют вид, показанный на рис. 9.2.

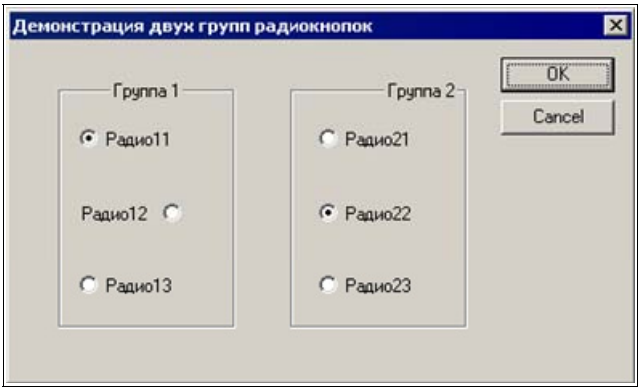


Рис. 9.4. Модальное диалоговое окно с группами радиокнопок проекта UsgGroupBox

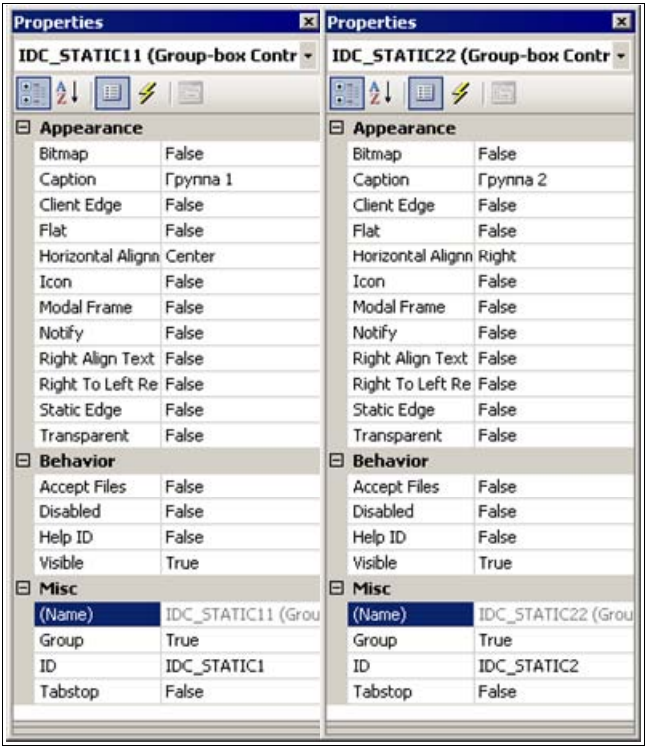


Рис. 9.5. Свойства элементов группировки проекта UsgGroupBox

Для настройки свойств элемента группировки достаточно вызвать его контекстное меню и выбрать пункт **Properties**. В появившейся одноименной вкладке задайте свойства элементов группировки (рис. 9.5).

Совет

Обратите внимание на такие свойства элемента группирования, как Caption (текст, отображаемый как название элемента группировки), Horizontal Alignment (размещение текста слева, справа или по центру), Disabled (отображение рамки и текста серым или темным цветом), Group (включение или отключение группирования), ID (идентификатор элемента группирования). Используя копию проекта UsgGroupBox, поэкспериментируйте, изменяя значения свойств элементов группирования.

9.1.2. Отмечаемые кнопки (флажки)

Как отмечалось ранее, одной из разновидностей кнопок являются отмечаемые кнопки (флажки). Они используются для задания значений нескольких опций, которые, в отличие от радиокнопок, не являются взаимоисключающими, т. е. пользователь может одновременно выбрать несколько вариантов.

В зависимости от значения свойства Tri-state из группы свойств Behavior, определяющих поведение кнопки, различают *стандартные отмечаемые кнопки* (флажки) с двумя состояниями (в этом случае значение Tri-state равно False) — включенным, выключенным, и *отмечаемые кнопки с тремя состояниями* — включенным, выключенным или неопределенным, в котором флажок выглядит "посеревшим" (значение Tri-state равно True).

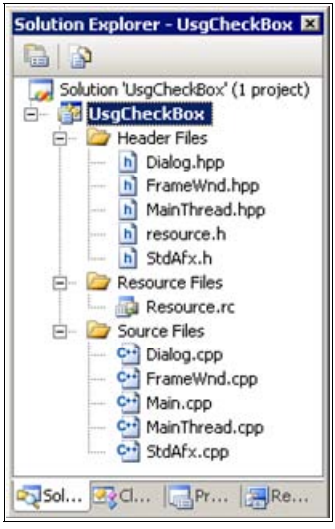


Рис. 9.6. Файловый состав проекта UsgCheckBox

Демонстрационный проект, иллюстрирующий работу с отмечаемыми кнопками, находится на прилагаемом компакт-диске, в папке \Глава 09\UsgCheckBox, а файловый состав проекта показан на рис. 9.6.

Совет

Копию проекта UsgCheckBox для экспериментов создавайте копированием папки проекта с компакт-диска. Файлы StdAfx.h, StdAfx.cpp, Main.cpp, MainThread.hpp, MainThread.cpp в данном проекте являются стандартными. Файлы FrameWnd.hpp и FrameWnd.cpp с точностью до обозначений и комментариев совпадают с соответствующими файлами из ранее рассмотренных проектов, а тексты файлов Dialog.hpp и Dialog.cpp представлены в листингах 9.5—9.6. Два оставшихся файла — resource.h и Resourse.rc — созданы редактором ресурсов и поддерживаются им. Ресурсы проекта представлены на рис. 9.7.

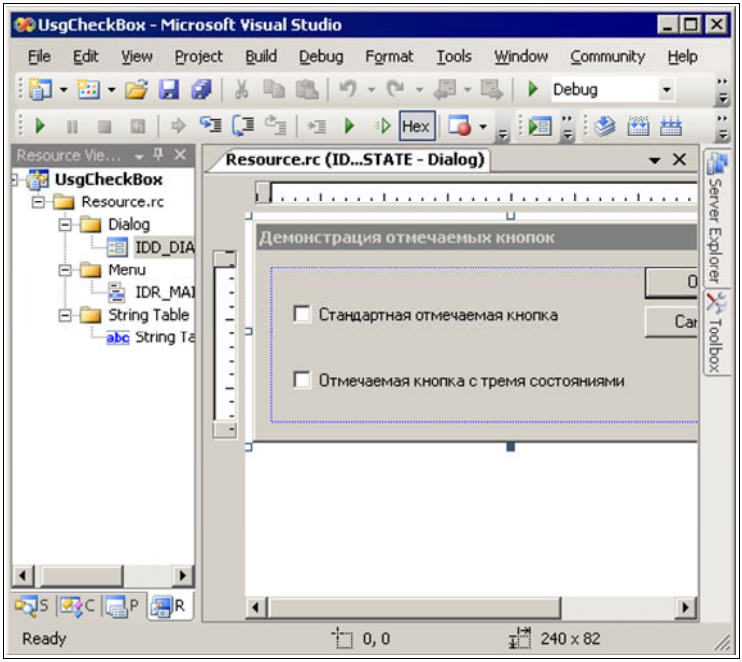


Рис. 9.7. Ресурсы проекта UsgCheckBox

Листинг 9.5. Файл Dialog.hpp

```
/*
   Файл              : Dialog.hpp

   Проект            : демонстрация Windows-приложения на основе
                       библиотеки классов MFC с модальным
                       диалоговым окном, использующим CheckBox
```

```
   Назначение         : объявление класса "Dialog", производного
                        от класса "CDialod" библиотеки классов
                        MFC (обработка сообщений)

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _Dialog_hpp
    #define _Dialog_hpp

    // Объявление класса
    class Dialog : public CDialog
    {
        // Данные

    private:

        // Указатель на CheckBox с двумя состояниями
        CButton         *m_pStdChk1;
        // Указатель на CheckBox с тремя состояниями
        CButton         *m_p3StChk2;

    public:

        // Состояние стандартного CheckBox
        bool            m_bStdChk1;
        // Состояние нестандартного CheckBox
        int             m_3StChk2;

        // Методы

    public:

        // Конструктор — передает идентификатор диалогового окна в
        //   базовый класс
        Dialog( CWnd *pParent = 0 );

    protected:

        // Перегружаем виртуальный метод для инициализации
        virtual BOOL OnInitDialog( void );
```

```
        // Обработчики CheckBox
        afx_msg void OnBtn1Click( void );
        afx_msg void OnBtn2Click( void );

        // Объявление таблицы сообщений
        DECLARE_MESSAGE_MAP( );
    };

#endif
```

Листинг 9.6. Файл Dialog.cpp

```
/*
   Файл               : Dialog.cpp

   Проект             : демонстрация Windows-приложения на основе
                        библиотеки классов MFC с модальным
                        диалоговым окном, использующим CheckBox

   Назначение         : реализация класса "Dialog", производного
                        от класса "CDialod" библиотеки классов
                        MFC (обработка сообщений окна)

   Microsoft Visual Studio C++ .NET 2005
*/


#include "StdAfx.h"                 // Прекомпилируемый файл
#include "Dialog.hpp"               // Объявление класса Dialog
// Идентификаторы ресурсов (поддержка редактором ресурсов)
#include "resource.h"

// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( Dialog, CDialog )
    ON_BN_CLICKED( IDC_STDCHK1, OnBtn1Click )
    ON_BN_CLICKED( IDC_3STCHK2, OnBtn2Click )
END_MESSAGE_MAP( )

// Конструктор
Dialog :: Dialog(
    CWnd                *pParent )  // Указатель на родительское
                                    //   окно (NULL — родительским
                                    //   является главное окно)
```

```
    : CDialog( IDD_DIALOG_3STATE, pParent )
    { }

// Инициализация диалогового окна
BOOL Dialog :: OnInitDialog( void )
{
    // Вызываем метод базового класса
    CDialog :: OnInitDialog( );

    // Получаем указатели на CheckBox
    m_pStdChk1 = static_cast< CButton * >
                          ( GetDlgItem( IDC_STDCHK1 ) );
    m_p3StChk2 = static_cast< CButton * >
                          ( GetDlgItem( IDC_3STCHK2 ) );

    // Устанавливаем состояние обычного CheckBox
    if ( m_bStdChk1 )
    {
        m_pStdChk1->SetCheck( 1 );
    }

    // Устанавливаем состояние необычного CheckBox
    switch ( m_3StChk2 )
    {
    case 1:
        m_p3StChk2->SetCheck( 1 );
        break;

    case 2:
        m_p3StChk2->SetCheck( 2 );
        break;
    }

    return TRUE;
}

// Обработчики CheckBox
// Обработчик обычного CheckBox
void Dialog :: OnBtn1Click( void )
{
    if ( m_pStdChk1->GetCheck( ) )
    {
        m_bStdChk1 = true;
    }
```

```
    else
    {
        m_bStdChk1 = false;
    }

    return;
}

// Обработчик CheckBox с тремя состояниями
void Dialog :: OnBtn2Click( void )
{
    m_3StChk2  = m_p3StChk2->GetCheck( );

    return;
}
```

Главное окно приложения представлено на рис. 9.8, а модальное диалоговое окно с отмечаемыми кнопками показано на рис. 9.9.

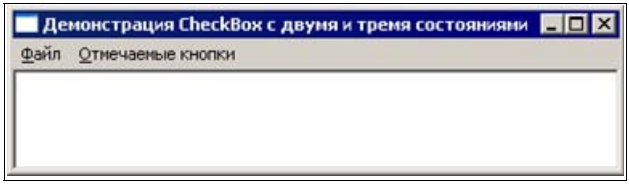


Рис. 9.8. Главное окно проекта UsgCheckBox

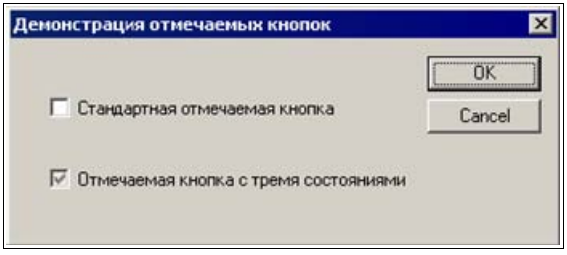


Рис. 9.9. Модальный диалог с отмечаемыми кнопками проекта UsgCheckBox

Надеемся, что приведенный проект еще более ясен после изучения предшествующих демонстрационных проектов. Поэтому, как и ранее, далее ограничимся лишь рассмотрением шаблона ресурсов диалогового окна в части создания и конфигурирования отмечаемых кнопок.

Для размещения в окне диалога отмечаемых кнопок можно воспользоваться панелью **Toolbox** (см. рис. 8.2). В демонстрационном примере для их включения в диалоговое окно достаточно выбрать с помощью мыши на панели **Toolbox** отмечаемую кнопку — флажок (Check Box) и перетащить ее в нужное место диалогового окна. Для демонстрационного примера размеры и расположение отмечаемых кнопок имеют вид, показанный на рис. 9.7.

Для настройки свойств отмечаемой кнопки достаточно вызвать ее контекстное меню, выбрать пункт **Properties** и, в появившейся одноименной вкладке, задать свойства отмечаемой кнопки. Свойства отмечаемых кнопок для демонстрационного примера показаны на рис. 9.10 и 9.11.

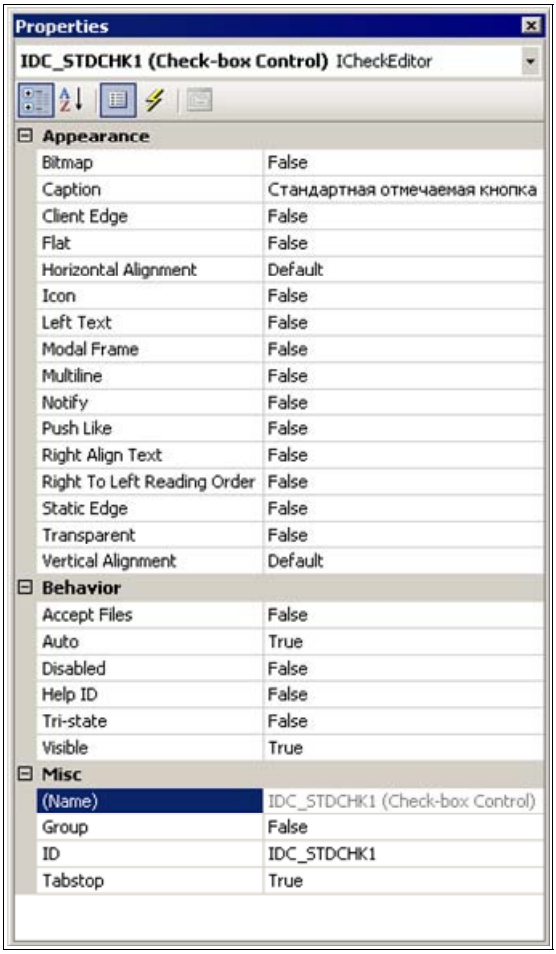


Рис. 9.10. Свойства стандартной отмечаемой кнопки (флажка) шаблона ресурсов диалогового окна проекта UsgCheckBox

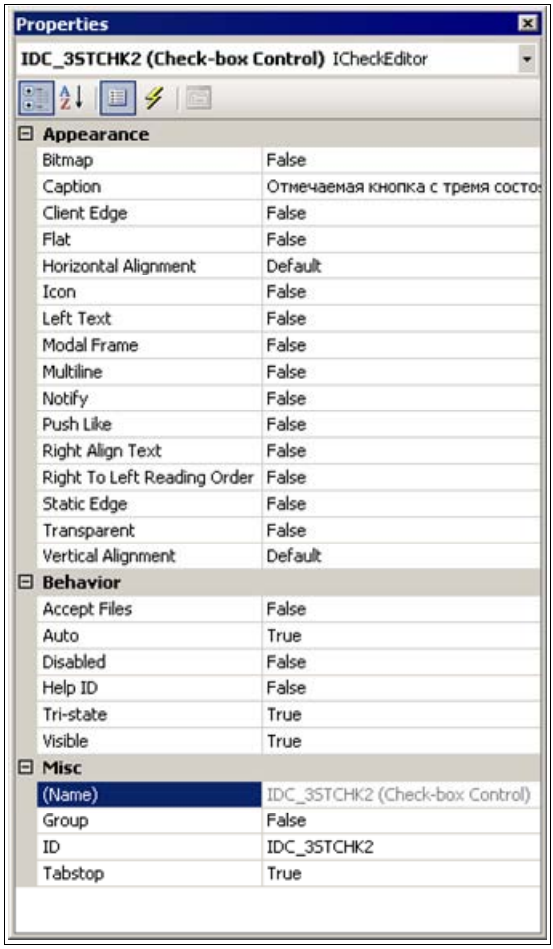


Рис. 9.11. Свойства отмечаемой кнопки (флажка) с тремя состояниями шаблона ресурсов диалогового окна проекта UsgCheckBox

Совет

Обратите внимание на такие свойства отмечаемой кнопки (флажка), как Caption (текст), Left Text (расположение текста: справа или слева от флажка), Right Align Text (выравнивание текста: по левой или правой границам) из группы свойств Appearance, определяющих вид кнопки, свойства Disabled (включение или выключение флажка), Tri-state, Visible (скрытие или отображение флажка) из группы Behavior, определяющих поведение кнопки, свойство ID (идентификатор) из группы смешанных свойств Misc. Используя копию проекта UsgCheckBox, поэкспериментируйте, изменяя значения не только перечисленных, но и остальных свойств отмечаемой кнопки.

Примечание

Обратите внимание на использование идентификатора `IDD_DIALOG_3STSTE` шаблона ресурсов модального диалогового окна (см. рис. 9.7) в конструкторе класса `Dialog`, производного от класса `CDialog` библиотеки MFC (см. листинг 9.6). Обратите также внимание на использование идентификаторов `IDC_STDCHK1` и `IDC_3STCHK2` отмечаемых кнопок (см. рис. 9.10 и 9.11) в таблице сообщений для отмечаемых кнопок (класс `Dialog`, листинг 9.6).

9.1.3. Растровые кнопки

Напомним, что библиотека MFC содержит также класс `CBitMapButton` для кнопок с растровыми изображениями, производный от класса `CButton` и наследующий все его возможности. Кнопка с растровым изображением — это стандартная нажимаемая кнопка, которая вместо поясняющего текста содержит растровый рисунок.

Класс `CBitMapButton`, наследуя все методы класса `CButton`, определяет еще два метода инициализации — `LoadBitmaps` и `AutoLoad` — расширяющие возможности обычной нажимаемой кнопки и обеспечивающие удобные средства для загрузки растровых изображений, которые затем будут отображены в клиентской области растровой кнопки. При этом метод `AutoLoad` автоматически связывает объект с типом `CBitMapButton` с требуемыми растровыми изображениями, пользуясь *связанными* идентификаторами кнопок и Bitmap-ресурсов.

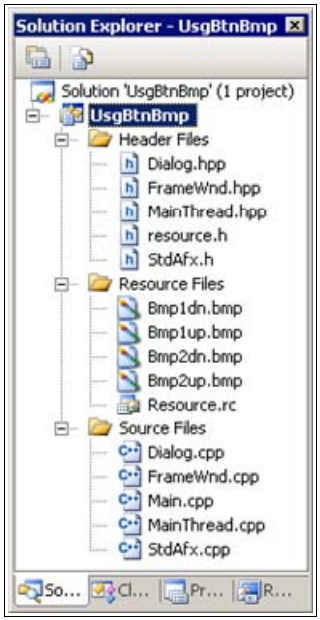


Рис. 9.12. Файловый состав проекта UsgBtnBmp

Демонстрационный проект, иллюстрирующий работу с растровыми кнопками, находится на прилагаемом компакт-диске, в папке \Глава 09\UsgBtnBmp, а файловый состав проекта показан на рис. 9.12.

Совет

Копию проекта UsgBtnBmp для экспериментов создавайте копированием папки проекта с компакт-диска. Файлы StdAfx.h, StdAfx.cpp, Main.cpp, MainThread.hpp, MainThread.cpp в данном проекте являются стандартными, а файлы FrameWnd.hpp, FrameWnd.cpp, Dialog.hpp и Dialog.cpp представлены в листингах 9.7—9.10. Оставшиеся файлы — resource.h, Resourse.rc, Bmp1dn.bmp, Bmp1up.bmp, Bmp2dn.bmp и Bmp2up.bmp — созданы редактором ресурсов и поддерживаются им. Ресурсы проекта представлены на рис. 9.13.

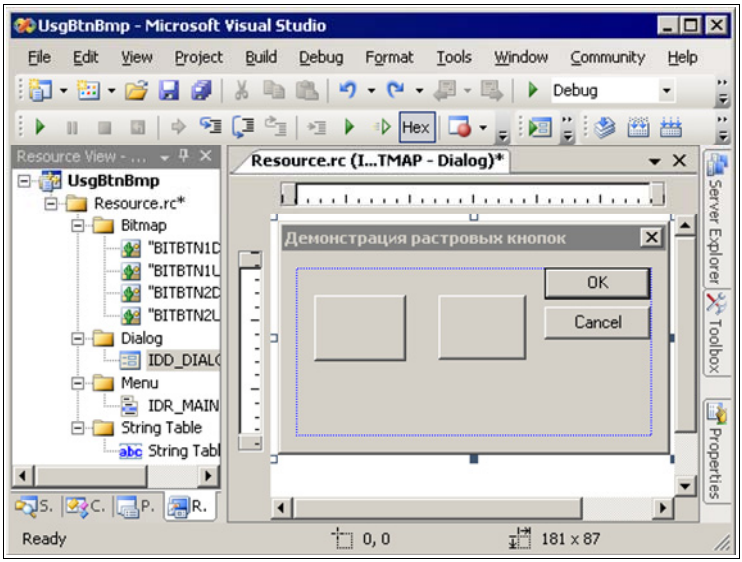


Рис. 9.13. Ресурсы проекта UsgBtnBmp

Листинг 9.7. Файл FrameWnd.hpp

```
/*
   Файл              : FrameWnd.hpp

   Проект            : приложение на основе библиотеки классов MFC
                       (ресурсы — Bitmap, Dialog, Menu и String
                       Table)

   Назначение        : объявление класса "FrameWnd", производного
                       от класса "CFrameWnd" библиотеки классов
                       MFC (обработка сообщений главного окна)
```

```
   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _FrameWnd_hpp
    #define _FrameWnd_hpp

    // Объявление класса
    class FrameWnd : public CFrameWnd
    {
        // Методы

    protected:

        // Прототипы функций, обрабатывающих сообщения
        // Обработка команды Файл | Выход
        afx_msg void OnMenuFileExit( void );
        // Обработка команды Растровые кнопки | Диалог
        afx_msg void OnMenuBitmapButtons( void );

        // Объявление таблицы сообщений
        DECLARE_MESSAGE_MAP( );

    };

#endif
```

Листинг 9.8. Файл FrameWnd.cpp

```
/*
   Файл              : FrameWnd.cpp

   Проект            : приложение на основе библиотеки классов MFC
                       (ресурсы — Bitmap, Dialog, Menu и String
                       Table)

   Назначение        : реализация класса "FrameWnd", производного
                       от класса "CFrameWnd" библиотеки классов
                       MFC (обработка сообщений главного окна)
```

```
   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAFX.h"                  // Прекомпилируемый файл
#include "FrameWnd.hpp"              // Объявление класса FrameWnd
#include "Dialog.hpp"                // Объявление класса Dialog
// Идентификаторы ресурсов (поддержка редактором ресурсов)
#include "Resource.h"

// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( FrameWnd, CFrameWnd )
    ON_COMMAND( ID_FILE_EXIT, OnMenuFileExit )
    ON_COMMAND( ID_BITMAP_BUTTONS, OnMenuBitmapButtons )
END_MESSAGE_MAP( )

// Обработка команды Файл | Выход
afx_msg void FrameWnd :: OnMenuFileExit( void )
{
    BOOL                rc = DestroyWindow( );
    if( !rc )
    {
        TRACE0( "\n Ошибка 2. Окно не было разрушено \n" );
        exit( 2 );
    }

    return afx_msg void( );
}

// Обработка команды Растровые кнопки | Диалог
afx_msg void FrameWnd :: OnMenuBitmapButtons( void )
{
    // Создаем объект диалога — автоматически вызывается
    //   обычный конструктор
    Dialog              Dlg( this );

    // Создается модальное диалоговое окно — оно будет
    //   автоматически закрыто после нажатия OK или Cancel
    Dlg.DoModal( );

    return afx_msg void( );
}
```

Листинг 9.9. Файл Dialog.hpp

```
/*
   Файл                : Dialog.hpp

   Проект              : приложение на основе библиотеки классов MFC
                         (ресурсы — Bitmap, Dialog, Menu и String
                         Table)

   Назначение          : объявление класса "Dialog",
                         производного от класса "CDialog" библиотеки
                         классов MFC (обработка сообщений окна)

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _Dialog_hpp
    #define _Dialog_hpp

    // Объявление класса
    class Dialog: public CDialog
    {
        // Данные

    private:

        // Создание рисованных кнопок — вызывается конструктор
        //   умолчания
        CBitmapButton   BmpBtn1;    // Кнопка 1
        CBitmapButton   BmpBtn2;    // Кнопка 2

        // Методы

    public:

        // Конструктор — передает идентификатор диалогового окна в
        //   базовый класс
        Dialog( CWnd *pParent = 0 );

    protected:

        // Перегружаем виртуальный метод для инициализации
        virtual BOOL OnInitDialog( void );
```

```
        // Обработчики рисованных кнопок
        afx_msg void OnBmpBtn1Click( void );
        afx_msg void OnBmpBtn2Click( void );

        // Объявление таблицы сообщений
        DECLARE_MESSAGE_MAP( );
    };

#endif
```

Листинг 9.10. Файл Dialog.cpp

```
/*
   Файл                : Dialog.cpp

   Проект              : приложение на основе библиотеки классов MFC
                         (ресурсы — Bitmap, Dialog, Menu и String
                         Table)

   Назначение          : реализация класса "BitmapDialog",
                         производного от класса "CDialog" библиотеки
                         классов MFC (обработка сообщений окна)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAFX.h"                  // Прекомпилируемый файл
#include "Dialog.hpp"                // Объявление класса Dialog
// Идентификаторы ресурсов (поддержка редактором ресурсов)
#include "Resource.h"

// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( Dialog, CDialog )
    ON_BN_CLICKED( IDC_BITBTN1, OnBmpBtn1Click )
    ON_BN_CLICKED( IDC_BITBTN2, OnBmpBtn2Click )
END_MESSAGE_MAP( )

// Конструктор
Dialog :: Dialog(
    CWnd                *pParent )  // Указатель на родительское
                                    //   окно (NULL — родительским
                                    //   является главное окно)
```

```
    : CDialog( IDD_DIALOG_BITMAP, pParent )
{ }

// Инициализация окна диалога
BOOL Dialog :: OnInitDialog( void )
{
    // Вызов метода базового класса
    CDialog :: OnInitDialog( );

    // Связывание кнопок с изображениями
    if ( !BmpBtn1.AutoLoad( IDC_BITBTN1, this ) )
    {
        TRACE0( "\n Ошибка 4 использования AutoLoad \n" );
        exit( 4 );
    }
    if ( !BmpBtn2.AutoLoad( IDC_BITBTN2, this ) )
    {
        TRACE0( "\n Ошибка 5 использования AutoLoad \n" );
        exit( 5 );
    }

    return TRUE;
}

// Обработка кнопки, вызывающей Paint
afx_msg void Dialog :: OnBmpBtn1Click( void )
{
    // Запуск постороннего процесса: первый аргумент — командная
    //   строка, второй — в каком виде отображать окно процесса
    WinExec( "mspaint bmp.bmp", SW_SHOWNORMAL );

    return afx_msg void( );
}

// Обработка кнопки, вызывающей NotePad
afx_msg void Dialog :: OnBmpBtn2Click( void )
{
   // Запуск постороннего процесса: первый аргумент — командная
    //   строка, второй — в каком виде отображать окно процесса
    WinExec( "notepad readme", SW_SHOWNORMAL );

    return afx_msg void( );
}
```

Главное окно приложения представлено на рис. 9.14, а модальное диалоговое окно с кнопками показано на рис. 9.15.

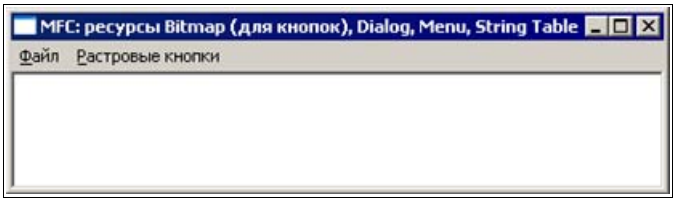


Рис. 9.14. Главное окно проекта UsgBtnBmp

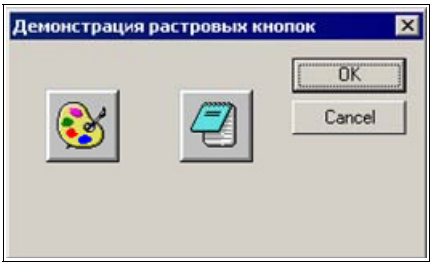


Рис. 9.15. Модальное диалоговое окно с рисованными кнопками проекта UsgBtnBmp

Реакцией на нажатие кнопок с рисунками являются, соответственно, запуск графического редактора Paint и текстового редактора Блокнот (см. листинг 9.11, рис. 9.16 и 9.17):

```
// Запуск постороннего процесса: первый аргумент — командная
//   строка, второй — в каком виде отображать окно процесса
WinExec( "mspaint bmp.bmp", SW_SHOWNORMAL );
WinExec( "notepad readme", SW_SHOWNORMAL );
```

Совет

Обратите внимание на стандартную функцию `WinExec`, используемую для запуска приложения. Возможные значения второго аргумента в ее вызове указаны в листинге 2.3, в *главе 2*.

Особенностью данного демонстрационного проекта является также то, что обработка кнопок выполняется в диалоговом окне и обработка данных при нажатии кнопки **OK** не требуется. По этой причине не надо анализировать и обрабатывать значение, возвращаемое методом `DoModal` (см. листинг 9.9):

```
// Создается модальное диалоговое окно — оно будет
//   автоматически закрыто после нажатия OK или Cancel
Dlg.DoModal( );
```

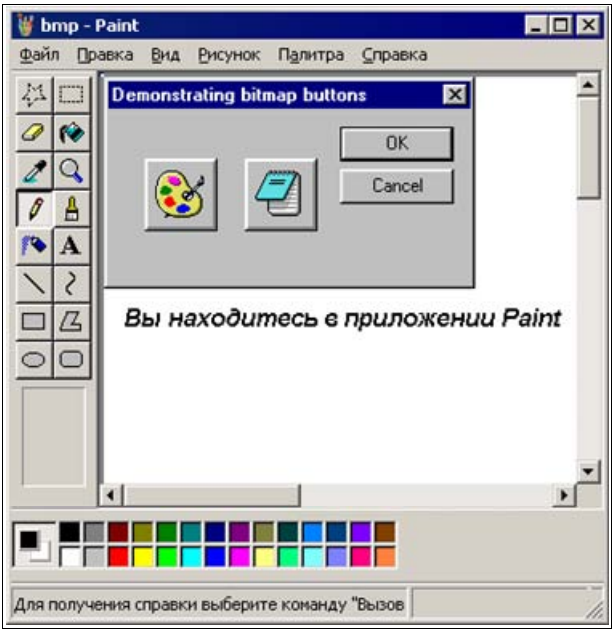


Рис. 9.16. Обработка рисованной кнопки в проекте UsgBtnBmp, запуск редактора Paint

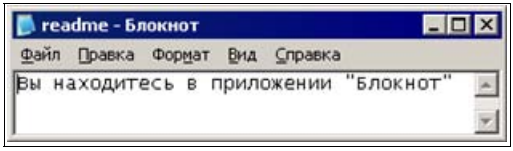


Рис. 9.17. Обработка рисованной кнопки в проекте UsgBtnBmp, запуск Блокнота

Остальная часть исходного кода представляется ясной. Поэтому далее ограничимся лишь рассмотрением ресурса шаблона диалогового окна и Bitmap-ресурсов, в части создания и конфигурирования кнопок с рисунком.

В данном демонстрационном проекте в качестве Bitmap-ресурсов используются четыре нарисованных изображения, соответствующих отжатому и нажатому состоянию двух кнопок (см. рис. 9.15). Для включения в проект указанных ресурсов достаточно на вкладке **Resource View** выбрать проект UsgBtnBmp и выполнить команду **Project | Add Resource....** В появившемся окне **Add Resource** выбрать ресурс **Bitmap** и нажать кнопку **New**. В результате создается шаблон Bitmap-изображения и на вкладке **Resource View** появится ресурс Bitmap с некоторым идентификатором, заданным редактором ресурсов по умолчанию. Если для данного идентификатора двойным "щелчком" левой кнопки мыши загрузить ресурс в окно редактирования, а затем

с помощью правой кнопки мыши вызвать контекстное меню и выбрать **Properties**, то можно задать требуемые свойства Bitmap-ресурса, а также нарисовать изображения растровых кнопок в отжатом и нажатом состояниях. Для демонстрационного проекта изображения кнопок и их свойства приведены на рис. 9.18—9.25.

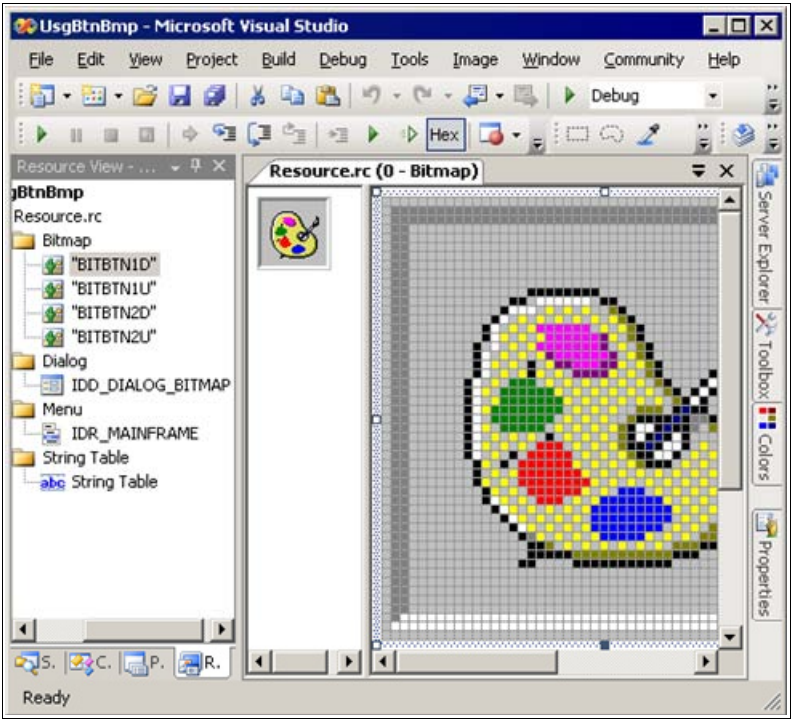


Рис. 9.18. Изображение кнопки Paint при ее нажатии в проекте UsgBtnBmp

Замечание

Обратите внимание на изображения кнопки в отжатом и нажатом состояниях. Они очень похожи (практически совпадают) и отличаются только небольшим сдвигом изображения и цветом внешнего слоя. Такое решение создает эффект нажатия кнопки в работающем приложении.

Для размещения в диалоговом окне кнопок с рисунком можно воспользоваться панелью **Toolbox** (см. рис. 8.2). Выберите кнопку (Button) и перетащите ее в нужное место диалогового окна. Связь кнопки с ее рисованным изображением выполняется программно (см. далее о методе `AutoLoad`). Размеры кнопки также можно легко изменить желаемым образом. Для демонстрационного примера размеры и расположение кнопок имеют вид, показанный на рис. 9.26.

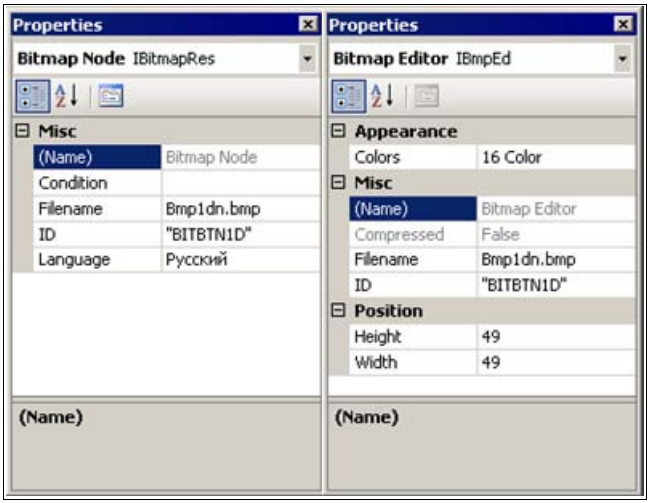


Рис. 9.19. Свойства Bitmap-ресурса и кнопки Paint при ее нажатии в проекте UsgBtnBmp

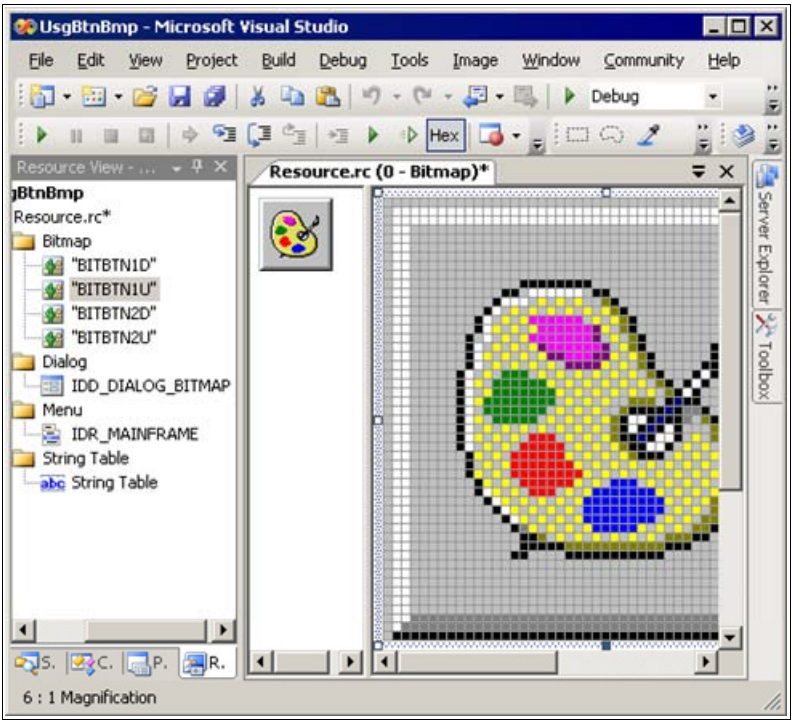


Рис. 9.20. Изображение кнопки Paint при ее отжатии в проекте UsgBtnBmp

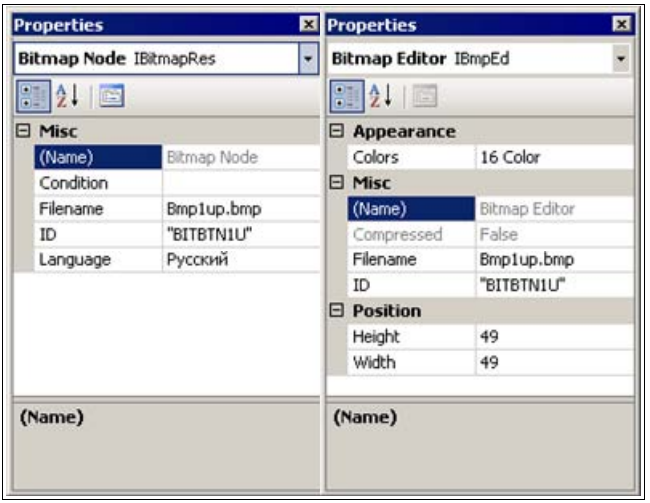


Рис. 9.21. Свойства Bitmap-ресурса и кнопки Paint при ее отжатии в проекте UsgBtnBmp

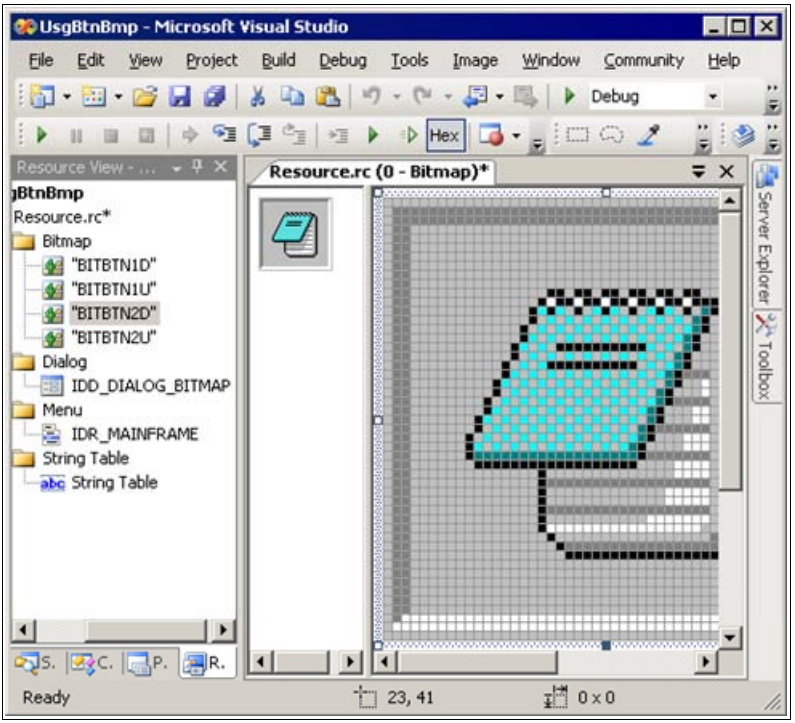


Рис. 9.22. Изображение кнопки Блокнот при ее нажатии в проекте UsgBtnBmp

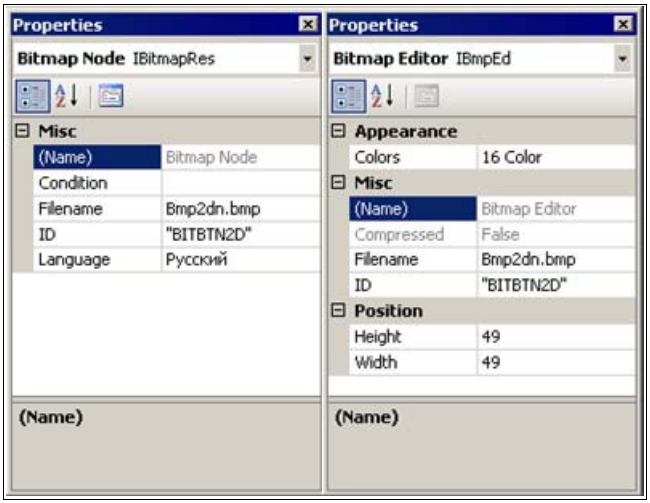


Рис. 9.23. Свойства Bitmap-ресурса и кнопки Блокнот при ее нажатии в проекте UsgBtnBmp

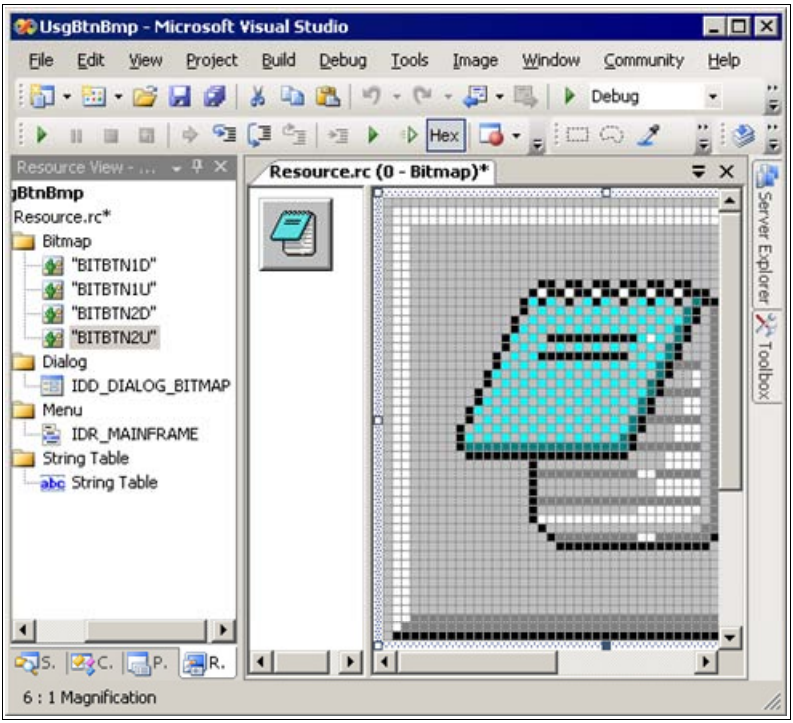


Рис. 9.24. Изображение кнопки Блокнот при ее отжатии в проекте UsgBtnBmp

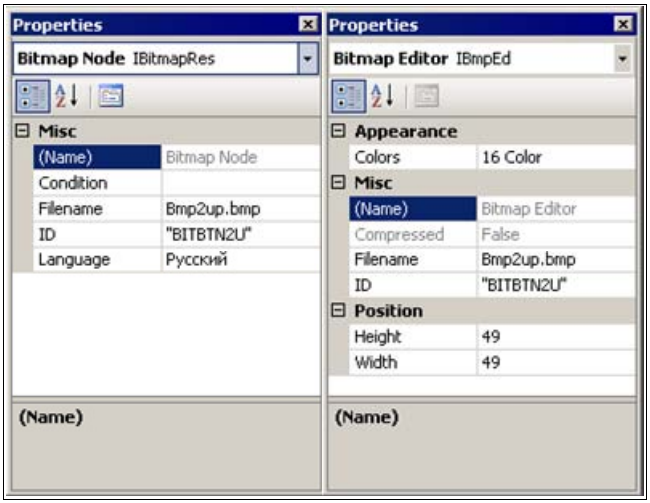


Рис. 9.25. Свойства Bitmap-ресурса и кнопки Блокнот при ее отжатии в проекте UsgBtnBmp

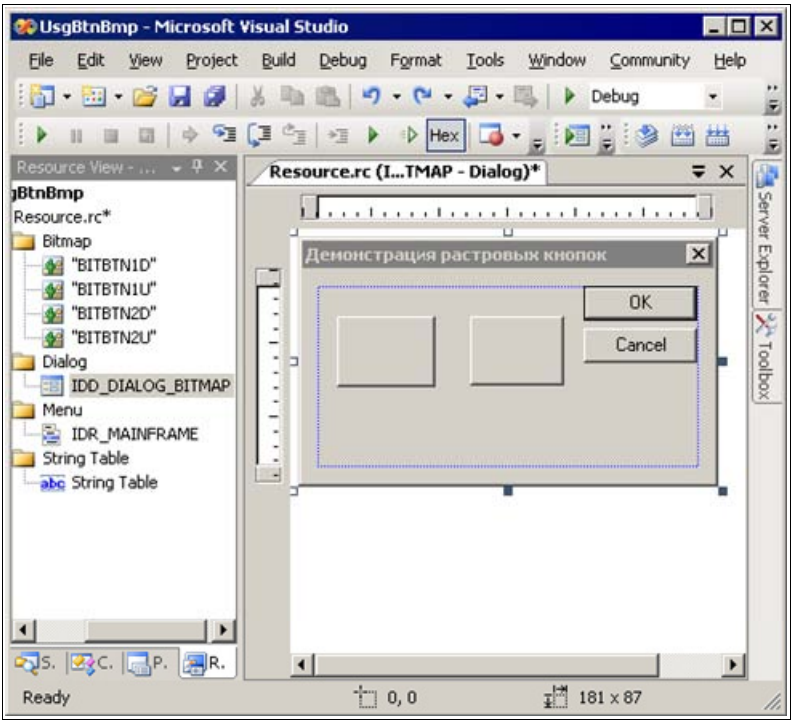


Рис. 9.26. Размещение кнопок с рисунком в модальном диалоговом окне проекта UsgBtnBmp

Для настройки свойств кнопки с рисунком достаточно вызвать ее контекстное меню, выполнить команду **Properties** и задать свойства кнопки. Свойства таких кнопок для демонстрационного примера показаны на рис. 9.27.

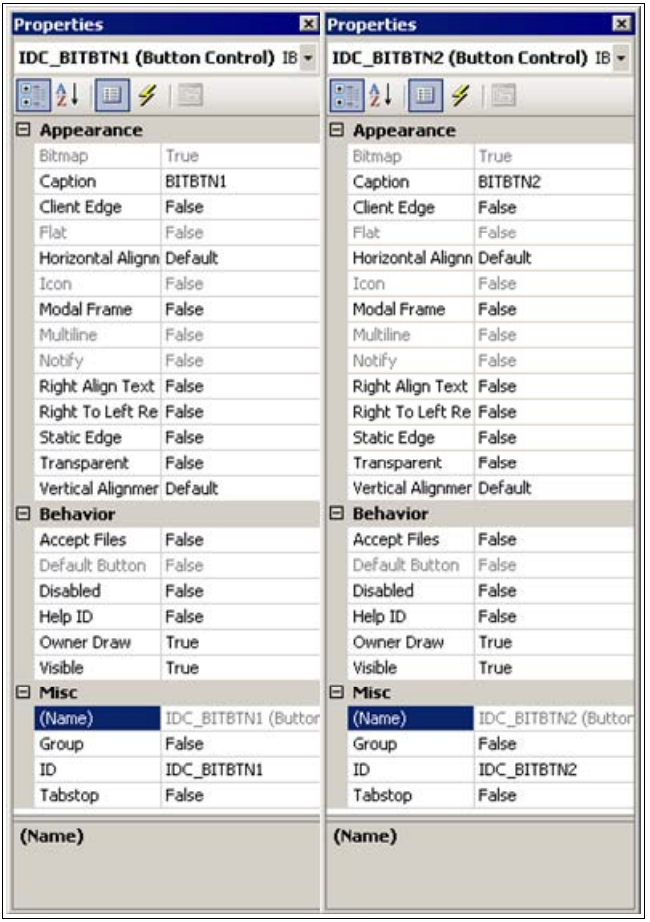


Рис. 9.27. Свойства кнопок шаблона ресурсов диалогового окна проекта UsgBtnBmp

Как указывалось ранее, метод `AutoLoad` автоматически связывает объект `CBitMapButton` с требуемыми растровыми изображениями, пользуясь связанными идентификаторами кнопок и Bitmap-ресурсов. В демонстрационном проекте идентификатору `IDC_BITBTN1` кнопки запуска приложения Paint соответствуют идентификаторы Bitmap-ресурсов `BITBTN1D` (нажатое состояние) и `BITBTN1U` (отжатое состояние). Аналогично, идентификатору `IDC_BITBTN2` растровой кнопки запуска приложения Блокнот соответствуют идентификаторы Bitmap-ресурсов `BITBTN2D` (нажатое состояние) и `BITBTN2U` (отжатое состоя-

ние). Автоматическое связывание кнопок с соответствующими изображениями выполняется следующим образом (см. листинг 9.10):

```
// Связывание кнопок с изображениями
if ( !BmpBtn1.AutoLoad( IDC_BITBTN1, this ) )
{
    TRACE0( "\n Ошибка 4 использования AutoLoad \n" );
    exit( 4 );
}
if ( !BmpBtn2.AutoLoad( IDC_BITBTN2, this ) )
{
    TRACE0( "\n Ошибка 5 использования AutoLoad \n" );
    exit( 5 );
}
```

Совет

Обратите внимание на свойства растровых кнопок и Bitmap-ресурсов. Используя копию проекта UsgBtnBmp, поэкспериментируйте, изменяя значения указанных свойств.

9.2. Спин (spin) с дружественным окном. Класс *CSpinButtonCtrl* библиотеки MFC

Спин (spin) — представляет собой элемент управления, состоящий из двух связанных кнопок с изображениями стрелок. Спины предназначены для получения информации от пользователя. Нажатие на кнопки со стрелками уменьшает или увеличивает внутреннее значение вращателя, называемое текущим положением. Это значение выводится в дружественное окно — окно редактирования, дополнительно позволяющее пользователю корректировать значение.

Спин и его дружественное окно выглядят и действуют как единый элемент управления. Спин автоматически посылает текущее положение прокрутки дружественному окну, изменяя его текст. Текущее положение ограничено минимальным и максимальным значениями, определяемыми приложением. На рис. 9.28 показан типичный вид спина с дружественным окном редактирования.

Все функциональные возможности спина определены в классе `CSpinButtonCtrl` библиотеки MFC. Класс `CSpinButtonCtrl` также является производным от класса `CWnd` и наследует все его возможности. Еще раз напомним, что полные сведения о положении класса `CSpinButtonCtrl` в иерархии классов библиотеки MFC можно получить в справочной системе.

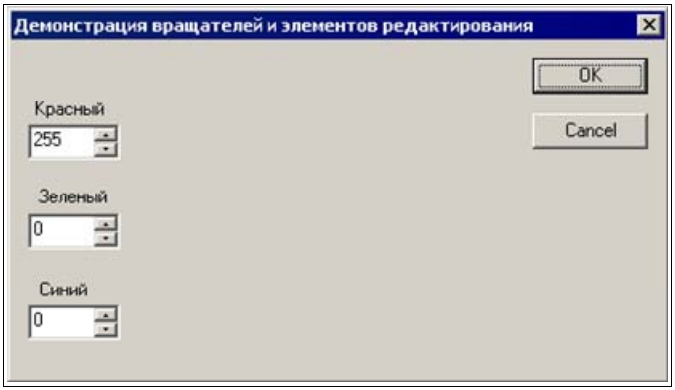


Рис. 9.28. Типичный вид вращателей с дружественными окнами редактирования

Как и для других, ранее рассмотренных управляющих элементов, спин может быть создан в программном коде как подчиненный элемент любого окна или же использован в шаблоне ресурсов диалогового окна, что гораздо удобнее. Спин посылает сообщения своему окну-родителю, которые могут быть перехвачены и обработаны в коде таблицы сообщений и методов их обработки.

9.2.1. Стили, сообщения и методы класса *CSpinButtonCtrl*

Спины могут использовать элементарные стили, доступные всем объектам класса `CWnd`, а также ряд специальных стилей (табл. 9.2). Стиль определяет его внешний вид и поведение. Стиль устанавливается при инициализации (методом `CSpinButtonCtrl::Create`) или задается в свойствах, в редакторе ресурсов.

Таблица 9.2. *Элементарные стили вращателя*

Макроопределение стиля	Значение
`UDS_ALIGNLEFT`, свойство **Alignment** равно **Left**	Выравнивает спин по левому краю дружественного окна редактирования
`UDS_ALIGNRIGHT`, свойство **Alignment** равно **Right Align**	Выравнивает спин по правому краю дружественного окна редактирования
`UDS_ARROWKEYS`, свойство **Arrow Keys** равно **True**	Позволяет вращателю уменьшать или увеличивать текущее положение с помощью его кнопок
`UDS_AUTOBUDDY`, свойство **Auto Buddy** равно **True**	Автоматически помещает дружественное окно редактирования в фокус

Таблица 9.2 *(окончание)*

Макроопределение стиля	Значение
UDS_HORZ, свойство **Orientation** равно **Horizontal**	Стрелки вращателя направлены горизонтально
UDS_HOTTRACK, свойство **Hot Track** равно **True**	Выдвигает на передний план кнопки вращателя, когда курсор перемещается над ними (действует только в операционных системах Windows 98/2000)
UDS_NOTHOUSANDS, свойство **No Thousands** равно **True**	Предотвращает отображение разделителя после каждых трех десятичных цифр
UDS_SETBUDDYINT, свойство **Set Buddy Integer** равно **True**	Задает текст в дружественном окне при изменении текущего положения. Текст представляет собой значение текущего положения в десятичном или шестнадцатеричном виде
UDS_WRAP, свойство **Wrap** равно **True**	Значения, превышающие максимум, вызывают переход к началу интервала и наоборот

Примечание

Все макросы стилей вращателя начинаются с префикса UDS_ (от UDS — Up-Down Styles), что указывает на первоначальное название этого элемента управления.

Сообщения вращателя могут быть перехвачены и обработаны, если существуют соответствующие элементы таблицы сообщений и методы их обработки. Элементы таблицы сообщений для спина приведены в табл. 9.3.

Таблица 9.3. *Элементы таблицы сообщений*

Элемент таблицы сообщений	Значение
ON_WM_HSCROLL	Посылается спином, имеющим стиль UDS_HORZ при нажатии на кнопки со стрелками
ON_WM_VSCROLL	Посылается спином, имеющим стиль UDS_VERT при нажатии на кнопки со стрелками
ON_EN_UPDATE	Посылается дружественным окном редактирования при изменении его текста

Примечание

При использовании спина с дружественным элементом редактирования достаточно использовать в таблице сообщений элемент ON_EN_UPDATE.

Методы класса `CSpinButtonCtrl` библиотеки MFC, перечисленные в табл. 9.4, описывают способы получения (get) и установки (set) свойств спина.

Таблица 9.4. *Методы класса* `CSpinButtonCtrl` *библиотеки MFC*

Метод	Описание
`GetAccel`	Возвращает информацию об акселераторе спина
`GetBase`	Возвращает основание текущей системы счисления вращателя
`GetBuddy`	Возвращает указатель на текущее дружественное окно редактирования
`GetPos`	Возвращает текущее положение
`GetRange`	Возвращает нижнюю и верхнюю границы диапазона значений текущего положения спина
`SetAccel`	Устанавливает акселератор спина
`SetBase`	Устанавливает основание текущей системы счисления
`SetBuddy`	Устанавливает текущее дружественное окно редактирования
`SetPos`	Устанавливает текущее положение
`SetRange`	Устанавливает нижнюю и верхнюю границы диапазона значений текущего положения

9.2.2. Демонстрационная программа UsgSpinEdit: вращатели с дружественными окнами редактирования

Демонстрационный проект, иллюстрирующий работу вращателей с дружественными окнами редактирования, находится на прилагаемом компакт-диске, в папке \Глава 09\UsgSpinEdit, а файловый состав проекта показан на рис. 9.29.

Совет

Копию проекта UsgSpinEdit для экспериментов создавайте копированием проекта с компакт-диска. Файлы StdAfx.h, StdAfx.cpp, Main.cpp, MainThread.hpp, MainThread.cpp, FrameWnd.hpp, FrameWnd.cpp в данном проекте являются, по сути, стандартными или рассмотрены в предыдущих демонстрационных примерах, а файлы Dialog.hpp и Dialog.cpp представлены в листингах 9.11 и 9.12. Оставшиеся файлы — resource.h, Resourse.rc — созданы редактором ресурсов и поддерживаются им. Ресурсы проекта представлены на рис. 9.30.

Рис. 9.29. Файловый состав проекта UsgSpinEdit

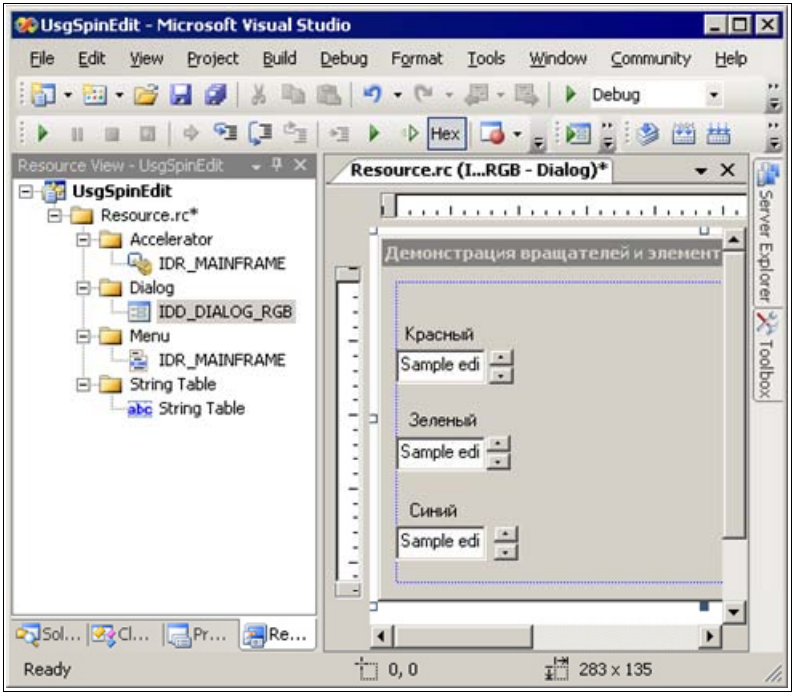


Рис. 9.30. Ресурсы проекта UsgSpinEdit

Листинг 9.11. Файл Dialog.hpp

```
/*
   Файл                : Dialog.hpp

   Проект              : демонстрация Windows-приложения на основе
                         библиотеки классов MFC с модальным
                         диалоговым окном, использующим спины и окна
                         редактирования

   Назначение          : объявление класса "Dialog", производного
                         от класса "CDialod" библиотеки классов
                         MFC (обработка сообщений окна)

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _Dialog_hpp
    #define _Dialog_hpp

    // Объявление класса
    class Dialog : public CDialog
    {
        // Данные

    private:

        // Указатели элементов редактирования красного, зеленого и
        //   синего цветов
        CEdit           *pEditRed;
        CEdit           *pEditGreen;
        CEdit           *pEditBlue;

        // Указатели вращателей для красного, зеленого и синего
        //   цветов
        CSpinButtonCtrl *pSpinRed;
        CSpinButtonCtrl *pSpinGreen;
        CSpinButtonCtrl *pSpinBlue;

        // true — инициализация окна диалога была успешной
        bool            m_InitIsDone;
```

```
    public:

        COLORREF        m_OldRGB;   // Старый RGB-цвет
        COLORREF        m_NewRGB;   // Новый RGB-цвет

        // Методы

    public:

        // Конструктор — передает идентификатор диалогового окна в
        //   базовый класс
        Dialog( CWnd *pParent = 0 );

    protected:

        // Перегружаем виртуальный метод для инициализации
        virtual BOOL OnInitDialog( void );

        // Перегружаем виртуальный метод для получения информации
        //   из диалогового окна
        virtual void OnOK( void );

        // Обработчик сообщений вращателей
        afx_msg void OnEditColor( void );

        DECLARE_MESSAGE_MAP( );
    };

#endif
```

Листинг 9.12. Файл Dialog.cpp

```
/*
   Файл              : Dialog.cpp

   Проект            : демонстрация Windows-приложения на основе
                       библиотеки классов MFC с модальным
                       диалоговым окном, использующим вращатели и окна
                       редактирования

   Назначение        : реализация класса "Dialog", производного
                       от класса "CDialod" библиотеки классов
                       MFC (обработка сообщений окна)
```

```
   Microsoft Visual Studio C++ .NET 2005
*/


#include "StdAfx.h"                    // Прекомпилируемый файл
#include "FrameWnd.hpp"                // Объявление класса FrameWnd
#include "Dialog.hpp"                  // Объявление класса Dialog
// Идентификаторы ресурсов (поддержка редактором ресурсов)
#include "resource.h"

// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( Dialog, CDialog )
    ON_EN_UPDATE( IDC_EDIT_RED, OnEditColor )
    ON_EN_UPDATE( IDC_EDIT_GREEN, OnEditColor )
    ON_EN_UPDATE( IDC_EDIT_BLUE, OnEditColor )
END_MESSAGE_MAP( )

// Конструктор
Dialog :: Dialog(
    CWnd                *pParent )  // Указатель на родительское
                                    //   окно (NULL — родительским
                                    //   является главное окно)
    : CDialog( IDD_DIALOG_RGB, pParent )
      {
          m_InitIsDone = false;
      }

// Инициализация диалогового окна
BOOL Dialog :: OnInitDialog( void )
{
    // Вызываем метод базового класса
    CDialog :: OnInitDialog( );

    BYTE                Red;        // Красная составляющая "m_RGB"
    BYTE                Green;      // Зеленая составляющая "m_RGB"
    BYTE                Blue;       // Синяя составляющая "m_RGB"

    // Получаем указатели на вращатели и элементы редактирования
    //   красного, зеленого и синего цветов
    pSpinRed = static_cast< CSpinButtonCtrl * >
               ( GetDlgItem( IDC_SPIN_RED ) );
    pSpinGreen = static_cast< CSpinButtonCtrl * >
                 ( GetDlgItem( IDC_SPIN_GREEN ) );
```

```
    pSpinBlue = static_cast< CSpinButtonCtrl * >
                ( GetDlgItem( IDC_SPIN_BLUE ) );
    // Нижеследующие указатели нужны только для задания ограничения
    //   на длину вводимого текста
    pEditRed = static_cast< CEdit * >
               ( GetDlgItem( IDC_EDIT_RED ) );
    pEditGreen = static_cast< CEdit * >
                 ( GetDlgItem( IDC_EDIT_GREEN ) );
    pEditBlue = static_cast< CEdit * >
                ( GetDlgItem( IDC_EDIT_BLUE ) );

    // Устанавливаем диапазоны значений для вращателей красного,
    //   зеленого и синего цветов (обращаем внимание на то, что
    //   можно обрабатывать возвращаемое значение — для сокращения
    //   объема исходного текста мы это не делаем, но вам
    //   предлагается выполнить такую модификацию)
    pSpinRed->SetRange( 0, 255 );
    pSpinGreen->SetRange( 0, 255 );
    pSpinBlue->SetRange( 0, 255 );

    // Выделяем красную, зеленую и синюю составляющие "старого"
    //   RGB-цвета
    Red = GetRValue( m_OldRGB );
    Green = GetGValue( m_OldRGB );
    Blue = GetBValue( m_OldRGB );

    // Устанавливаем значения цветов для вращателей
    pSpinRed->SetPos( Red );
    pSpinGreen->SetPos( Green );
    pSpinBlue->SetPos( Blue );

    // Устанавливаем длины текста в байтах (3) для элементов
    //   редактирования цветов
    pEditRed->SetLimitText( 3 );
    pEditGreen->SetLimitText( 3 );
    pEditBlue->SetLimitText( 3 );

    // Инициализация диалога выполнена
    m_InitIsDone = true;

    return TRUE;
}
```

```
// Получение информации из диалогового окна (обработчик кнопки OK)
void Dialog :: OnOK( void )
{
    // Вызываем метод базового класса
    CDialog :: OnOK( );

    BYTE                Red;        // Красная составляющая "m_RGB"
    BYTE                Green;      // Зеленая составляющая "m_RGB"
    BYTE                Blue;       // Синяя составляющая "m_RGB"

    // Получаем текущие значения цветов от вращателей красного,
    //   зеленого и синего цветов
    Red = pSpinRed->GetPos( );
    Green = pSpinGreen->GetPos( );
    Blue = pSpinBlue->GetPos( );

    // Формируем новый RGB-цвет
    m_NewRGB = RGB( Red, Green, Blue );

    return;
}

// Редактирование составляющих цвета
void Dialog :: OnEditColor( void )
{
    BYTE                Red;        // Красная составляющая "m_RGB"
    BYTE                Green;      // Зеленая составляющая "m_RGB"
    BYTE                Blue;       // Синяя составляющая "m_RGB"
    // Указатель родительского окна
    FrameWnd           *pParent;

    if ( !m_InitIsDone )
        return;                     // Выход, если не было успешной
                                    //   инициализации

    // Получаем текущие значения цветов от вращателей красного,
    //   зеленого и синего цветов
    Red = pSpinRed->GetPos( );
    Green = pSpinGreen->GetPos( );
    Blue = pSpinBlue->GetPos( );

    // Формируем новый RGB-цвет
    m_NewRGB = RGB( Red, Green, Blue );
```

```
    // Получаем указатель родительского окна и устанавливаем для
    //   него новый RGB-цвет
    pParent = static_cast< FrameWnd * >( GetParent( ) );
    pParent->TryPenRGB( m_NewRGB );

    return;
}
```

Главное окно приложения представлено на рис. 9.31, а модальное диалоговое окно с вращателями и дружественными элементами редактирования показано на рис. 9.28.

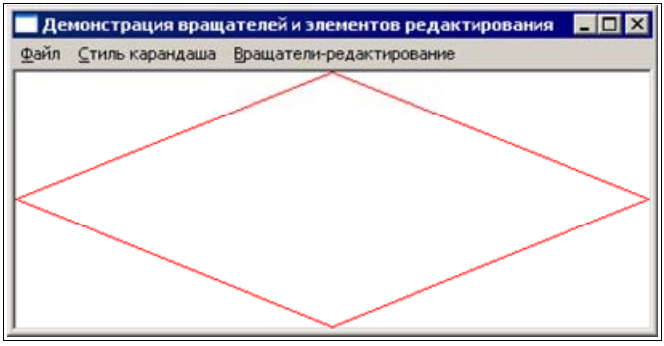


Рис. 9.31. Главное окно проекта UsgSpinEdit

В листингах 9.11 и 9.12 следует, прежде всего, обратить внимание на таблицу сообщений и методы их обработки. Поскольку в демонстрационной программе используются вращатели с дружественными элементами редактирования, в которых автоматически изменяется текст, отображающий значение текущего положения вращателя, то в таблице сообщений достаточно лишь использовать макрос `ON_EN_UPDATE`. Обратите также внимание на использование методов `SetRange`, `SetPos` и `GetPos` (см. табл. 9.4 и листинг 9.12). Остальная часть программного кода представляется ясной. Поэтому далее ограничимся лишь рассмотрением шаблона ресурсов диалогового окна и вращателей в части создания и конфигурирования вращателей с дружественными элементами редактирования.

Для размещения спинов с дружественными элементами редактирования в диалоговом окне можно воспользоваться панелью управляющих элементов **Toolbox** (см. рис. 8.2). Выберите спин (Spin Control) на панели **Toolbox**, перетащите его в нужное место диалогового окна, выберите элемент редактирования (Edit Control) и разместите его рядом с вращателем, затем выберите элемент надписи (Static Text) и разместите его над элементом редактирования

и вращателем. Для демонстрационного примера размеры и расположение элементов имеют вид, показанный ранее на рис. 9.28.

Для настройки свойств любого управляющего элемента диалогового окна достаточно вызвать его контекстное меню, выполнить команду **Properties** и в появившейся вкладке задать свойства элемента. Свойства управляющих элементов для задания красной составляющей цвета, для демонстрационного примера, показаны на рис. 9.32—9.34.

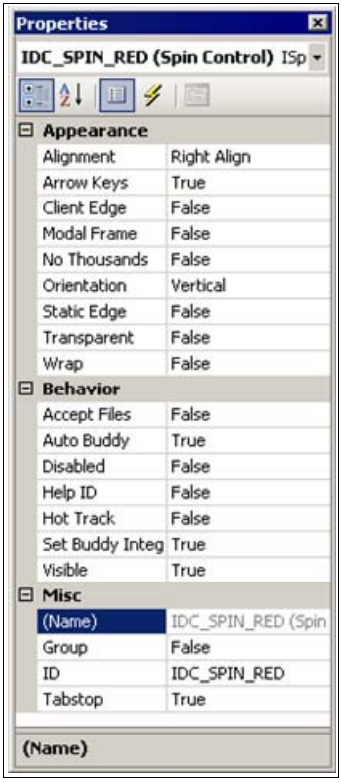


Рис. 9.32. Свойства вращателя для задания красной составляющей цвета шаблона ресурсов диалогового окна проекта UsgSpinEdit

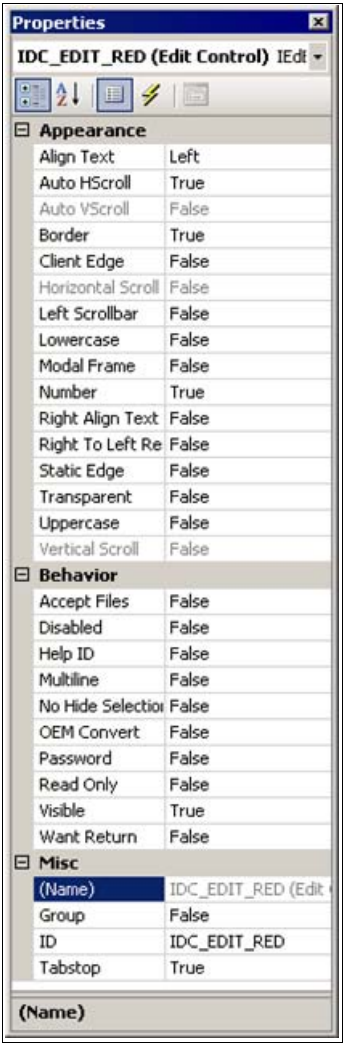


Рис. 9.33. Свойства дружественного элемента редактирования для задания красной составляющей цвета шаблона ресурсов диалогового окна проекта UsgSpinEdit

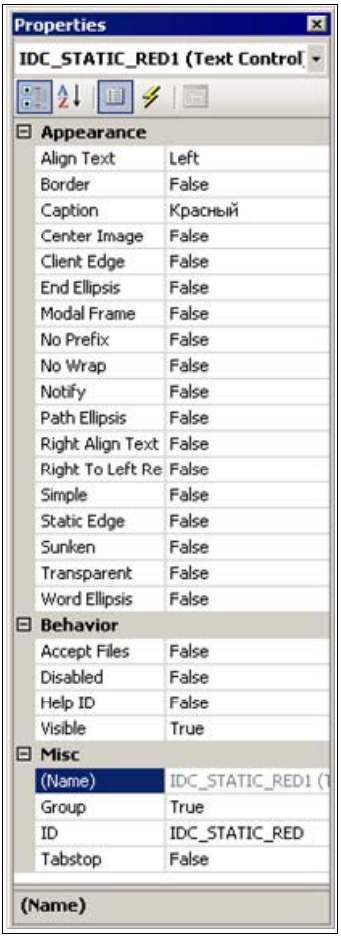


Рис. 9.34. Свойства поясняющего текста для задания красной составляющей цвета шаблона ресурсов диалогового окна проекта UsgSpinEdit

Остальные управляющие элементы для задания зеленой и синей составляющих цвета имеют аналогичные настройки.

Совет

Обратите внимание на идентификаторы вращателей (`IDC_SPIN_RED`, `IDC_SPIN_GREEN` и `IDC_SPIN_BLUE`), дружественных элементов редактирования (`IDC_EDIT_RED`, `IDC_EDIT_GREEN` и `IDC_EDIT_BLUE`) и их использование в исходном коде (листинг 9.12). Изучите свойства спинов и дружественных элементов редактирования. Используя копию проекта UsgSpinEdit, поэкспериментируйте, изменяя значения указанных свойств.

9.3. Полоса прокрутки. Класс *CScrollBar* библиотеки MFC

Полоса прокрутки — это интерактивный визуальный элемент управления, способный реагировать на ввод пользователя несколькими способами. Полоса прокрутки состоит из бегунка, перемещающегося по полосе, и кнопок со стрелками на каждом конце полосы (рис. 9.35).

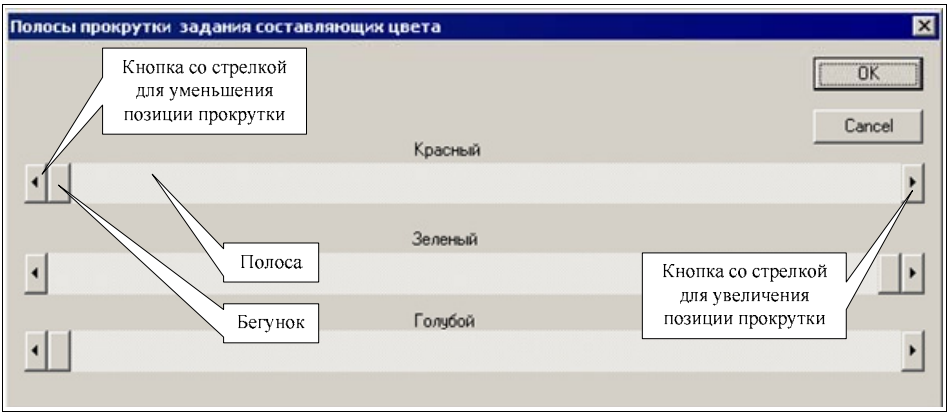


Рис. 9.35. Компоненты типичной полосы прокрутки (проект UsgScroll)

Полоса прокрутки может быть расположена горизонтально или вертикально. При нажатии на кнопки со стрелками происходит уменьшение или увеличение внутреннего значения, называемого позицией прокрутки. Позицию прокрутки можно также изменить, если передвинуть бегунок или щелкнуть в нужном месте на полосе. Максимальное и минимальное значения позиции прокрутки устанавливаются программно.

Все функциональные возможности полосы прокрутки определены в классе `CScrollBar` библиотеки MFC. Как и для всех других управляющих элементов библиотеки MFC, класс `CScrollBar` является производным от класса `CWnd` и наследует всю гамму его возможностей. Самые точные и полные сведения о положении класса `CScrollBar` в иерархии классов библиотеки MFC можно получить в справочной системе.

Полоса прокрутки может быть создана в программном коде как подчиненный элемент любого окна или же использована в шаблоне диалогового окна (последнее предпочтительнее и удобнее). Полоса прокрутки посылает сообщения своему окну-родителю (обычно это объект класса, производного от класса `CDialog`), которые могут быть перехвачены и обработаны в коде таблицы сообщений и методов их обработки.

9.3.1. Стили, сообщения полосы прокрутки и методы класса *CScrollBar*

Полоса прокрутки может использовать элементарные стили, доступные всем объектам класса `CWnd`, а также ряд специальных стилей (табл. 9.5). Стиль полосы прокрутки определяет ее внешний вид и поведение. Стиль устанавливается либо при инициализации методом `CScrollBar::Create`, либо при использовании полосы прокрутки в шаблоне ресурсов диалогового окна путем задания ее свойств.

Таблица 9.5. *Элементарные стили полосы прокрутки*

Макроопределение стиля, элемент управления	**Значение**
`SBS_HORZ`, в диалоговом окне — элемент управления **Horizontal Scroll Bar**	Создает горизонтальную полосу прокрутки с высотой, шириной и координатами, указанными в методе `Create` (если только не указаны стили `SBS_BOTTOMALIGN` или `SBS_TOPALIGN`)
`SBS_BOTTOMALIGN`, свойство **Align** равно **Bottom**	Выравнивает нижнюю границу полосы прокрутки по нижней границе прямоугольника, указанного в методе `Create`. Полоса прокрутки имеет стандартную для всех системных полос прокрутки высоту. Используется вместе со стилем `SBS_HORZ`
`SBS_TOPALIGN`, свойство **Align** равно **Top**	Выравнивает верхнюю границу полосы прокрутки по верхней границе прямоугольника, указанного в методе `Create`. Полоса прокрутки имеет стандартную для всех системных полос прокрутки высоту. Используется вместе со стилем `SBS_HORZ`
`SBS_VERT`, в диалоговом окне элемент управления **Vertical Scroll Bar**	Создает вертикальную полосу прокрутки с высотой, шириной и координатами, заданными в методе `Create` (если только не указаны стили `SBS_LEFTALIGN` или `SBS_RIGHTALIGN`)
`SBS_LEFTALIGN`, свойство **Align** равно **Left**	Выравнивает левую границу полосы прокрутки по левой границе прямоугольника, указанного в методе `Create`. Полоса прокрутки имеет стандартную для всех системных полос прокрутки высоту. Используется вместе со стилем `SBS_VERT`
`SBS_RIGHTALIGN`, свойство **Align** равно **Right**	Выравнивает правую границу полосы прокрутки по правой границе прямоугольника, указанного в методе `Create`. Полоса прокрутки имеет стандартную для всех системных полос прокрутки высоту. Используется вместе со стилем `SBS_VERT`

Примечание

Все макросы стилей полосы прокрутки начинаются с префикса `SBS_` (SBS от Scroll Bar Styles).

Сообщения полосы прокрутки могут быть перехвачены и обработаны, если существуют соответствующие элементы таблицы сообщений и методы их обработки. Элементы таблицы сообщений для полосы прокрутки приведены в табл. 9.6.

Таблица 9.6. *Элементы таблицы сообщений для полосы прокрутки*

Элемент таблицы сообщений	Значение
`ON_WM_HSCROLL`	Посылается полосой прокрутки, имеющей стиль `SBS_HORZ` при изменении позиции прокрутки
`ON_WM_VSCROLL`	Посылается полосой прокрутки, имеющей стиль `SBS_VERT` при изменении позиции прокрутки

Полосы прокрутки — "ленивые" элементы управления, которые сами по себе ничего не делают. Поэтому каждый раз приходится сообщать им, что вы от них хотите. Элементам таблицы сообщений соответствуют два метода обработки сообщений полос прокрутки. Прототипы методов обработки сообщений от горизонтальной и вертикальной полос прокрутки имеют следующий вид:

```
// Обработчик сообщений от горизонтальной полосы прокрутки
afx_msg void OnHScroll( UINT nSBCode, UINT nPos,
                        CScrollBar *pScrollBar );
// Обработчик сообщений от вертикальной полосы прокрутки
afx_msg void OnVScroll( UINT nSBCode, UINT nPos,
                        CScrollBar *pScrollBar );
```

Первый параметр — `nSBCode` — это один из возможных кодов уведомлений прокрутки (табл. 9.7), сообщающий приложению, какие действия совершает пользователь с полосой прокрутки.

Таблица 9.7. *Коды уведомлений полос прокрутки, посылаемые методам* `OnHScroll` *и* `OnVScroll`

Код уведомления	Значение
`SB_LINELEFT`	Прокрутка влево на одну единицу (нажата левая кнопка горизонтальной полосы прокрутки — уменьшение позиции прокрутки)
`SB_LINERIGHT`	Прокрутка вправо на одну единицу (нажата правая кнопка горизонтальной полосы прокрутки — увеличение позиции прокрутки)

Таблица 9.7 *(окончание)*

Код уведомления	Значение
SB_PAGELEFT	Прокрутка влево на одну страницу (щелчок левее бегунка горизонтальной полосы прокрутки — уменьшение позиции прокрутки)
SB_PAGERIGHT	Прокрутка вправо на одну страницу (щелчок правее бегунка горизонтальной полосы прокрутки — увеличение позиции прокрутки)
SB_LINEUP	Прокрутка вверх на одну единицу (нажата верхняя кнопка вертикальной полосы прокрутки — уменьшение позиции прокрутки)
SB_LINEDOWN	Прокрутка вниз на одну единицу (нажата нижняя кнопка вертикальной полосы прокрутки — увеличение позиции прокрутки)
SB_PAGEUP	Прокрутка вверх на одну страницу (щелчок выше бегунка вертикальной полосы прокрутки — уменьшение позиции прокрутки)
SB_PAGEDOWN	Прокрутка вниз на одну страницу (щелчок ниже бегунка вертикальной полосы прокрутки — увеличение позиции прокрутки)
SB_THUMBTRACK	Перемещение бегунка (новая позиция хранится в nPos)

Второй параметр методов OnHScroll и OnVScroll — nPos — задает новую позицию бегунка после его перемещения.

Третий параметр — pScrollBar — задает указатель на полосу прокрутки, которая посылает сообщение.

Методы класса CScrollBar библиотеки MFC, перечисленные в табл. 9.8, описывают способы получения (get) и установки (set) свойств полосы прокрутки.

Таблица 9.8. *Методы класса CScrollBar библиотеки MFC*

Метод	Описание
EnableScrollBar	Включает или выключает полосу прокрутки
ShowScrollBar	Показывает или прячет полосу прокрутки
GetScrollInfo	Извлекает информацию о полосе прокрутки
GetScrollLimit	Извлекает предельное значение полосы прокрутки
GetScrollPos	Извлекает текущее положение бегунка
GetScrollRange	Извлекает минимальное и максимальное значения прокрутки
SetScrollInfo	Устанавливает информацию о полосе прокрутки

Таблица 9.8 *(окончание)*

Метод	Описание
`SetScrollPos`	Устанавливает текущее положение бегунка
`SetScrollRange`	Устанавливает минимальное и максимальное значения прокрутки

9.3.2. Демонстрационная программа UsgScroll: использование полос прокрутки

Демонстрационный проект, иллюстрирующий работу с полосами прокрутки, находится на прилагаемом компакт-диске, в папке \Глава 09\UsgScroll, а файловый состав проекта показан на рис. 9.36.

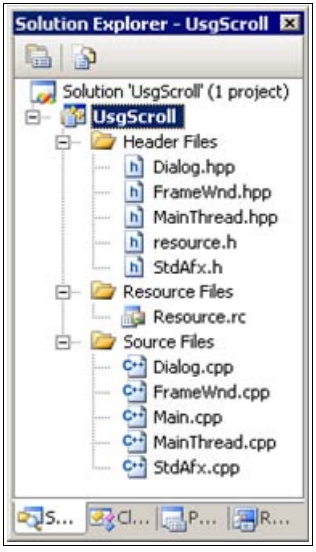


Рис. 9.36. Файловый состав проекта UsgScroll

Совет

Копию проекта UsgScroll для экспериментов создавайте копированием папки проекта с компакт-диска. Файлы StdAfx.h, StdAfx.cpp, Main.cpp, MainThread.hpp, MainThread.cpp, FrameWnd.hpp, FrameWnd.cpp в данном проекте являются, по сути, стандартными или рассмотрены в предыдущих демонстрационных примерах, а файлы Dialog.hpp и Dialog.cpp представлены в листингах 9.13 и 9.14. Файлы resource.h и Resourse.rc созданы редактором ресурсов и поддерживаются им. Ресурсы проекта представлены на рис. 9.37.

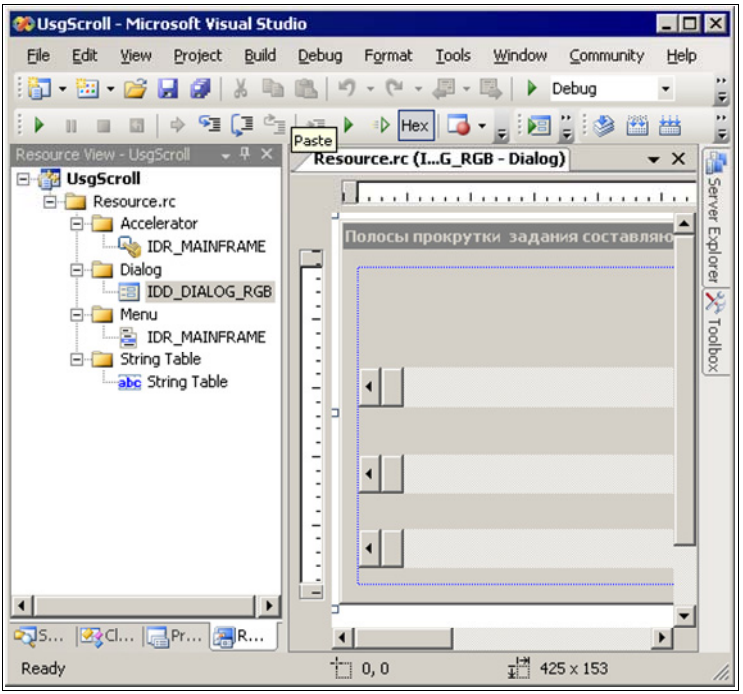


Рис. 9.37. Ресурсы проекта UsgScroll

Листинг 9.13. Файл Dialog.hpp

```
/*
   Файл                 : Dialog.hpp

   Проект               : демонстрация Windows-приложения на основе
                          библиотеки классов MFC с модальным
                          диалоговым окном, использующим полосы прокрутки

   Назначение           : объявление класса "Dialog", производного
                          от класса "CDialod" библиотеки классов
                          MFC

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _Dialog_hpp
    #define _Dialog_hpp
```

```
    // Объявление класса
    class Dialog : public CDialog
    {
        // Данные

    private:

        // Указатели ползунков красного, зеленого и синего цветов
        CScrollBar      *pScrollBarRed;
        CScrollBar      *pScrollBarGreen;
        CScrollBar      *pScrollBarBlue;

    public:

        COLORREF        m_OldRGB;   // Старый RGB-цвет
        COLORREF        m_NewRGB;   // Новый RGB-цвет

        // Методы

    public:

        // Конструктор — передает идентификатор диалога в базовый
        //   класс
        Dialog( CWnd *pParent = 0 );

    protected:

        // Перегружаем виртуальный метод для инициализации диалога
        virtual BOOL OnInitDialog( void );

        // Перегружаем виртуальный метод для получения информации
        //   из диалога
        virtual void OnOK( void );

        // Обработчик сообщений от горизонтальной полосы прокрутки
        afx_msg void OnHScroll( UINT nSBCode, UINT nPos,
                                CScrollBar *pScrollBar );

        // Объявление таблицы сообщений
        DECLARE_MESSAGE_MAP( );
    };

#endif
```

Листинг 9.14. Файл Dialog.cpp

```
/*
   Файл                : Dialog.cpp

   Проект              : демонстрация Windows-приложения на основе
                         библиотеки классов MFC с модальным
                         диалоговым окном, использующим полосы прокрутки

   Назначение          : реализация класса "Dialog", производного
                         от класса "CDialod" библиотеки классов
                         MFC (обработка окна диалога)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAfx.h"                  // Прекомпилируемый файл
#include "FrameWnd.hpp"              // Объявление класса FrameWnd
#include "Dialog.hpp"                // Объявление класса Dialog
// Идентификаторы ресурсов (поддержка редактором ресурсов)
#include "resource.h"

// Определение карты сообщений
BEGIN_MESSAGE_MAP( Dialog, CDialog )
   ON_WM_HSCROLL( )
END_MESSAGE_MAP( )

// Конструктор
Dialog :: Dialog(
    CWnd                *pParent )  // Указатель на родительское
                                    //   окно (NULL — родительским
                                    //   является главное окно)
    : CDialog( IDD_DIALOG_RGB, pParent )
    {
    }

// Инициализация окна диалога
BOOL Dialog :: OnInitDialog( void )
{
    // Вызываем метод базового класса
    CDialog :: OnInitDialog( );
```

```
    BYTE                Red;        // Красная составляющая "m_RGB"
    BYTE                Green;      // Зеленая составляющая "m_RGB"
    BYTE                Blue;       // Синяя составляющая "m_RGB"

    // Получаем указатели на полосы прокрутки красного, зеленого и
    //   синего цветов
    pScrollBarRed   = static_cast< CScrollBar * >
        ( GetDlgItem( IDC_SCROLLBARRED ) );
    pScrollBarGreen   = static_cast< CScrollBar * >
        ( GetDlgItem( IDC_SCROLLBARGREEN ) );
    pScrollBarBlue   = static_cast< CScrollBar * >
        ( GetDlgItem( IDC_SCROLLBARBLUE ) );

    // Устанавливаем диапазоны значений для полос прокрутки
    //   красного, зеленого и синего цветов (обращаем внимание на
    //   то, что можно обрабатывать возвращаемое значение — для
    //   сокращения объема исходного текста мы это не делаем, но вам
    //   предлагается выполнить такую модификацию)
    pScrollBarRed->SetScrollRange( 0, 255 );
    pScrollBarGreen->SetScrollRange( 0, 255 );
    pScrollBarBlue->SetScrollRange( 0, 255 );

    // Выделяем красную, зеленую и синюю составляющие "старого"
    //   RGB-цвета
    Red   = GetRValue( m_OldRGB );
    Green = GetGValue( m_OldRGB );
    Blue  = GetBValue( m_OldRGB );

    // Устанавливаем значения цветов для полос прокрутки
    pScrollBarRed->SetScrollPos( Red );
    pScrollBarGreen->SetScrollPos( Green );
    pScrollBarBlue->SetScrollPos( Blue );

     return TRUE;
}

// Получение информации из диалогового окна (обработчик кнопки OK)
void Dialog :: OnOK( void )
{
    // Вызываем метод базового класса
    CDialog :: OnOK( );
```

```
    BYTE                Red;        // Красная составляющая "m_RGB"
    BYTE                Green;      // Зеленая составляющая "m_RGB"
    BYTE                Blue;       // Синяя составляющая "m_RGB"

    // Получаем текущие значения цветов от полос прокрутки красного,
    //   зеленого и синего цветов
    Red   = pScrollBarRed->GetScrollPos( );
    Green = pScrollBarGreen->GetScrollPos( );
    Blue  = pScrollBarBlue->GetScrollPos( );

    // Формируем новый RGB-цвет
    m_NewRGB = RGB( Red, Green, Blue );

    return;
}

// Обработчик сообщений от горизонтальной полосы прокрутки
afx_msg void Dialog :: OnHScroll(
    // Код уведомления полосы прокрутки
    UINT                nSBCode,
    UINT                nPos,       // Текущая позиция бегунка
    // Указатель на полосу прокрутки
    CScrollBar          *pScrollBar )
{
    int                 nCurPos;    // Позиция бегунка

   // Получаем позицию бегунка
   nCurPos = pScrollBar->GetScrollPos( );

   // Вызываем метод базового класса
   CDialog :: OnHScroll( nSBCode, nPos, pScrollBar );

    // Обрабатываем сообщения полосы прокрутки
    switch ( nSBCode )
    {
    // Нажата правая кнопка
    case SB_LINEDOWN:
        nCurPos += 1;
        break;

    // Нажата левая кнопка
    case SB_LINEUP:
        nCurPos -= 1;
        break;
```

```
    // Нажата кнопка мыши справа от бегунка
    case SB_PAGEDOWN:
        nCurPos += 10;
        break;

    // Нажата кнопка мыши слева от бегунка
    case SB_PAGEUP:
        nCurPos -= 10;
        break;

    // Выполнено протаскивание бегунка
    case SB_THUMBTRACK:
        nCurPos = nPos;
        break;
    }

    // Устанавливаем новое местоположение бегунка
    pScrollBar->SetScrollPos( nCurPos );

    return afx_msg void( );
}
```

Главное окно приложения представлено на рис. 9.38, а модальное диалоговое окно с полосами прокрутки показано на рис. 9.39.

Рис. 9.38. Главное окно проекта UsgScroll

В листингах 9.13 и 9.14 следует, прежде всего, обратить внимание на таблицу сообщений и методы их обработки. Поскольку в демонстрационной программе используются только горизонтальные полосы прокрутки, то в таблице сообщений достаточно лишь использовать макрос `ON_WM_HSCROLL`. Также обратите внимание на использование методов `SetScrollRange`, `SetScrollPos` и `GetScrollPos` (см. табл. 9.8, листинг 9.14) и обработчик сообщений горизонтальных полос прокрутки `OnHScroll`. Остальная часть программного кода представляется ясной. Поэтому далее также ограничимся лишь рассмотрени-

ем шаблона ресурсов диалогового окна и полос прокрутки в части создания и конфигурирования горизонтальных полос прокрутки.

Рис. 9.39. Модальный диалог с полосами прокрутки проекта UsgScroll

Для размещения в диалоговом окне полос прокрутки можно воспользоваться панелью **Toolbox** (см. рис. 8.2) — выберите горизонтальную полосу прокрутки (Horizontal Scroll Bar) и перетащите ее в нужное место диалогового окна. Для демонстрационного примера их размеры и расположение имеют вид, показанный на рис. 9.39.

Для настройки свойств любого управляющего элемента диалогового окна вызовите контекстное меню и выберите **Properties**. Свойства горизон-

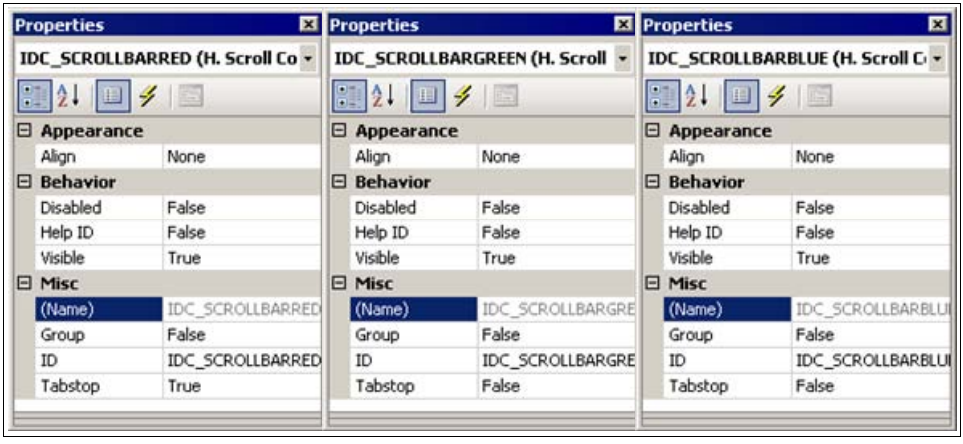


Рис. 9.40. Свойства горизонтальных полос прокрутки шаблона ресурсов диалогового окна проекта UsgScroll

тальных полос прокрутки для задания составляющих цвета для демонстрационного примера показаны на рис. 9.40.

Совет

Обратите внимание на идентификаторы горизонтальных полос прокрутки (`IDC_SCROLLBARRED`, `IDC_SCROLLBARGREEN` и `IDC_SCROLLBARBLUE`) и их использование в исходном коде (листинг 9.14). Изучите свойства горизонтальных полос прокрутки. Используя копию проекта UsgScroll, поэкспериментируйте, изменяя значения свойств.

9.4. Ползунок. Класс *CSliderCtrl* библиотеки MFC

Подобно полосе прокрутки, ползунок — это интерактивный визуальный элемент управления, состоящий из перемещаемой ручки и соответствующей разметки, отмечающей интервалы изменений (рис. 9.41).

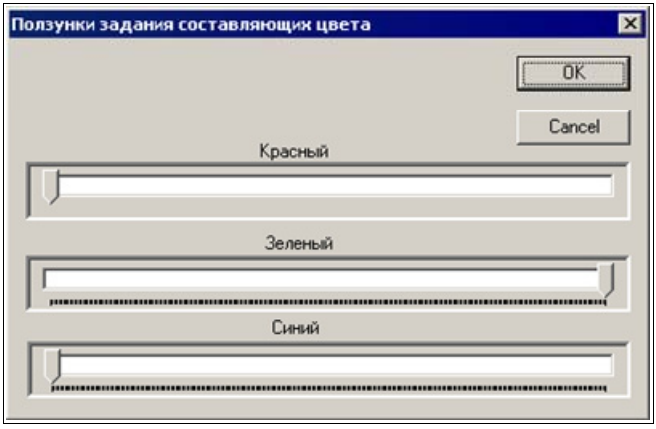


Рис. 9.41. Пример ползунков задания составляющих цвета проекта UsgSlider

Ползунок может быть расположен горизонтально или вертикально. При перемещении ручки в одном или другом направлении происходит уменьшение или увеличение значения ползунка соответственно. Максимальное и минимальное значения ползунка устанавливаются программно.

Все функциональные возможности ползунка определены в классе `CSliderCtrl` библиотеки MFC. Класс `CSliderCtrl` является производным от класса `CWnd` и наследует все его возможности.

Ползунок может быть создан либо программно, либо размещен в шаблоне ресурсов диалогового окна. Ползунок посылает сообщения своему окну-родителю, которые могут быть перехвачены и обработаны.

9.4.1. Стили, сообщения ползунка и методы класса *CSliderControl*

Ползунок может использовать элементарные стили, доступные всем объектам класса `CWnd`, а также ряд специальных стилей (табл. 9.9). Стиль ползунка определяет его внешний вид и поведение. Стиль устанавливается либо при инициализации методом `CSliderCtrl::Create`, либо в шаблоне ресурсов диалогового окна, в свойствах ползунка.

Таблица 9.9. *Элементарные стили ползунка*

Макроопределение стиля	Значение
`TBS_AUTOTICKS`, свойство **Auto Ticks** равно **True**	Задает разметку с делениями, соответствующими единице изменения значения ползунка
`TBS_BOTH`, свойство **Point** равно **Both**	Размечает обе стороны ползунка, вне зависимости от его ориентации
`TBS_BOTTOM`, свойство **Point** равно **Bottom/Right**	Помещает разметку на нижней границе горизонтального ползунка
`TBS_ENABLESELRANGE`, свойство **Enable Selection** равно **True**	Создает особые метки ползунка в форме треугольника, которые обозначают начало и конец выделенного диапазона
`TBS_HORZ`, свойство **Orientation** равно **Horizontal**	Ориентирует ползунок горизонтально (по умолчанию)
`TBS_LEFT`, свойство **Point** равно **Top/Left**	Помещает разметку на левой границе вертикального ползунка
`TBS_NOTICKS`, свойство **Tick Marks** равно **False**	Ползунок не имеет разметки
`TBS_RIGHT`, свойство **Point** равно **Bottom/Right**	Помещает разметку на правой границе вертикального ползунка
`TBS_TOP`, свойство **Point** равно **Top/Left**	Помещает разметку на верхней границе горизонтального ползунка
`TBS_VERT`, свойство **Orientation** равно **Vertical**	Ориентирует ползунок

Примечание

Все макросы стилей ползунка начинаются с префикса `TBS_` (от Track Bar Styles).

Сообщения ползунка могут быть перехвачены и обработаны, если существуют соответствующие элементы таблицы сообщений и методы их обработки. Элементы таблицы сообщений для ползунка приведены в табл. 9.10.

Таблица 9.10. *Элементы таблицы сообщений для ползунка*

Элемент таблицы сообщений	Значение
ON_WM_HSCROLL	Посылается ползунком, имеющим стиль TBS_HORZ при перемещении его ручки с помощью мыши, клавиш-стрелок, нажатии клавиш <Home> или <End>, или щелчках мыши по полосе ползунка слева или справа от его ручки
ON_WM_VSCROLL	Посылается ползунком, имеющим стиль TBS_VERT при перемещении его ручки с помощью мыши, клавиш-стрелок, нажатии клавиш <Home> или <End>, или щелчках мыши по полосе ползунка сверху или снизу от его ручки

Как и в случае полосы прокрутки, сообщения WM_HSCROLL и WM_VSCROLL содержат в себе скрытую информацию. Для доступа к этой информации воспользуйтесь методами OnHScroll и OnVScroll, вызываемыми при перемещении ручки (бегунка) с помощью мыши, клавиш-стрелок, нажатием клавиш <Home> или <End>, или щелкая мышью по полосе ползунка. Прототипы методов обработки сообщений от горизонтального и вертикального ползунков имеют вид:

```
// Обработчик сообщений от горизонтальной полосы прокрутки
afx_msg void OnHScroll( UINT nSBCode, UINT nPos,
                        CSliderControl *pScrollBar );
// Обработчик сообщений от вертикальной полосы прокрутки
afx_msg void OnVScroll( UINT nSBCode, UINT nPos,
                        CSliderControl *pScrollBar );
```

Первый параметр — nSBCode — это один из возможных кодов уведомлений ползунка (табл. 9.11), сообщающий приложению, какие действия совершает пользователь с ползунком.

Таблица 9.11. *Коды уведомлений ползунка, посылаемые методам* OnHScroll *и* OnVScroll

Код уведомления	Значение
TB_BOTTON	Нажата клавиша <End>
TB_LINEDOWN	Нажата клавиша "стрелка вниз" или клавиша "стрелка вправо"
TB_LINEUP	Нажата клавиша "стрелка вверх" или клавиша "стрелка влево"

Таблица 9.11 *(окончание)*

Код уведомления	Значение
TB_PAGEDOWN	Выполнен щелчок мышью справа или снизу от бегунка или нажата клавиша <Page Down>
TB_PAGEUP	Выполнен щелчок мышью слева или сверху от бегунка или нажата клавиша <Page Up>
TB_THUMBTRACK	Выполнено перемещение бегунка
TB_TOP	Нажата клавиша <Home>

Второй параметр методов `OnHScroll` и `OnVScroll` — `nPos` — задает новую позицию бегунка после его перемещения.

Третий параметр — `pScrollBar` — задает указатель на ползунок, который посылает сообщение.

При перегрузке методов `OnHScroll` и `OnVScroll` необходимо привести этот указатель к типу `CSliderCtrl`:

```
afx_msg void OnHScroll( UINT nSBCode, UINT nPos,
                        CSliderControl *pScrollBar )
{
    CSliderCtrl         *pSlider = (CSliderCtrl * )pScrollBar;

    // Использовать pSlider
}
```

Примечание

Методы `OnHScroll` и `OnVScroll` можно, вообще говоря, не перегружать в классе, производном от класса `CDialog` библиотеки MFC. В этом случае будут использованы методы базового класса `CDialog`. Приводимый далее пример демонстрирует такую возможность.

Методы класса `CSliderCtrl`, перечисленные в табл. 9.12, описывают способы получения (get) и установки (set) свойств ползунка.

Таблица 9.12. *Методы класса* `CSliderCtrl`

Метод	Описание
GetCannelRect	Возвращает размер полосы ползунка
GetLineSize	Возвращает величину изменения значения ползунка при нажатии клавиш-стрелок

Таблица 9.12 *(окончание)*

Метод	Описание
GetNumTics	Возвращает число линий разметки ползунка
GetPageSize	Возвращает величину изменения значения ползунка при нажатии клавиш <Page Up> и <Page Down>
GetPos	Возвращает текущее положение бегунка
GetRange	Возвращает минимальное и максимальное значения ползунка
GetRangeMax	Возвращает максимальное значение ползунка
GetRangeMin	Возвращает минимальное значение ползунка
GetSelection	Возвращает текущий выбранный диапазон
GetThumbRect	Возвращает размер бегунка
GetTic	Возвращает положение указанной линии разметки
GetTicArray	Возвращает массив положений линий разметки ползунка
GetTicPos	Возвращает положение указанной линии разметки в координатах клиентской области окна
SetLineSize	Устанавливает величину изменения значения ползунка при нажатии клавиш-стрелок
SetPageSize	Устанавливает величину изменения значения ползунка при нажатии клавиш <Page Up> и <Page Down>
SetPos	Устанавливает текущее положение бегунка
SetRange	Устанавливает минимальное и максимальное значения ползунка
SetRangeMax	Устанавливает максимальное значение ползунка
SetRangeMin	Устанавливает минимальное значение ползунка
SetSelection	Устанавливает текущий выбранный диапазон
SetTic	Устанавливает положение указанной линии разметки
SetTicFreq	Устанавливает частоту линий разметки — их число на единицу изменения значения ползунка
ClearSel	Снимает выделение выбранного диапазона ползунка
ClearTics	Удаляет разметку ползунка
VerifyPos	Проверяет, находился ли бегунок в нулевой позиции

9.4.2. Демонстрационная программа UsgSlider: использование ползунков

Демонстрационный проект, иллюстрирующий работу с полосами прокрутки, находится на прилагаемом компакт-диске, в папке \Глава 09\UsgSlider, а файловый состав проекта показан на рис. 9.42.

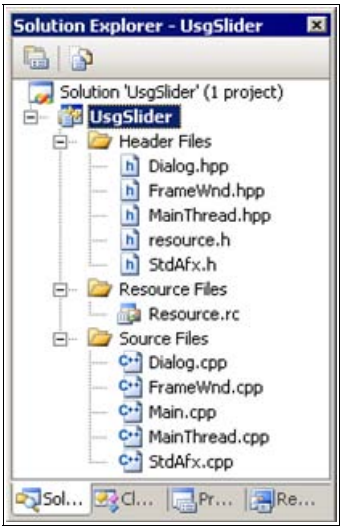


Рис. 9.42. Файловый состав проекта UsgSlider

Совет

Копию проекта UsgSlider создавайте копированием папки проекта с компакт-диска. Файлы StdAfx.h, StdAfx.cpp, Main.cpp, MainThread.hpp, MainThread.cpp, FrameWnd.hpp, FrameWnd.cpp в данном проекте являются, по сути, стандартными или рассмотрены в предыдущих демонстрационных примерах, а файлы Dialog.hpp и Dialog.cpp представлены в листингах 9.15 и 9.16. Файлы resource.h и Resourse.rc созданы редактором ресурсов и поддерживаются им. Ресурсы проекта представлены на рис. 9.43.

Листинг 9.15. Файл Dialog.hpp

```
/*
   Файл                : Dialog.hpp

   Проект              : демонстрация Windows-приложения на основе
                         библиотеки классов MFC с модальным
                         диалоговым окном, использующим ползунки
```

```
   Назначение          : объявление класса "Dialog", производного
                         от класса "CDialod" библиотеки классов
                         MFC

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _Dialog_hpp
    #define _Dialog_hpp

    // Объявление класса
    class Dialog : public CDialog
    {
        // Данные

    private:

        // Указатели ползунков красного, зеленого и синего цветов
        CSliderCtrl     *pSliderRed;
        CSliderCtrl     *pSliderGreen;
        CSliderCtrl     *pSliderBlue;

    public:

        COLORREF        m_OldRGB;   // Старый RGB-цвет
        COLORREF        m_NewRGB;   // Новый RGB-цвет

        // Методы

    public:

        // Конструктор — передает идентификатор диалогового окна
        //   в базовый класс
        Dialog( CWnd *pParent = 0 );

    protected:

        // Перегружаем виртуальный метод для инициализации
        virtual BOOL OnInitDialog( void );
```

```
        // Перегружаем виртуальный метод для получения информации
        //   из диалога
        virtual void OnOK( void );

    };

#endif
```

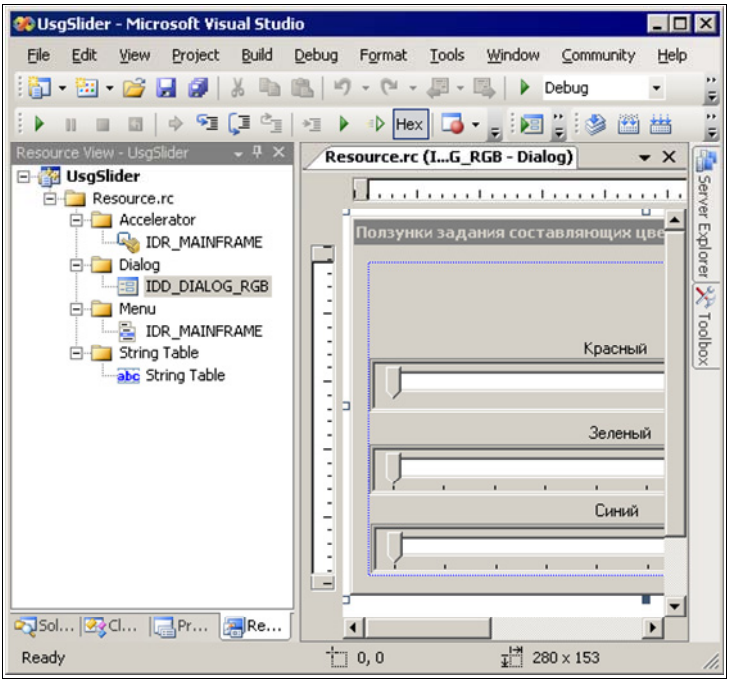


Рис. 9.43. Ресурсы проекта UsgSlider

Листинг 9.16. Файл Dialog.cpp

```
/*
   Файл                 : Dialog.cpp

   Проект               : демонстрация Windows-приложения на основе
                          библиотеки классов MFC с модальным
                          диалоговым окном, использующим ползунки

   Назначение           : реализация класса "Dialog", производного
                          от класса "CDialod" библиотеки классов
                          MFC (обработка окна диалога)
```

```
   Microsoft Visual Studio C++ .NET 2005
*/


#include "StdAfx.h"                  // Прекомпилируемый файл
#include "FrameWnd.hpp"              // Объявление класса FrameWnd
#include "Dialog.hpp"                // Объявление класса Dialog
// Идентификаторы ресурсов (поддержка редактором ресурсов)
#include "resource.h"

// Конструктор
Dialog :: Dialog(
    CWnd                *pParent )   // Указатель на родительское
                                     //   окно (NULL — родительским
                                     //   является главное окно)
    : CDialog( IDD_DIALOG_RGB, pParent )
    {
    }

// Инициализация окна диалога
BOOL Dialog :: OnInitDialog( void )
{
    // Вызываем метод базового класса
    CDialog :: OnInitDialog( );

    BYTE                Red;         // Красная составляющая "m_RGB"
    BYTE                Green;       // Зеленая составляющая "m_RGB"
    BYTE                Blue;        // Синяя составляющая "m_RGB"

    // Получаем указатели на ползунки красного, зеленого и синего
    //   цветов
    pSliderRed   = static_cast< CSliderCtrl * >
        ( GetDlgItem( IDC_SLIDERR ) );
    pSliderGreen   = static_cast< CSliderCtrl * >
        ( GetDlgItem( IDC_SLIDERG ) );
    pSliderBlue   = static_cast< CSliderCtrl * >
        ( GetDlgItem( IDC_SLIDERB ) );

    // Устанавливаем диапазоны значений для ползунков красного,
    //   зеленого и синего цветов (обращаем внимание на то, что
    //   можно обрабатывать возвращаемое значение — для сокращения
    //   объема исходного текста мы это не делаем, но вам
    //   предлагается выполнить такую модификацию)
```

```
    pSliderRed->SetRange( 0, 255 );
    pSliderGreen->SetRange( 0, 255 );
    pSliderBlue->SetRange( 0, 255 );

    // Выделяем красную, зеленую и синюю составляющие "старого"
    //   RGB-цвета
    Red   = GetRValue( m_OldRGB );
    Green = GetGValue( m_OldRGB );
    Blue  = GetBValue( m_OldRGB );

    // Устанавливаем значения цветов для ползунков
    pSliderRed->SetPos( Red );
    pSliderGreen->SetPos( Green );
    pSliderBlue->SetPos( Blue );

     return TRUE;
}

// Получение информации из окна (обработчик кнопки OK)
void Dialog :: OnOK( void )
{
    // Вызываем метод базового класса
    CDialog :: OnOK( );

    BYTE                Red;        // Красная составляющая "m_RGB"
    BYTE                Green;      // Зеленая составляющая "m_RGB"
    BYTE                Blue;       // Синяя составляющая "m_RGB"

    // Получаем текущие значения цветов от ползунков красного,
    //   зеленого и синего цветов
    Red   = pSliderRed->GetPos( );
    Green = pSliderGreen->GetPos( );
    Blue  = pSliderBlue->GetPos( );

    // Формируем новый RGB-цвет
    m_NewRGB = RGB( Red, Green, Blue );

    return;
}
```

Главное окно приложения представлено на рис. 9.44, а модальное диалоговое окно с полосами прокрутки показано на рис. 9.41.

В листингах 9.15 и 9.16 следует, прежде всего, обратить внимание на отсутствие таблицы сообщений и методов их обработки, о чем говорилось ранее

(см. разд. 9.4.1). Обратите также внимание на использование методов `SetRange`, `SetPos` и `GetPos` (см. табл. 9.12, листинг 9.16) и обработчик сообщений горизонтальных полос прокрутки `OnHScroll`. Остальная часть программного кода представляется ясной. Поэтому далее рассмотрим шаблона ресурсов диалогового окна и ползунков в части создания и конфигурирования горизонтальных ползунков.

Рис. 9.44. Главное окно проекта UsgSlider

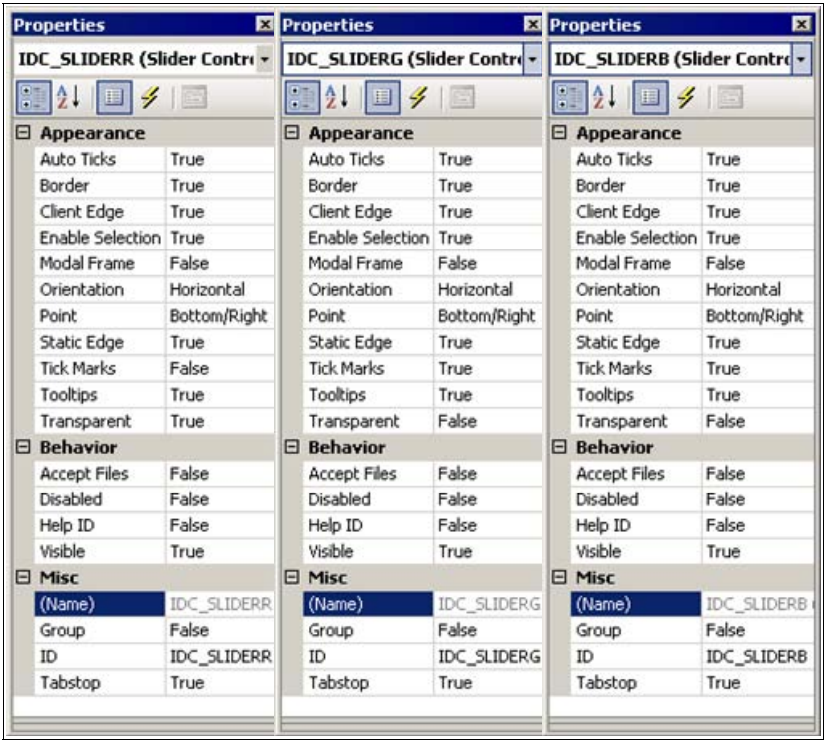


Рис. 9.45. Свойства горизонтальных ползунков шаблона ресурсов диалогового окна проекта UsgSlider

Для размещения ползунка в диалоговом окне воспользуйтесь панелью **Toolbox** (см. рис. 8.2), выберите ползунок (Slider Control) и перетащите его в нужное место диалогового окна. Для демонстрационного примера размеры и расположение ползунков имеют вид, показанный на рис. 9.43.

Для настройки свойств ползунка выберите пункт **Properties** в контекстном меню и задайте свойства элемента. Свойства горизонтальных ползунков для задания составляющих цвета для демонстрационного примера показаны на рис. 9.45.

Совет

Обратите внимание на идентификаторы горизонтальных ползунков (`IDC_SLIDERR`, `IDC_SLIDERG` и `IDC_SLIDERB`) и их использование в исходном коде (листинг 9.16). Изучите свойства горизонтальных ползунков. Используя копию проекта UsgSlider, поэкспериментируйте, изменяя значения указанных свойств.

9.5. Вопросы и упражнения для самопроверки

Для самопроверки ответьте на следующие вопросы:

1. Какие классы библиотеки MFC поддерживают работу с кнопками?
2. Перечислите основные виды кнопок.
3. Что представляет собой обычная кнопка?
4. Укажите способы применения кнопок.
5. Какие элементарные стили можно использовать для кнопок?
6. Укажите основные элементы таблицы сообщений от кнопок.
7. В чем состоит отличительная особенность отмечаемых кнопок?
8. Перечислите и охарактеризуйте разновидности отмечаемых кнопок.
9. Как разместить отмечаемую кнопку в диалоговом окне?
10. Как настроить свойства отмечаемой кнопки?
11. Что собой представляет кнопка с растровым изображением? Какой класс библиотеки MFC предназначен для работы с рисованными кнопками?
12. Как связать растровую кнопку с изображением?
13. Как из одного приложения запустить другое приложение?
14. Как включить в проект Bitmap-ресурс и задать его свойства?
15. Как создать изображение растровой кнопки в отжатом и нажатом состояниях?

16. Какие ограничения накладывает метод `AutoLoad` при связывании с изображениями?
17. Что представляет собой спин (spin) с дружественным окном?
18. В каком классе библиотеки MFC определены функциональные возможности вращателя?
19. Перечислите элементарные стили спина и дайте им характеристику.
20. Перечислите элементы таблицы сообщений для спина и дайте им характеристику.
21. Перечислите и охарактеризуйте основные методы класса `CSpinButtonCtrl` библиотеки MFC.
22. Что представляет собой полоса прокрутки?
23. В каком классе библиотеки MFC определены функциональные возможности полосы прокрутки?
24. Перечислите элементарные стили полосы прокрутки и дайте им характеристику.
25. Опишите прототипы методов обработки сообщений от полос прокрутки.
26. Перечислите и охарактеризуйте основные методы класса `CScrollBar` библиотеки MFC.
27. Что представляет собой ползунок?
28. В каком классе библиотеки MFC определены функциональные возможности ползунка?
29. Перечислите элементарные стили ползунка и дайте им характеристику.
30. Что обозначает префикс `TBS_`, используемый в макросах элементарных стилей ползунка?
31. Опишите прототипы методов обработки сообщений от ползунков.
32. Обязательно ли перегружать методы обработки сообщений от ползунков?
33. Перечислите и охарактеризуйте основные методы класса `CSliderCtrl` библиотеки MFC.

Ответы на вопросы можно проверить — они приведены в *разд. П1.9 приложения 1*.

При экспериментах с созданной копией демонстрационного проекта UsgGroupBox выполните следующие упражнения:

1. Попробуйте менять свойства элементов группировки и посмотрите, на что это влияет.
2. Попробуйте вместо двух групп радиокнопок создать только одну.

При экспериментах с созданной копией демонстрационного проекта UsgCheckBox выполните следующие упражнения:

1. Попробуйте менять свойства отмечаемых кнопок и посмотрите, на что это влияет.
2. По полученным результатам опишите действие свойств отмечаемой кнопки.

При экспериментах с созданной копией демонстрационного проекта UsgBtnBmp выполните следующие упражнения:

1. Попробуйте менять свойства рисованных кнопок и изображений (Bitmap-ресурсов). Посмотрите, на что это влияет.
2. Оставьте одну рисованную кнопку и запустите с ее помощью какое-либо приложение.

При экспериментах с созданной копией демонстрационного проекта UsgSpinEdit выполните следующие упражнения:

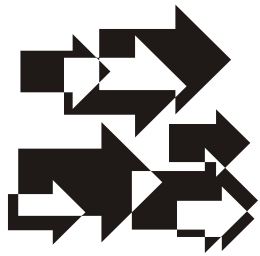
1. Попробуйте менять свойства спина. Посмотрите, на что это влияет.
2. Попробуйте менять свойства дружественных элементов редактирования. Посмотрите, на что это влияет.
3. Добавьте в диалоговое окно еще один спин с дружественным элементом редактирования и добейтесь его функционирования в диалоговом окне.

При экспериментах с созданной копией демонстрационного проекта UsgScroll выполните следующие упражнения:

1. Попробуйте менять свойства горизонтальных полос прокрутки. Посмотрите, на что это влияет.
2. Попробуйте заменить горизонтальные полосы прокрутки на вертикальные.
3. Добавьте в диалоговое окно еще одну полосу прокрутки и добейтесь ее функционирования.

При экспериментах с созданной копией демонстрационного проекта UsgScroll выполните следующие упражнения:

1. Попробуйте менять свойства ползунков. Посмотрите, на что это влияет.
2. Попробуйте заменить горизонтальные ползунки на вертикальные.
3. Добавьте в диалоговое окно еще один ползунок и добейтесь его функционирования.



ГЛАВА 10

Элементы управления: список, комбинированный список, индикатор прогресса и таймер [5]

Новым в этой главе является использование редактора ресурсов для создания и настройки в диалоговых окнах таких широко применяемых элементов управления, как список, комбинированный список, индикатор прогресса и таймер.

10.1. Список. Класс *CListBox* библиотеки MFC

Список является удобным средством для выбора одного или несколько элементов. Список представляет собой прямоугольное окно, в котором обычно находится несколько строк. Это широко распространенный стандартный элемент управления, позволяющий выбрать. Списки могут иметь собственные полосы прокрутки.

Существуют две разновидности списков:

- список с возможностью выбора одного элемента (список единичного выбора);
- список с возможностью выбора нескольких элементов (список множественного выбора).

Библиотека MFC обеспечивает все функциональные возможности стандартного списка с помощью класса `CListBox`. В соответствии с иерархией классов библиотеки MFC, класс `CListBox` является производным от класса `CWnd`, а это означает, что список — тоже окно, которое наследует некоторые возможности объекта `CWnd`.

Класс `CListBox` имеет набор методов, упрощающих работу со списком. Списки могут быть созданы, как подчиненные элементы управления любого ок-

на, непосредственно в коде, но, как правило, они определяются с помощью редактора ресурсов в шаблоне ресурсов диалогового окна.

По списку можно перемещаться как на один элемент с помощью клавиш-стрелок, так и на одну страницу с помощью клавиш <Page Up> и <Page Down>. Если для списка установлен соответствующий стиль, список позволяет выбирать несколько элементов с помощью сочетаний клавиш <Ctrl> (или <Shift>) и кнопки мыши. Каждый элемент в списке имеет свой индекс, используемый для обращения к этому элементу, причем, обратите на это внимание, — индексация начинается с нуля.

10.1.1. Стили, сообщения списка и методы класса *CListBox*

Списки могут использовать элементарные стили, доступные всем объектам класса `CWnd`, а также ряд специальных стилей, представленных в табл. 10.1. Стиль списка определяет его внешний вид и поведение. Стиль устанавливается либо при инициализации списка методом `CListBox::Create`, либо заданием его свойств в редакторе ресурсов.

Таблица 10.1. *Стили списка*

Стиль	Описание
`LBS_DISABLENOSCROLL`, свойство **Disable No Scroll** равно **True**	Запрещает использование вертикальной полосы прокрутки списка, не убирая ее в том случае, когда все элементы списка помещаются в окне
`LBS_EXTENDEDSEL`, свойство **Selection** равно **Extended**	Список с возможностью выбора нескольких элементов, но с более широкими возможностями. Используется в сочетании со стилем `LBS_MULTIPLESEL`
`LBS_HASSTRING`, свойство **Has String** равно **True**	Задает список, содержащий строковые элементы. Список сам выделяет память под строки. Текст конкретного элемента списка может быть извлечен с помощью метода `GetText`
`LBS_MULTICOLUMN`, свойство **Multicolumn** равно **True**	Задает список с несколькими колонками и горизонтальной полосой прокрутки. Можно воспользоваться методом `GetColumnWidth`, чтобы задать ширину колонок
`LBS_MULTIPLESEL`, свойство **Selection** равно **Multiple**	Задает список с возможностью выбора нескольких элементов
`LBS_NOINTEGRALHEIGHT`, свойство **No Integral Height** равно **True**	Определяет точные размеры окна списка, указанные при его создании. При этом элемент списка, не помещающийся в окно, будет отображаться частично

Таблица 10.1 *(окончание)*

Стиль	Описание
LBS_NOREDRAW, свойство **No Redraw** равно **True**	Список не перерисовывается при изменениях. Можно в любой момент изменить этот стиль, послав сообщение WM_SETREDRAW
LBS_NOSEL, свойство **Selection** равно **None**	Элементы списка можно просматривать, но нельзя выбирать
LBS_NOTIFY, свойство **Notify** равно **True**	Список посылает сообщение родительскому окну, когда пользователь производит одинарный или двойной щелчок на элементе списка
LBS_OWNERDRAWFIXED, свойство **Owner Draw** равно **Fixed**	Изображение элементов списка осуществляется пользователем. Элементы списка имеют одинаковую высоту
LBS_OWNERDRAWVARIABLE, свойство **Owner Draw** равно **Variable**	Изображение элементов списка осуществляется пользователем. Элементы списка могут быть разной высоты
LBS_SORT, свойство **Sort** равно **True**	Задает список, элементы которого сортируются в алфавитном порядке
LBS_STANDARD	Стандартный список с комбинированным стилем LBS_NOTIFY | LBS_SORT | LBS_VSCROLL | WS_BORDER
LBS_USETABSTOPS, свойство **Tabstop** равно **True**	Задает список, расширяющий символы табуляции при отображении элементов. Позиции табуляции по умолчанию устанавливаются в 32 единицы
LBS_WANTKEYBOARDINPUT, свойство **Want Key Input** равно **True**	Задает список, расширяющий символы табуляции при отображении элементов. Позиции табуляции по умолчанию устанавливаются в 32 единицы

Примечание

Все макросы стилей списка начинаются с префикса LBS_ (от List Box Styles).

Библиотека MFC абстрагирует сообщения стандартного списка, такие как LB_INSERTSTRING и LB_ADDSTRING, методами класса CListBox и приложению приходится обрабатывать только уведомления (и то не всегда). Список, имеющий стиль LBS_NOTIFY, посылает уведомления родительскому окну (обычно это объект класса, производного от класса CDialog библиотеки MFC). Эти сообщения списка могут быть перехвачены и обработаны, если существуют соответствующие элементы таблицы сообщений и методы их обработки. Элементы карты сообщений для списка приведены в табл. 10.2.

Таблица 10.2. *Элементы карты сообщений для списка*

Элемент таблицы сообщений	Значение
ON_LBN_DBLCLK	Посылается списком, имеющим стиль LBS_NOTIFY при двойном щелчке по элементу списка
ON_LBN_ERRSPACE	Посылается списком, если операция не выполнена из-за недостатка памяти
ON_LBN_KILLFORUS	Посылается, когда список теряет фокус ввода
ON_LBN_SELCANCEL	Посылается списком, имеющим стиль LBS_NOTIFY, когда отменяется текущий выбор элементов списка
ON_LBN_SELCHANGE	Посылается списком, имеющим стиль LBS_NOTIFY, когда изменяется выбор элементов списка
ON_LBN_SETFORUS	Посылается, когда список попадает в фокус ввода

Методы класса CListBox можно разделить на методы общего назначения, единичного выбора, множественного выбора и виртуальные методы.

Методы общего назначения, перечисленные в табл. 10.3, описывают способы получения (get) и установки (set) значений данных и свойств списка.

Таблица 10.3. *Методы общего назначения класса* CListBox

Метод	Описание
GetCount	Возвращает количество элементов в списке
GetHorizontalExtent	Возвращает ширину горизонтальной прокрутки списка (в пикселах)
GetItemData	Возвращает 32-разрядное значение, связанное с элементом списка
GetItemDataPtr	Возвращает указатель на элемент списка
GetItemHeight	Возвращает высоту элементов в списке
GetItemRect	Возвращает размеры прямоугольника списка
GetLocale	Возвращает локальный идентификатор (LCID) списка
GetSel	Возвращает индекс выбранного элемента
GetText	Копирует строку списка в буфер

Таблица 10.3 *(окончание)*

Метод	Описание
`GetTextLen`	Возвращает длину строки списка в байтах
`GetTopIndex`	Возвращает индекс первого видимого элемента списка
`SetColumnWidth`	Устанавливает ширину колонки списка с несколькими колонками
`SetHorizontalExtent`	Устанавливает ширину горизонтальной прокрутки списка в пикселах
`SetItemData`	Устанавливает 32-разрядное значение, связанное с элементом списка
`SetItemDataPtr`	Устанавливает указатель на элемент списка
`SetItemHeight`	Устанавливает высоту элементов в списке
`SetLocale`	Устанавливает локальный идентификатор (LCID) списка
`SetTabStops`	Устанавливает позиции табуляции в списке
`SetTopIndex`	Устанавливает индекс первого видимого элемента списка

Примечание

Методы `GetLocale` и `SetLocale` предназначаны для манипулирования локальными установками.

Список по умолчанию — это список единичного выбора, к которому применимы все методы общего назначения. Однако для списков единичного выбора существуют еще два метода, применимых только для них: для извлечения индекса выбранного элемента — `GetCurSel`, и выбора элемента — `SetCurSel`.

Списки множественного выбора расширяют возможности списка единичного выбора за счет использования дополнительных методов, приведенных в табл. 10.4.

Таблица 10.4. *Специальные методы класса* `CListBox` *для списков множественного выбора*

Метод	Описание
`GetAnchorIndex`	Возвращает индекс начального элемента в списке множественного выбора
`GetCaretIndex`	Возвращает индекс элемента в списке множественного выбора, который находится в фокусе

Таблица 10.4 *(окончание)*

Метод	Описание
`GetSelCount`	Возвращает число выбранных элементов списка множественного выбора
`GetSelItems`	Помещает индексы всех выбранных элементов списка множественного выбора в массив
`SetItemRange`	Переключает состояние выбран\не выбран диапазона элементов в списке множественного выбора
`SetAnchorIndex`	Устанавливает индекс начального элемента в списке множественного выбора
`SetCaretIndex`	Помещает фокус на элемент с указанным индексом
`SetSel`	Выбирает или отменяет выбор элемента списка множественного выбора

Специальные методы работы со строками (табл. 10.5) применимы как к спискам единичного выбора, так и к спискам множественного выбора.

Таблица 10.5. *Специальные методы для работы со строками класса* `CListBox`

Метод	Описание
`AddString`	Добавляет строку в список
`DeleteString`	Удаляет строку из списка
`Dir`	Добавляет имена файлов текущего каталога в список
`FindString`	Ищет строку в списке
`FindStringExact`	Ищет первую в списке строку, совпадающую со строкой поиска
`InsertString`	Вставляет в список строку в позицию с указанным индексом
`ResetContent`	Удаляет из списка все элементы
`SelectString`	Ищет и выделяет строку в списке единичного выбора

В классе `CListBox` имеется несколько виртуальных методов, которые можно перегрузить в производных классах (табл. 10.6). Перегрузка виртуальных методов позволит реализовать дополнительные возможности, не поддерживаемые библиотекой MFC.

Таблица 10.6. *Виртуальные методы класса* `CListBox`

Метод	Описание
`CharToItem`	Метод можно перегрузить, чтобы реализовать собственную обработку сообщения `WM_CHAR` для списков, не содержащих строки и имеющих стиль `LBS_OWNERDRAW`
`CompareItem`	Вызывается для того, чтобы определить положение нового элемента в отсортированном списке, имеющем стиль `LBS_OWNERDRAW`
`DeleteItem`	Вызывается в том случае, когда удаляется элемент из списка, имеющего стиль `LBS_OWNERDRAW`
`DrawItem`	Вызывается при перерисовке списка, имеющего стиль `LBS_OWNERDRAW`
`MeasureItem`	Вызывается при создании списка, имеющего стиль `LBS_OWNERDRAW`, чтобы определить размеры элементов списка
`VKeyToItem`	Метод можно перегрузить, чтобы реализовать собственную обработку сообщения `WM_KEYDOWN` для списков, у которых установлен стиль `LBS_WANTKEYBOARDINPUT`

10.1.2. Демонстрационная программа UsgListBoxes: стандартные списки

Демонстрационный проект, иллюстрирующий работу стандартных списков, находится на прилагаемом компакт-диске, в папке ..\Глава 10\UsgListBoxes, а файловый состав проекта показан на рис. 10.1.

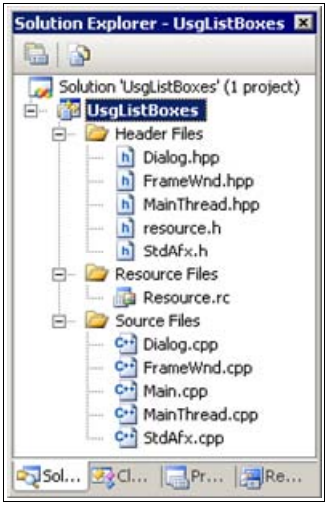


Рис. 10.1. Файловый состав проекта UsgListBoxes

Совет

Копию проекта UsgListBoxes для экспериментов создавайте копированием папки проекта с компакт-диска. Файлы StdAfx.h, StdAfx.cpp, Main.cpp, MainThread.hpp, MainThread.cpp, FrameWnd.hpp, FrameWnd.cpp в данном проекте являются, по сути, стандартными или рассмотрены в предыдущих демонстрационных примерах, а файлы Dialog.hpp и Dialog.cpp представлены в листингах 10.1—10.2. Оставшиеся файлы — resource.h, Resourse.rc — созданы редактором ресурсов и поддерживаются им. Ресурсы проекта представлены на рис. 10.2.

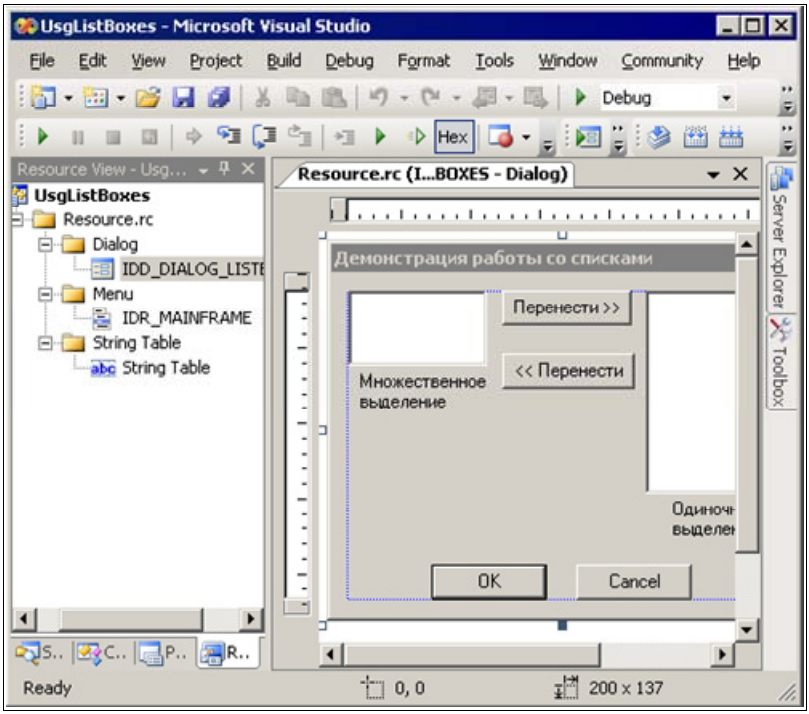


Рис. 10.2. Ресурсы проекта UsgListBoxes

Листинг 10.1. Файл Dialog.hpp

```
/*
   Файл                : Dialog.hpp

   Проект              : демонстрация Windows-приложения на основе
                         библиотеки классов MFC с модальным
                         диалоговым окном, использующим списки
```

```
   Назначение         : объявление класса "Dialog", производного
                        от класса "CDialod" библиотеки классов
                        MFC (обработка сообщений окна)

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _Dialog_hpp
    #define _Dialog_hpp

    // Объявление класса
    class Dialog : public CDialog
    {
        // Данные

    private:

        // Указатель на левый список (MULTI)
        CListBox        *m_plbMulti;
        // Указатель на правый список (SINGLE)
        CListBox        *m_plbSingle;

    public:

        // Переменные, связанные со списками

        // Левый список
        // Указатель на массив строк
        CString         *m_Enabled;
        // Индекс последней строки
        int             m_IxEnbLast;

        // Правый список
        // Указатель на массив строк
        CString         *m_Disabled;
        // Индекс последней строки
        int             m_IxDisLast;

        // Методы

    public:
```

```
        // Конструктор — передает идентификатор диалогового окна в
        //   базовый класс
        Dialog( CWnd *pParent = 0 );

    protected:

        // Перегружаем виртуальный метод для инициализации
        virtual BOOL OnInitDialog( void );

        // Перегружаем виртуальный метод для получения информации
        //   из диалога
        virtual void OnOK( void );

        // Обработчик кнопки переноса данных из левого списка в
        //   правый
        afx_msg void OnButtonLRClick( void );
        // Обработчик кнопки переноса данных из правого списка в
        //   левый
        afx_msg void OnButtonRLClick( void );

        DECLARE_MESSAGE_MAP( );
    };

#endif
```

Листинг 10.2. Файл Dialog.cpp

```
/*
   Файл              : Dialog.cpp

   Проект            : демонстрация Windows-приложения на основе
                       библиотеки классов MFC с модальным
                       диалоговым окном, использующим списки

   Назначение        : реализация класса "Dialog", производного
                       от класса "CDialod" библиотеки классов
                       MFC (обработка сообщений окна)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAfx.h"                 // Прекомпилируемый файл
#include "Dialog.hpp"               // Объявление класса Dialog
```

```
// Идентификаторы ресурсов (поддержка редактором ресурсов)
#include "resource.h"

// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( Dialog, CDialog )
    ON_BN_CLICKED( IDC_BUTTON_LR, OnButtonLRClick )
    ON_BN_CLICKED( IDC_BUTTON_RL, OnButtonRLClick )
END_MESSAGE_MAP( )

// Конструктор
Dialog :: Dialog(
    CWnd                *pParent )  // Указатель на родительское
                                    //   окно (NULL — родительским
                                    //   является главное окно)
    : CDialog( IDD_DIALOG_LISTBOXES, pParent )
    { }

// Инициализация диалогового окна
BOOL Dialog :: OnInitDialog( void )
{
    // Вызываем метод базового класса
    CDialog :: OnInitDialog( );

    // Получаем указатели на списки
    m_plbMulti = static_cast< CListBox * >
        ( GetDlgItem( IDC_LIST_MULTI ) );
    m_plbSingle = static_cast< CListBox * >
        ( GetDlgItem( IDC_LIST_SINGLE ) );

    int                 IxStr;      // Индекс элемента списка
    // Заносим элементы в левый список
    for ( IxStr = 0; IxStr <= m_IxEnbLast; IxStr++ )
    {
        m_plbMulti->AddString( m_Enabled[ IxStr ] );
    }

    // Заносим элементы в правый список
    for ( IxStr = 0; IxStr <= m_IxDisLast; IxStr++ )
    {
        m_plbSingle->AddString( m_Disabled[ IxStr ] );
    }

    return TRUE;
}
```

```
// Получение информации из диалогового окна (обработчик кнопки OK)
void Dialog :: OnOK( void )
{
    // Вызываем метод базового класса
    CDialog :: OnOK( );

    // Индекс элемента списка
    int                 IxStr;

    // Получаем индекс последнего элемента левого списка
    m_IxEnbLast = m_plbMulti->GetCount( ) — 1;
    // Извлекаем информацию из левого списка
    for ( IxStr = 0; IxStr <= m_IxEnbLast; IxStr++ )
    {
        m_plbMulti->GetText( IxStr, m_Enabled[ IxStr ] );
    }

    // Получаем индекс последнего элемента правого списка
    m_IxDisLast = m_plbSingle->GetCount( ) — 1;
    // Извлекаем информацию из правого списка
    for ( IxStr = 0; IxStr <= m_IxDisLast; IxStr++ )
    {
        m_plbSingle->GetText( IxStr, m_Disabled[ IxStr ] );
    }

    return;
}

// Обработчик кнопки "Move>>" переноса информации из левого списка
//   в правый
void Dialog :: OnButtonLRClick( void )
{
    // Для индексов выделенных строк левого списка
    int                 Selected[ 10 ];
    // Индекс последней заполненной строки в Selected
    int                 IxSelLast;
    // Индекс текущей выделенной строки левого списка
    int                 IxSel;
    // Копия перемещаемой строки
    CString             Copy;
    // Индекс обрабатываемой строки
    int                 IxStr;
```

```
    // Получаем уменьшенное на 1 число выделенных строк левого
    //   списка
    IxSelLast = m_plbMulti->GetSelCount( ) — 1;
    if (IxSelLast < 0)              // Выделенных строк нет
        return;

    // Заносим индексы выделенных строк левого списка в Selected
    m_plbMulti->GetSelItems( 10, Selected );
    // Переносим выделенные строки в правый список
    for ( IxSel = IxSelLast; IxSel >= 0; IxSel-- )
    {
        IxStr = Selected[ IxSel ];
        m_plbMulti->GetText( IxStr, Copy );
        m_plbMulti->DeleteString( IxStr );
        m_plbSingle->AddString( Copy );
    }

    return;
}

// Обработчик кнопки "Move<<" переноса информации из правого списка
//   в левый
void Dialog :: OnButtonRLClick( void )
{
    // Копия перемещаемой строки
    CString             Copy;
    // Индекс обрабатываемой строки
    int                 IxStr;

    // Получаем индекс выделенной строки правого списка
    IxStr = m_plbSingle->GetCurSel( );
    if ( IxStr == LB_ERR )          // Выделенной строки нет
        return;

    // Переносим выделенную строку из правого списка в левый
    m_plbSingle->GetText( IxStr, Copy );
    m_plbSingle->DeleteString( IxStr );
    m_plbMulti->AddString( Copy );

    return;
}
```

Главное окно приложения представлено на рис. 10.3, а модальное диалоговое окно со списками показано на рис. 10.4.

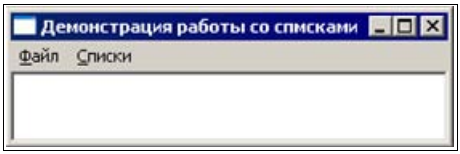


Рис. 10.3. Главное окно проекта UsgListBox

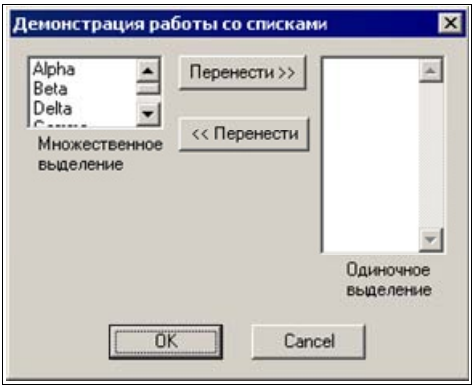


Рис. 10.4. Модальное диалоговое окно со списками проекта UsgListBox

В листингах 10.1 и 10.2 следует, прежде всего, обратить внимание на таблицу сообщений и методы их обработки. В демонстрационной программе используется модальное диалоговое окно с двумя списками, один из которых обеспечивает множественное, а другой — одиночное выделение. Перемещение выделенных элементов между списками осуществляется с помощью двух нажимаемых кнопок. Поэтому в классе `Dialog`, производном от класса `CDialog`, достаточно обрабатывать только нажатие кнопок переноса информации — макрос `ON_BN_CLICKED`, обработчики `OnButtonLRClick` и `OnButtonRLClick`. Инициализация списков осуществляется путем перегрузки виртуального метода `OnInitDialog`, а получение информации из диалогового окна выполняется с помощью перегруженного виртуального метода `OnOK`.

Обратите первоочередное внимание на использование методов `GetCount` и `GetText` (см. табл. 10.3, листинг 10.2), `GetCurSel` (см. листинг 10.2), `GetSelCount` и `GetSelItems` (см. табл. 10.4, листинг 10.2) и `AddString` и `DeleteString` (см. табл. 10.5, листинг 10.2). Остальная часть исходного кода представляется ясной. Поэтому далее рассмотрим шаблон ресурсов диалогового окна и списков в части создания и конфигурирования списков.

Для размещения в диалоговом окне списков воспользуйтесь панелью **Toolbox** (см. рис. 8.2). Выберите список (List Box) и перетащите его в нужное место диалогового окна. Для демонстрационного примера их размеры и расположение имеют вид, показанный на рис. 10.2.

Для настройки свойств достаточно вызвать контекстное меню, выполнить команду **Properties** и, в появившейся одноименной вкладке, задать свойства элемента. Свойства списков для демонстрационного примера показаны на рис. 10.5 и 10.6.

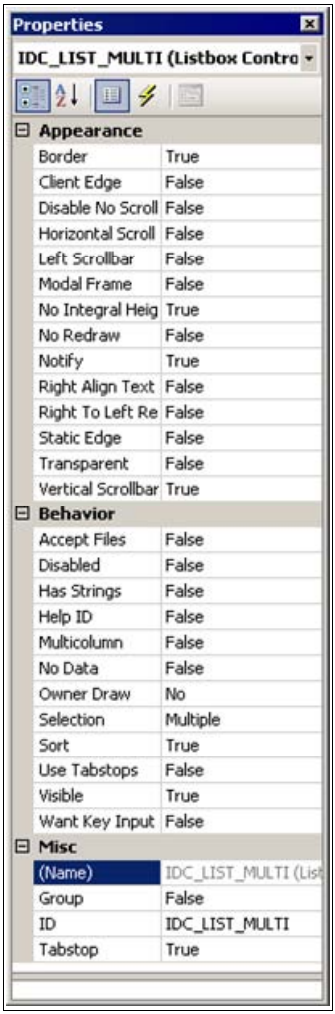


Рис. 10.5. Свойства списка с множественным выделением шаблона ресурсов диалогового окна проекта UsgListBoxes

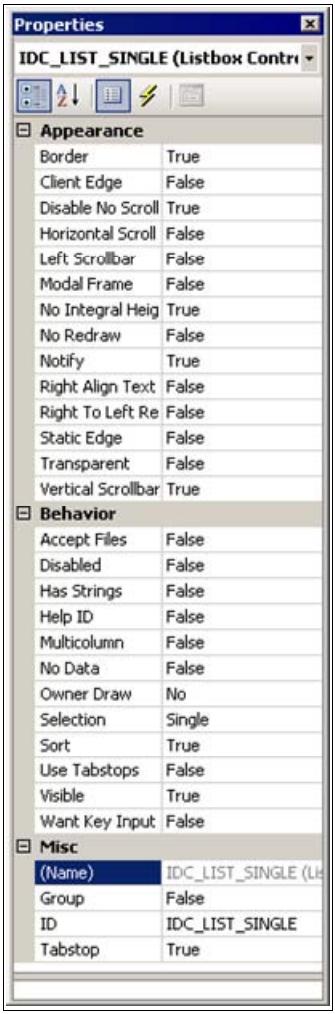


Рис. 10.6. Свойства списка с одиночным выделением шаблона ресурсов диалогового окна проекта UsgListBoxes

Совет

Обратите внимание на следующие идентификаторы: шаблона ресурсов диалогового окна (`IDD_DIALOG_LISTBOXES`), списков (`IDC_LIST_MULTI` и `IDC_LIST_`

SINGLE), кнопок переноса (IDC_BUTTON_LR и IDC_BUTTON_RL) и их использование в программном коде (см. листинг 10.2). Изучите свойства списков. Используя копию проекта UsgListBoxes, поэкспериментируйте, изменяя значения свойств списков.

10.2. Комбинированный список. Класс *CComboBox* библиотеки MFC

Комбинированный список обеспечивает всю полноту функциональных возможностей списка в сочетании с возможностями внедренного элемента управления — статического текста или элемента редактирования. Отсюда и происходит название элемента управления — комбинированный список. Список может быть либо раскрывающимся, в ответ на нажатие кнопки в правой части элемента, либо постоянно видимым. Выбранный элемент отображается в окне списка.

Существуют три разновидности комбинированных списков:

- простой комбинированный список;
- раскрывающийся комбинированный список;
- раскрывающийся список.

Простой комбинированный список виден всегда, а окно редактирования является дружественным (подчиненным) элементом управления. Раскрывающийся комбинированный список выводится на экран только тогда, когда его раскрывают, а окно редактирования является дружественным элементом. Раскрывающийся список выводится на экран, когда его раскрывают, а дружественным элементом является статический текст.

Все функциональные возможности комбинированного списка определены в классе CComboBox библиотеки MFC. Класс CComboBox является производным от класса CWnd и наследует все его возможности.

Комбинированный список может быть создан как программно, так и добавлен в редакторе ресурсов (последний вариант гораздо удобнее).

10.2.1. Стили, сообщения комбинированного списка и методы класса *CComboBox*

Комбинированный список может использовать элементарные стили, доступные всем объектам класса CWnd, а также ряд специальных стилей (табл. 10.7). Стиль комбинированного списка определяет его внешний вид и поведение. Стили устанавливаются либо при инициализации методом CComboBox::Create, либо заданием соответствующих свойств в редакторе ресурсов.

Таблица 10.7. *Элементарные стили комбинированного списка*

Макроопределение стиля	Значение
CBS_DISABLENOSCROLL, свойство **Disable No Scroll** равно **True**	Блокирует (затеняет) вертикальную полосу прокрутки списка, вместо того чтобы убрать ее в том случае, когда список содержит недостаточное для прокрутки количество элементов
CBS_DROPDOWN, свойство **Type** равно **Dropdown**	Список не выводится на экран до тех пор, пока пользователь не нажмет на кнопку раскрытия. Выбранный элемент списка отображается в окне редактирования
CBS_DROPDOWNLIST, свойство **Type** равно **Drop List**	Список не выводится на экран до тех пор, пока пользователь не нажмет на кнопку раскрытия. Выбранный элемент списка отображается в окне статического текста
CBS_LOWERCASE, свойство **Uppercase** равно **False**	Преобразует весь текст в нижний регистр, как в окне выбора, так и в списке
CBS_NOINTEGRALHEIGHT, свойство **No Integral Height** равно **True**	Устанавливает точно такие же размеры комбинированного списка, какие были заданы при его создании. Список показывает часть элементов
CBS_OEMCONVERT, свойство **OEM Convert** равно **True**	Преобразует строку в кодировку OEM (применяется в сочетании со стилями CBS_SIMPLE и CBS_DROPDOWN)
CBS_OWNERDRAWFIXED, свойство **Owner Draw** равно **Fixed**	Устанавливает одинаковую высоту элементов в списке
CBS_OWNERDRAWVARIABLE, свойство **Owner Draw** равно **Variable**	Устанавливает неодинаковую высоту элементов в
CBS_SIMPLE, свойство **Type** равно **Simple**	Список всегда виден, а выбранный элемент списка отображается в подчиненном окне редактирования
CBS_SORT, свойство **Sort** равно **True**	Автоматически сортирует строки в списке
CBS_UPPERCASE, свойство **Uppercase** равно **True**	Преобразует весь текст в верхний регистр, как в окне выбора, так и в списке

Примечание

Все макросы стилей полосы прокрутки начинаются с префикса CBS_ (от Combo Box Styles).

Часть комбинированного списка — непосредственно список — может выводиться на экран все время (стиль CBS_SIMPLE) или может раскрываться из текстового поля (стили CBS_DROPDOWN и CBS_DROPDOWNLIST). Текущая выбранная

запись списка выводится в прямоугольном окне, которое представляет собой окно простейшего текстового поля и может быть доступна (стили CBS_SIMPLE и CBS_DROPDOWN) или недоступна (стиль CBS_DROPDOWNLIST) для редактирования. Если комбинированный список определен как раскрывающийся, то при нажатии символьной клавиши на клавиатуре строка, начинающаяся с данного символа, будет выделена.

На рис. 10.7 представлены три типа комбинированных списков.

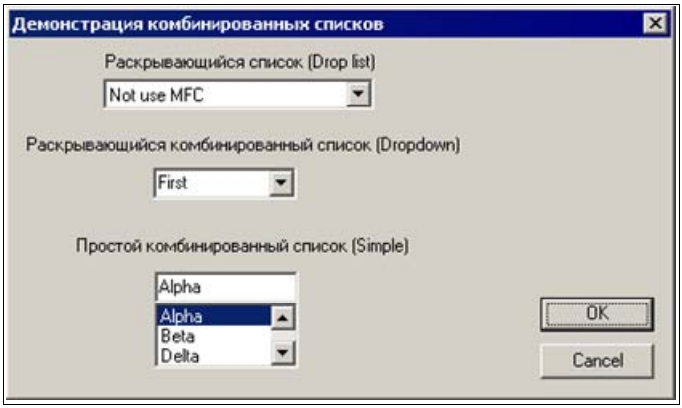


Рис. 10.7. Три общих стиля комбинированных списков

Так как библиотека MFC абстрагирует стандартные сообщения комбинированного списка CB_FINDSTRING и CB_SELECTSTRING методами FindString и SelectString класса CComboBox, то, обычно, приложение обрабатывает только уведомления. Комбинированный список посылает уведомления своему родительскому окну (обычно объекту класса, производного от класса CDialog библиотеки MFC). Эти уведомления могут быть перехвачены и обработаны, если существуют соответствующие уведомления таблицы сообщений и методы их обработки. Уведомления карты сообщений для комбинированного списка приведены в табл. 10.8.

Таблица 10.8. *Уведомления карты сообщений для комбинированного списка*

Уведомление таблицы сообщений	Значение
ON_CBN_CLOSEUP	Посылается при закрытии списка (если он не имеет стиля CBS_SIMPLE)
ON_CBN_DBLCLK	Посылается при двойном щелчке по элементу списка со стилем CBS_SIMPLE. Двойной щелчок не может быть выполнен по списку со стилем CBS_DROPDOWN или CBS_DROPDOWNLIST, т. к. одинарный щелчок сворачивает список

Таблица 10.8 *(окончание)*

Уведомление таблицы сообщений	Значение
ON_CBN_DROPDOWN	Посылается непосредственно перед раскрытием списка (комбинированный список должен иметь стиль CBS_DROPDOWN или CBS_DROPDOWNLIST)
ON_CBN_EDITUPDATE	Посылается непосредственно перед тем, как окно редактирования комбинированного списка выведет измененный текст, но после того, как этот текст отформатирован. От раскрывающегося комбинированного списка со стилем CBS_DROPDOWNLIST это сообщение не передается
ON_CBN_EDITCHANGE	Посылается, если пользователь изменил текст в окне редактирования комбинированного списка (после обновления изображения на экране). От раскрывающегося комбинированного списка со стилем CBS_DROPDOWNLIST это сообщение не передается
ON_CBN_ERRSPACE	Посылается, если комбинированный список не может выделить достаточно памяти
ON_CBN_KILLFOKUS	Посылается, если комбинированный список теряет фокус ввода
ON_CBN_SELCHANGE	Посылается, когда пользователь щелкает по списку или выбирает другой элемент с помощью клавиш со стрелками
ON_CBN_SELENDCANCEL	Посылается, когда пользователь щелкает мышью по элементу списка, а затем щелкает вне списка. В результате список сворачивается (выбор пользователя будет проигнорирован)
ON_CBN_SELENDOK	Посылается, когда пользователь выбирает элемент, а затем закрывает список (выбор пользователя допустим)
ON_CBN_SETFOKUS	Комбинированный список оказывается в фокусе ввода

Методы класса CComboBox библиотеки MFC делятся на методы общего назначения, специальные строковые методы и виртуальные методы.

Методы общего назначения класса CComboBox перечислены в табл. 10.9 и применимы для всех трех типов комбинированных списков.

Таблица 10.9. *Методы общего назначения класса* CComboBox

Метод	Описание
Clear	Удаляет элемент из списка
Copy	Копирует выбранный элемент (в формате текста) в буфер обмена

Таблица 10.9 *(продолжение)*

Метод	Описание
`Cut`	"Вырезает" выбранный элемент текста и помещает его в буфер обмена (в формате текста)
`GetCount`	Возвращает число элементов в списке
`GetCurSel`	Возвращает индекс выбранного элемента
`GetDroppedControlRect`	Возвращает экранные координаты видимой части раскрывающегося комбинированного списка
`GetDroppedState`	Определяет, видна ли раскрывающаяся часть списка
`GetDroppedWidth`	Возвращает минимально допустимую ширину раскрывающейся части списка
`GetEditSel`	Извлекает положение первого и последнего символов выделенного текста в окне редактирования комбинированного списка
`GetExtendedUI`	Определяет, имеет ли комбинированный список стандартный или расширенный пользовательский интерфейс
`GetHorizontalExtent`	Возвращает величину в пикселах, на которую возможно горизонтальное прокручивание раскрывающейся части списка
`GetItemData`	Возвращает 32-разрядное значение, которое приложение связывает с указанным элементом комбинированного списка
`GetItemDataPtr`	Возвращает указатель типа `void *` на специальное 32-разрядное значение, которое приложение связывает с указанным элементом комбинированного списка
`GetItemHeight`	Возвращает высоту элементов комбинированного списка
`GetLBText`	Возвращает строку из комбинированного списка
`GetLBTextLen`	Извлекает длину строки элемента в комбинированном списке
`GetLocale`	Извлекает локальный идентификатор комбинированного списка
`GetTopIndex`	Возвращает индекс первого видимого элемента в списочной части комбинированного списка
`LimitText`	Ограничивает длину текста, который пользователь может ввести в окно редактирования комбинированного списка
`Paste`	Вставляет данные из буфера обмена в окно редактирования комбинированного списка
`SetCurSel`	Устанавливает строку в окне редактирования комбинированного списка

Таблица 10.9 *(окончание)*

Метод	Описание
SetDroppedWidth	Устанавливает минимально допустимую ширину раскрывающейся части комбинированного списка
SetEditSel	Устанавливает положение первого и последнего символов выделенного текста в окне редактирования комбинированного списка
SetExtendedUI	Устанавливает стандартный или расширенный пользовательский интерфейс
SetHorizontalExtent	Устанавливает величину в пикселах, на которую возможно горизонтальное прокручивание раскрывающейся части комбинированного списка
SetItemData	Устанавливает 32-разрядное значение, которое приложение связывает с указанным элементом комбинированного списка
SetItemDataPtr	Устанавливает указатель типа `void *` на 32-разрядное значение, которое приложение связывает с указанным элементом комбинированного списка
SetItemHeight	Устанавливает высоту элементов комбинированного списка
SetLocale	Устанавливает локальный идентификатор комбинированного списка
SetTopIndex	Устанавливает индекс первого видимого элемента в списочной части комбинированного списка
ShowDropDown	Показывает или прячет раскрывающуюся часть комбинированного списка

Строковые методы класса `CComboBox` перечислены в табл. 10.10, применимы для всех трех типов комбинированных списков и предназначены для обработки строковых данных элементов списка.

Таблица 10.10. *Специальные методы для работы со строками класса* `CComboBox`

Метод	Описание
AddString	Добавляет строку в конец списочной части комбинированного списка или в соответствующую алфавитную позицию
DeleteString	Удаляет строку из списка
Dir	Добавляет имена файлов текущего каталога в комбинированный список

Таблица 10.10 *(окончание)*

Метод	Описание
`FindString`	Ищет первый строковый элемент в комбинированном списке, имеющий указанный строковый префикс
`FindStringExact`	Находит первый строковый элемент в комбинированном списке, совпадающий со строкой поиска
`InsertString`	Вставляет в комбинированный список строку в позицию с указанным индексом
`ResetContent`	Удаляет из комбинированного списка все элементы и очищает окно редактирования
`SelectString`	Ищет строку в комбинированном списке и, если находит, выделяет ее и копирует в окно редактирования

В классе `CComboBox` имеется несколько виртуальных методов, которые можно перегрузить в производных классах (табл. 10.11). Перегрузка виртуальных методов позволит реализовать дополнительные возможности, не поддерживаемые библиотекой MFC.

Таблица 10.11. *Виртуальные методы класса* `CComboBox`

Метод	Описание
`CompareItem`	Вызывается для того, чтобы определить положение нового элемента в отсортированном списке
`DeleteItem`	Вызывается при удалении элемента из списка
`DrawItem`	Вызывается при перерисовке списка
`MeasureItem`	Вызывается при создании списка, чтобы определить размеры элементов списка

10.2.2. Демонстрационная программа UsgComboBoxes: использование комбинированных списков

Демонстрационный проект, иллюстрирующий работу комбинированных списков, находится на прилагаемом компакт-диске, в папке \Глава 10 \UsgComboBoxes, а файловый состав проекта показан на рис. 10.8.

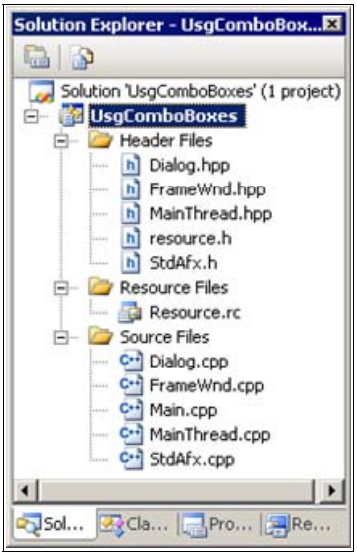


Рис. 10.8. Файловый состав проекта UsgComboBoxes

Совет

Копию проекта UsgComboBoxes для экспериментов создавайте копированием папки проекта с компакт-диска. Файлы StdAfx.h, StdAfx.cpp, Main.cpp, MainThread.hpp и MainThread.cpp в данном проекте являются стандартными, а файлы FrameWnd.hpp, FrameWnd.cpp, Dialog.hpp и Dialog.cpp представлены в листингах 10.3—10.6. Оставшиеся файлы — resource.h и Resourse.rc — созданы редактором ресурсов и поддерживаются им. Ресурсы проекта представлены на рис. 10.9.

Листинг 10.3. Файл FrameWnd.hpp

```
/*
   Файл                : FrameWnd.hpp

   Проект              : демонстрация Windows-приложения на основе
                         библиотеки классов MFC с модальным
                         диалоговым окном, использующим различные
                         комбинированные списки

   Назначение          : объявление класса "FrameWnd",
                         производного от класса "CFrameWnd"
                         библиотеки классов MFC (обработка сообщений
                         главного окна)
```

```
   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _FrameWnd_hpp
    #define _FrameWnd_hpp

    // Объявление класса
    class FrameWnd : public CFrameWnd
    {
        // Данные

    private:

        // Переменные, связанные с простым комбинированным списком
        // Строки для списка
        CString         m_List1[ 10 ];
        int             m_IxLast1;  // Индекс последнего элемента
        CString         m_Str1;     // Текущая строка списка

        // Переменные, связанные с раскрывающимся комбинированным
        //   списком
        CString         m_List2[ 10 ];
        int             m_IxLast2;
        CString         m_Str2;

        // Переменные, связанные с раскрывающимся списком
        CString         m_List3[ 10 ];
        int             m_IxLast3;
        CString         m_Str3;

        // Методы

    public:

        // Конструктор умолчания (инициализация комбинированных
        //   списков)
        FrameWnd( void );

    protected:

        // Прототипы функций, обрабатывающих сообщения
        // Обработка команды Файл | Выход
        afx_msg void OnMenuFileExit( void );
```

```
        // Обработка команды КомбСписок | Диалог
        afx_msg void OnMenuComboBoxDialog( void );

        // Объявление таблицы сообщений
        DECLARE_MESSAGE_MAP( );
    };

#endif
```

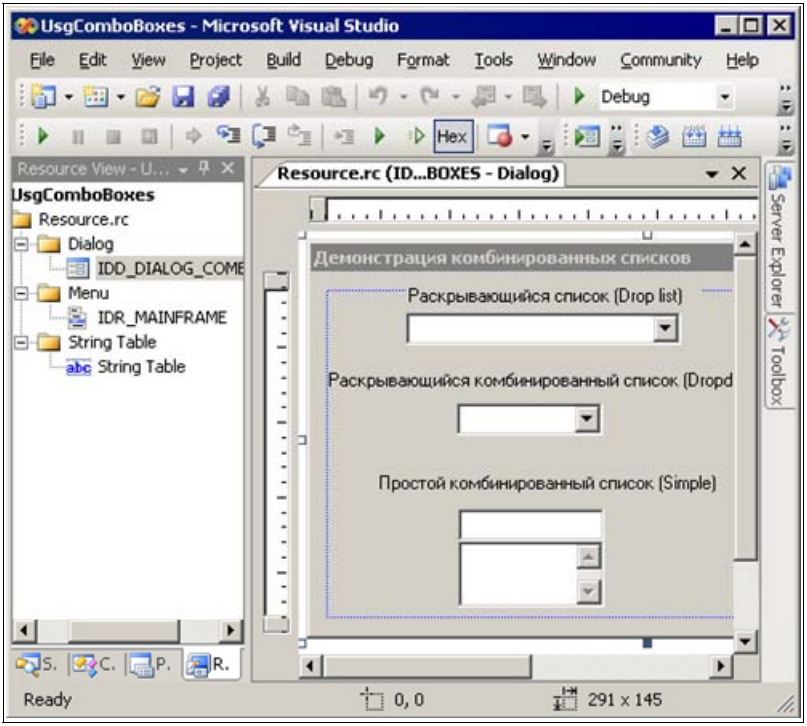


Рис. 10.9. Ресурсы проекта UsgComboBoxes

Листинг 10.4. Файл FrameWnd.cpp

```
/*
   Файл                 : FrameWnd.cpp

   Проект               : демонстрация Windows-приложения на основе
                          библиотеки классов MFC, использующего
                          различные комбинированные списки
```

```
   Назначение         : реализация класса "FrameWnd",
                        производного от класса "CFrameWnd"
                        библиотеки классов MFC (обработка сообщений
                        главного окна)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAfx.h"                  // Прекомпилируемый файл
#include "FrameWnd.hpp"              // Объявление класса FrameWnd
#include "Dialog.hpp"                // Объявление класса Dialog
// Идентификаторы ресурсов (поддержка редактором ресурсов)
#include "resource.h"

// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( FrameWnd, CFrameWnd )
    ON_COMMAND( ID_FILE_EXIT, OnMenuFileExit )
    ON_COMMAND( ID_COMBOBOX_DIALOG, OnMenuComboBoxDialog )
END_MESSAGE_MAP( )

// Конструктор по умолчанию (инициализация комбинированных списков)
FrameWnd :: FrameWnd( void )
{
    // Простой комбинированный список
    m_List1[ 0 ] = "Alpha";
    m_List1[ 1 ] = "Beta";
    m_List1[ 2 ] = "Gamma";
    m_List1[ 3 ] = "Delta";
    m_List1[ 4 ] = "Omega";
    m_IxLast1    = 4;

    // Раскрывающийся комбинированный список
    m_List2[ 0 ] = "First";
    m_List2[ 1 ] = "Second";
    m_List2[ 2 ] = "Third";
    m_List2[ 3 ] = "Fourth";
    m_IxLast2    = 3;

    // Раскрывающийся список
    m_List3[ 0 ] = "Not use MFC";
    m_List3[ 1 ] = "Use MFC in a Static Library";
    m_List3[ 2 ] = "Use MFC in a Shared DLL";
    m_IxLast3    = 2;
}
```

```
// Обработка команды Файл | Выход
afx_msg void FrameWnd :: OnMenuFileExit( void )
{
    BOOL                rc = MessageBeep( -1 );
    if( !rc )
    {
        TRACE0( "\n Ошибка 2. Неверное завершение MessageBeep \n" );
        exit( 2 );
    }
    rc = DestroyWindow( );
    if( !rc )
    {
        TRACE0( "\n Ошибка 3. Окно не было разрушено \n" );
        exit( 3 );
    }

    return afx_msg void( );
}

// Обработка команды КомбСписок | Диалог
afx_msg void FrameWnd :: OnMenuComboBoxDialog( void )
{
    // Создаем объект диалогового окна — автоматически вызывается
    //   конструктор
    Dialog              Dlg( this );

    // Передаем информацию для комбинированных списков из класса
    //   FrameWnd в класс Dialog
    Dlg.m_List1   = m_List1;
    Dlg.m_IxLast1 = m_IxLast1;
    Dlg.m_List2   = m_List2;
    Dlg.m_IxLast2 = m_IxLast2;
    Dlg.m_List3   = m_List3;
    Dlg.m_IxLast3 = m_IxLast3;

    // Создается модальное диалоговое окно — оно будет
    //   автоматически закрыто после нажатия OK или Cancel
    int                 DlgResult = static_cast<int>
                                    ( Dlg.DoModal( ) );
    if ( DlgResult == IDOK )
    {             // Выход из диалогового окна по OK
        // Передаем выбранные элементы комбинированных списков из
        //   класса Dialog в класс FrameWnd
```

```
        m_Str1 = Dlg.m_Str1;
        m_Str2 = Dlg.m_Str2;
        m_Str3 = Dlg.m_Str3;
        // Визуализируем выбранные элементы
        MessageBox( m_Str1 + "\n" + m_Str2 + "\n" + m_Str3,
                    _T( "Выбраны элементы" ),
                    MB_OK | MB_ICONINFORMATION );
    }

    return afx_msg void( );
}
```

Листинг 10.5. Файл Dialog.hpp

```
/*
   Файл                : Dialog.hpp

   Проект              : демонстрация Windows-приложения,
                         использующего различные
                         комбинированные списки

   Назначение          : объявление класса "Dialog", производного
                         от класса "CDialod" библиотеки классов
                         MFC

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _Dialog_hpp
    #define _Dialog_hpp

    // Объявление класса
    class Dialog : public CDialog
    {
        // Данные

    private:

        // Указатель на простой комбинированный список
        CComboBox       *m_pCombo1;
```

```
        // Указатель на раскрывающийся комбинированный список
        CComboBox *     m_pCombo2;
        // Указатель на раскрывающийся список
        CComboBox *     m_pCombo3;

    public:

        // Переменные, связанные с простым комбинированным списком
        CString         *m_List1;
        int             m_IxLast1;
        CString         m_Str1;

        // Переменные, связанные с раскрывающимся комбинированным
        //   списком
        CString         *m_List2;
        int             m_IxLast2;
        CString         m_Str2;

        // Переменные, связанные с раскрывающимся списком
        CString         *m_List3;
        int             m_IxLast3;
        CString         m_Str3;

        // Методы

    public:

        // Конструктор — передает идентификатор диалогового окна в
        //   базовый класс
        Dialog( CWnd *pParent = 0 );

    protected:

        // Перегружаем виртуальный метод для инициализации
        virtual BOOL OnInitDialog( void );

        // Перегружаем виртуальный метод для получения информации
        //   из диалога
        virtual void OnOK( void );
    };

#endif
```

Листинг 10.6. Файл Dialog.cpp

```
/*
   Файл                : Dialog.cpp

   Проект              : демонстрация Windows-приложения на основе
                         библиотеки классов MFC с модальным
                         диалоговым окном,
                         использующим различные
                         комбинированные списки

   Назначение          : реализация класса "Dialog", производного
                         от класса "CDialod" библиотеки классов
                         MFC (обработка сообщений окна)

   Microsoft Visual Studio C++ .NET 2005
*/


#include "StdAfx.h"                  // Прекомпилируемый файл
#include "Dialog.hpp"                // Объявление класса Dialog
// Идентификаторы ресурсов (поддержка редактором ресурсов)
#include "resource.h"

// Конструктор
Dialog :: Dialog(
    CWnd                *pParent )   // Указатель на родительское
                                     //   окно (NULL — родительским
                                     //   является главное окно)
    : CDialog( IDD_DIALOG_COMBOBOXES, pParent )
    { }

// Инициализация окна диалога
BOOL Dialog :: OnInitDialog( void )
{
    // Вызываем метод базового класса
    CDialog :: OnInitDialog( );

    // Получаем указатели на комбинированные списки
    m_pCombo1 = static_cast< CComboBox * >
        ( GetDlgItem( IDC_COMBO1 ) );
    m_pCombo2 = static_cast< CComboBox * >
        ( GetDlgItem( IDC_COMBO2 ) );
```

```
    m_pCombo3 = static_cast< CComboBox * >
        ( GetDlgItem( IDC_COMBO3 ) );

    int                 IxStr;      // Индекс элемента списка
    // Заносим элементы в простой комбинированный список и
    //   выбираем строку в окне редактирования
    for ( IxStr = 0; IxStr <= m_IxLast1; IxStr++ )
    {
        m_pCombo1->AddString( m_List1[ IxStr ] );
    }
    m_pCombo1->SetCurSel( 0 );

    // Заносим элементы в раскрывающийся комбинированный список и
    //   выбираем строку в окне редактирования
    for ( IxStr = 0; IxStr <= m_IxLast2; IxStr++ )
    {
        m_pCombo2->AddString( m_List2[ IxStr ] );
    }
    m_pCombo2->SetCurSel( 0 );

    // Заносим элементы в раскрывающийся список и выбираем строку
    //   в окне редактирования
    for ( IxStr = 0; IxStr <= m_IxLast3; IxStr++ )
    {
        m_pCombo3->AddString( m_List3[ IxStr ] );
    }
    m_pCombo3->SetCurSel( 0 );

    return TRUE;
}

// Получение информации из диалогового окна (обработчик кнопки OK)
void Dialog :: OnOK( void )
{
    // Вызываем метод базового класса
    CDialog :: OnOK( );

    m_pCombo1->GetWindowText( m_Str1 );
    m_pCombo2->GetWindowText( m_Str2 );
    m_pCombo3->GetWindowText( m_Str3 );

    return;
}
```

Главное окно приложения представлено на рис. 10.10, а модальное диалоговое окно с комбинированными списками показано на рис. 10.11.

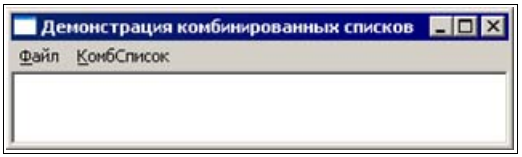


Рис. 10.10. Главное окно проекта UsgComboBoxes

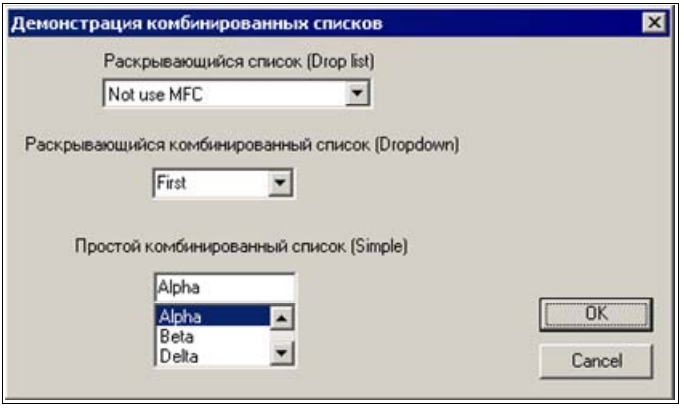


Рис. 10.11. Модальное диалоговое окно
с комбинированными списками проекта UsgComboBoxes

Совет

Пользуясь копией проекта UsgComboBoxes, поэкспериментируйте с комбинированными списками, пользуясь как мышью, так и клавиатурой. Выполните раскрытие или закрытие раскрывающихся списков, выбор элемента списка, редактирование текста в дружественном окне редактирования. При работе с комбинированными списками изучите действия клавиш-стрелок, клавиши табуляции <Tab>, клавиш <Home>, <End> и <Del>, а также клавиатурных комбинаций <Ctrl>+<C>, <Ctrl>+<V> и <Ctrl>+<X>.

В приведенных листингах 10.3—10.6 следует, прежде всего, обратить внимание на то, что относится к комбинированным спискам.

В демонстрационной программе используется модальное диалоговое окно с тремя типами комбинированного списка. Программная обработка комбинированных списков выполняется в двух классах — `FrameWnd` и `Dialog`, производных, соответственно, от классов `CFrameWnd` и `CDialog` библиотеки MFC.

В классе `FrameWnd` имеются: закрытые данные, связанные с комбинированными списками (см. листинг 10.3), конструктор для их инициализации (см. лис-

тинги 10.3—10.4) и обработчик `OnMenuComboBoxDialog` для команды меню **КомбСписок | Диалог** для создания модального диалогового окна с комбинированными списками (см. листинги 10.3—10.4).

В классе `Dialog` имеются: открытые данные, связанные с комбинированными списками (см. листинг 10.5), доступные в классе `FrameWnd`, и перегруженные методы `OnInitDialog` и `OnOK`, представленные в листингах 10.5—10.6. Инициализация комбинированных списков осуществляется с помощью метода `OnInitDialog`, а получение информации из окна диалога выполняется с по-

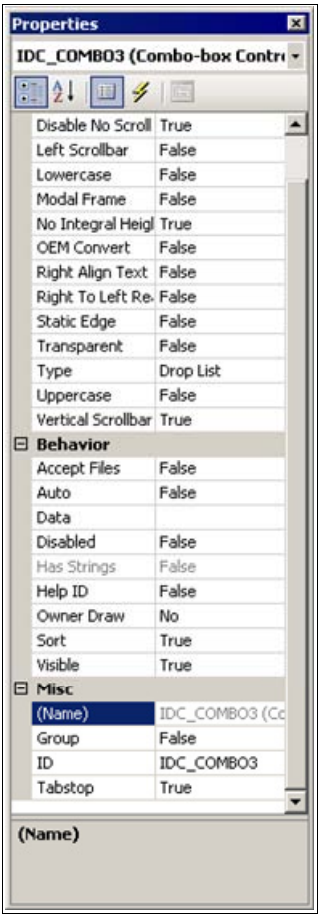


Рис. 10.12. Свойства раскрывающегося списка с множественным выделением шаблона ресурсов диалогового окна проекта UsgComboBoxes

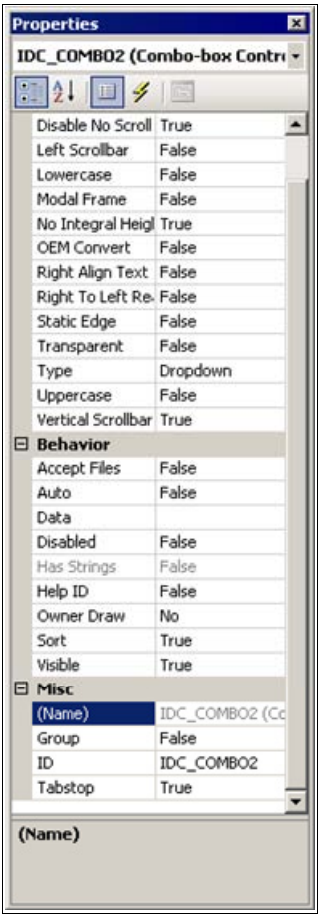


Рис. 10.13. Свойства раскрывающегося комбинированного списка шаблона ресурсов диалогового окна проекта UsgComboBoxes

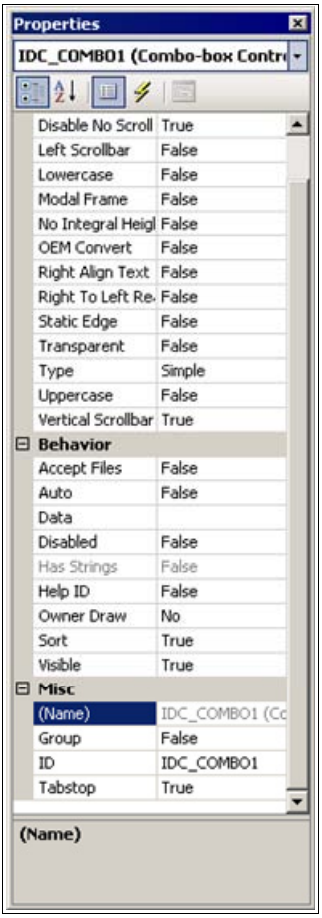


Рис. 10.14. Свойства комбинированного списка шаблона ресурсов диалогового окна проекта UsgComboBoxes

мощью метода `OnOK`. Обратите внимание на члены классов `FrameWnd` и `Dialog` и на методы `CComboBox::AddString`, `CComboBox::SetCurSel` (см. листинг 10.6, табл. 10.9 и 10.10), также на метод `CWnd::GetWindowText` (см. листинг 10.6).

Рассмотрим шаблон ресурсов диалогового окна и комбинированных списков в части создания и конфигурирования комбинированных списков. Для размещения комбинированных списков воспользуйтесь панелью **Toolbox** (см. рис. 8.2) — выберите комбинированный список (Combo Box) и перетащите его в нужное место диалогового окна. Для демонстрационного примера их размеры и расположение имеют вид, показанный на рис. 10.9.

Для настройки свойств достаточно вызвать контекстное меню для списка, выполнить команду **Properties** и, в появившейся одноименной вкладке, задать свойства. Свойства комбинированных списков для демонстрационного примера показаны на рис. 10.12—10.14.

Совет

Обратите внимание на следующие идентификаторы: шаблона ресурсов диалогового окна (`IDD_DIALOG_COMBOBOXES`), комбинированных списков (`IDC_COMBO1`, `IDC_COMBO2` и `IDC_COMBO3`) и их использование в коде (см. листинги 10.4 и 10.6). Изучите свойства комбинированных списков. Используя копию проекта UsgComboBoxes, поэкспериментируйте, изменяя значения свойств.

10.3. Индикатор прогресса. Класс *CProgressCtrl* библиотеки MFC. Таймер

Индикатор прогресса — это окно, обеспечивающее визуальную обратную связь во время какой-либо длительной операции, выполняемой приложением. Индикатор прогресса дает возможность наблюдать за ходом операции и применяется только для вывода.

Индикатор прогресса, как и ползунок, имеет диапазон и текущую позицию. Диапазон соответствует продолжительности операции, а текущая позиция показывает, насколько, к данному моменту, выполнена операция. На основе этих значений подсчитывается процент наполнения индикатора (рис. 10.15).

Все функциональные возможности индикатора прогресса определены в классе `CProgressCtrl` библиотеки MFC. Класс `CProgressCtrl` является производным от класса `CWnd` и наследует все его возможности.

Индикатор прогресса может быть создан как в исходном коде, как подчиненный элемент любого окна, так и размещен в окне, в редакторе ресурсов.

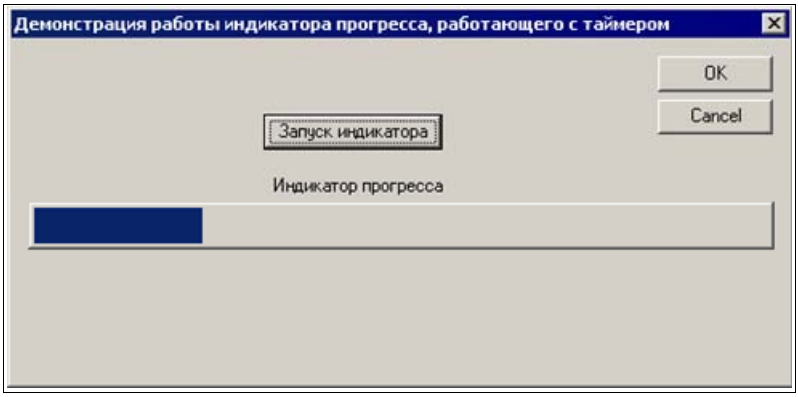


Рис. 10.15. Индикатор прогресса проекта UsgProgressTimer

10.3.1. Стили индикатора прогресса и методы класса *CProgressCtrl*

Индикатор прогресса может использовать элементарные стили, доступные всем объектам класса `CWnd`, а также ряд специальных стилей (табл. 10.12). Стиль индикатора прогресса определяет его внешний вид и поведение и устанавливается либо при инициализации, методом `CProgressCtrl::Create`, либо при использовании индикатора прогресса в шаблоне ресурсов диалогового окна при задании его свойств.

Таблица 10.12. *Элементарные стили индикатора прогресса*

Макроопределение стиля	Значение
`PBS_VERTICAL`, свойство Vertical равно True	Задает вертикальное расположение индикатора прогресса
`PBS_SMOOTH`, свойство Smooth равно True	Задает сглаживание при заполнении индикатора прогресса

Примечание

Макросы стилей индикатора прогресса начинаются с префикса `PBS_` (от Progress Box Styles).

Методы класса `CProgressCtrl` перечислены в табл. 10.13.

Таблица 10.13. *Методы класса* `CProgressCtrl`

Метод	Описание
`OffsetPos`	Продвигает текущую позицию индикатора прогресса на указанную величину и перерисовывает его, чтобы отобразить новую текущую позицию
`SetPos`	Устанавливает текущую позицию индикатора прогресса и перерисовывает его, чтобы отобразить новую текущую позицию
`SetRange`	Устанавливает минимальное и максимальное значения индикатора прогресса и перерисовывает его, чтобы отобразить новый диапазон
`SetStep`	Устанавливает пошаговое увеличение для индикатора прогресса
`StepIt`	Продвигает текущую позицию индикатора прогресса на величину пошагового увеличения и перерисовывает его, чтобы отобразить новую текущую позицию

10.3.2. Работа с таймером

Таймер является элементом управления, который периодически извещает приложение об истечении заданного интервала времени. Происходит это с помощью посылки сообщения `WM_TIMER`. Таймер запускается с помощью метода:

```
UINT_PTR CWnd::SetTimer( UINT_PTR nIDEvent, UINT nElapse,
    void(CALLBACK *lpfnTimer)(HWND, UINT, UINT_PTR, DWORD) );
```

Первый параметр `nIDEvent` определяет идентификатор таймера, который может быть любым беззнаковым целым числом (в Windows одновременно допускается не более 16 таймеров).

Второй параметр `nElapse` задает интервал таймера в миллисекундах (в Windows реальное разрешение составляет 1/18.2 сек).

Последний параметр определяет функцию, которая будет получать сообщения таймера из приложения (`NULL` означает, что такая функция отсутствует и сообщение `WM_TIMER` будет помещено в очередь сообщений приложения).

Остановить поток сообщений таймера можно в любой момент вызвав метод:

```
BOOL CWnd :: KillTimer( UINT_PTR nIDEvent );
```

Единственным параметром этого метода является идентификатор таймера, который нужно остановить.

Сообщения каждого из таймеров можно обработать функцией-обработчиком:

```
afx_msg void OnTimer( UINT nIDEvent );
```

Единственным параметром этой функции также является идентификатор таймера.

10.3.3. Демонстрационная программа UsgProgressTimer: использование индикатора прогресса и таймера

Демонстрационный проект, иллюстрирующий использование индикатора прогресса и таймера, находится на компакт–диске, в папке \Глава 10 \UsgProgressTimer, а файловый состав проекта показан на рис. 10.16.

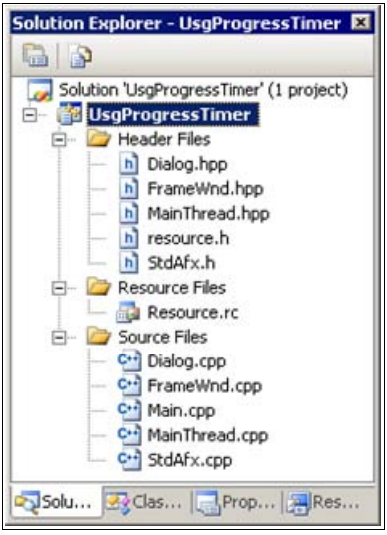


Рис. 10.16. Файловый состав проекта UsgProgressTimer

Совет

Копию проекта UsgProgressTimer для экспериментов создавайте копированием папки проекта с компакт-диска. Файлы StdAfx.h, StdAfx.cpp, Main.cpp, MainThread.hpp и MainThread.cpp в данном проекте являются стандартными, а файлы FrameWnd.hpp, FrameWnd.cpp, Dialog.hpp и Dialog.cpp представлены в листингах 10.7—10.10. Оставшиеся файлы — resource.h и Resourse.rc — созданы редактором ресурсов и поддерживаются им. Ресурсы проекта представлены на рис. 10.17.

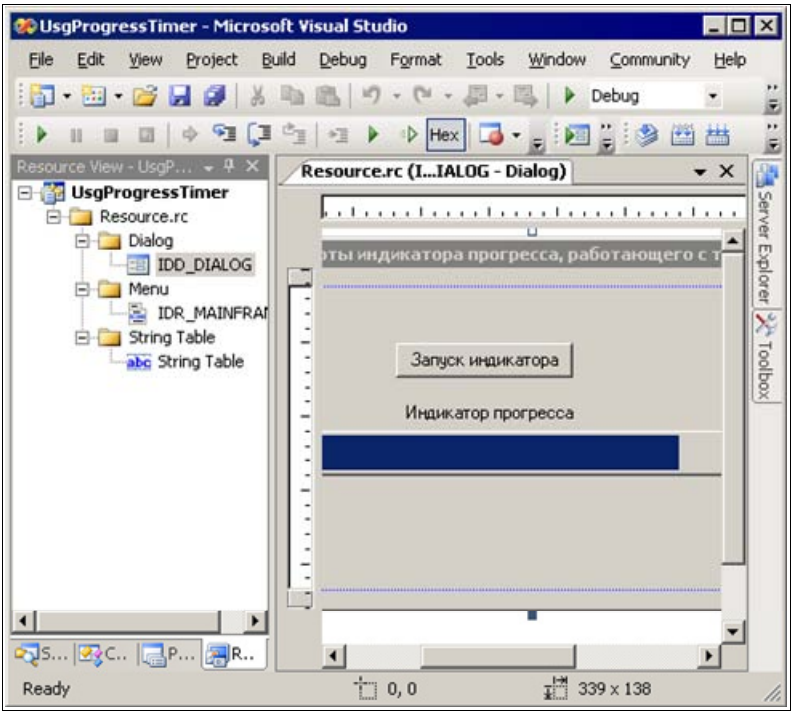


Рис. 10.17. Ресурсы проекта UsgProgressTimer

Листинг 10.7. Файл FrameWnd.hpp

```
/*
   Файл                : FrameWnd.hpp

   Проект              : демонстрация Windows-приложения,
                         использующего индикатор прогресса
                         и таймер

   Назначение          : объявление класса "FrameWnd", производного
                         от класса "CFrameWnd" библиотеки классов
                         MFC (обработка сообщений главного окна)

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
```

```
#ifndef _FrameWnd_hpp
    #define _FrameWnd_hpp

    // Объявление класса
    class FrameWnd : public CFrameWnd
    {
        // Методы

    protected:

        // Прототипы функций, обрабатывающих сообщения
        // Обработка команды Файл | Выход
        afx_msg void OnMenuFileExit( void );
        // Обработка команды Индикатор прогресса и таймер | Диалог
        afx_msg void OnMenuProgressTimer( void );

        // Объявление таблицы сообщений
        DECLARE_MESSAGE_MAP( );
    };

#endif
```

Листинг 10.8. Файл FrameWnd.cpp

```
/*
   Файл               : FrameWnd.cpp

   Проект             : демонстрация Windows-приложения,
                        использующего индикатор прогресса
                        и таймер

   Назначение         : реализация класса "FrameWnd", производного
                        от класса "CFrameWnd" библиотеки классов
                        MFC (обработка сообщений главного окна)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAfx.h"                 // Прекомпилируемый файл
#include "FrameWnd.hpp"             // Объявление класса FrameWnd
#include "Dialog.hpp"               // Объявление класса Dialog
// Идентификаторы ресурсов (поддержка редактором ресурсов)
#include "resource.h"
```

```
// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( FrameWnd, CFrameWnd )
    ON_COMMAND( ID_FILE_EXIT, OnMenuFileExit )
    ON_COMMAND( ID_PROGRESS_TIMER, OnMenuProgressTimer )
END_MESSAGE_MAP( )

// Обработка команды Файл | Выход
afx_msg void FrameWnd :: OnMenuFileExit( void )
{
    BOOL                rc = MessageBeep( -1 );
    if( !rc )
    {
        TRACE0( "\n Ошибка 2. Неверное завершение MessageBeep \n" );
        exit( 2 );
    }
    rc = DestroyWindow( );
    if( !rc )
    {
        TRACE0( "\n Ошибка 3. Окно не было разрушено \n" );
        exit( 3 );
    }

    return afx_msg void( );
}

// Обработка команды Индикатор прогресса и таймер | Диалог
afx_msg void FrameWnd :: OnMenuProgressTimer( void )
{
    // Создаем объект диалогового окна — автоматически вызывается
    //   конструктор
    Dialog              Dlg( this );

    // Создается модальное диалоговое окно — оно будет
    //   автоматически закрыто после нажатия OK или Cancel
    Dlg.DoModal( );

    return afx_msg void( );
}
```

Листинг 10.9. Файл Dialog.hpp

```
/*
   Файл                : Dialog.hpp

   Проект              : демонстрация Windows-приложения,
                         использующего индикатор прогресса
                         и таймер

   Назначение          : объявление класса "Dialog", производного
                         от класса "CDialog" библиотеки классов
                         MFC (обработка сообщений окна)

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _Dialog_hpp
    #define _Dialog_hpp

    // Объявление класса
    class Dialog : public CDialog
    {
        // Данные

    private:

        // Указатель на индикатор прогресса
        CProgressCtrl   *m_pProgressCtrl;
        // Возвращаемое значение для SetTimer
        UINT_PTR        m_nTimer;
        // Счетчик запусков таймера
        unsigned int    m_nCount;
        // Максимальное число запусков таймера
        enum { nMaxCount = 100 };

    public:

        // Конструктор — передает идентификатор диалогового окна в
        //   базовый класс
        Dialog( CWnd *pParent = 0 );
```

```
    protected:

        // Обработчик таймера
        afx_msg void OnTimer( UINT nIDEvent );
        // Обработчик кнопки "Запуск индикатора"
        afx_msg void OnStart( void );

        // Объявление таблицы сообщений
        DECLARE_MESSAGE_MAP( );
    };

#endif
```

Листинг 10.10. Файл Dialog.cpp

```
/*
   Файл               : Dialog.cpp

   Проект             : демонстрация Windows-приложения,
                        использующего индикатор прогресса
                        и таймер

   Назначение         : реализация класса "Dialog", производного
                        от класса "CDialod" библиотеки классов
                        MFC (обработка сообщений диалогового окна)

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAfx.h"                 // Прекомпилируемый файл
#include "Dialog.hpp"               // Объявление класса Dialog
// Идентификаторы ресурсов (поддержка редактором ресурсов)
#include "resource.h"

// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( Dialog, CDialog )
    ON_WM_TIMER( )
    ON_BN_CLICKED( IDC_START, OnStart )
END_MESSAGE_MAP( )

// Конструктор
Dialog :: Dialog(
    CWnd               *pParent )  // Указатель на родительское
                                   //   окно (NULL — родительским
                                   //   является главное окно)
```

```
    : CDialog( IDD_DIALOG, pParent )
    {
        m_nCount = 0;
    }

// Обработчик кнопки "Запуск индикатора"
afx_msg void Dialog :: OnStart( void )
{
    if( ( m_nCount > 0 ) && ( m_nCount < nMaxCount ) )
        return afx_msg void( );

    m_nCount = 0;

    // Запускаем таймер с интервалом 0.1 секунды
    m_nTimer = SetTimer( 0, 100, NULL );
    ASSERT( m_nTimer );

    return afx_msg void( );
}

// Обработчик таймера
afx_msg void Dialog :: OnTimer( UINT nIDEvent )
{
    if( m_nCount > nMaxCount )
    {                                  // Индикатор прогресса работу
                                       //   закончил
        ASSERT( KillTimer( 0 ) );
        return afx_msg void( );
    }

    m_nCount++;

    // Обновляем состояние индикатора прогресса
    m_pProgressCtrl = static_cast< CProgressCtrl * >
                      ( GetDlgItem( IDC_PROGRESS ) );
    m_pProgressCtrl->SetPos( m_nCount*100/nMaxCount );

    return afx_msg void( );
}
```

Главное окно приложения представлено на рис. 10.18, а модальное диалоговое окно с индикатором прогресса показано на рис. 10.17.

В листингах 10.7—10.10 рассмотрим то, что относится к индикатору прогресса и таймеру. В демонстрационной программе используется модальное диа-

логовое окно с индикатором прогресса, запускаемым с помощью кнопки. В качестве диапазона индикатора прогресса задан временной интервал, а текущая позиция определяется значением времени, прошедшего от момента запуска индикатора прогресса. Обновление состояния индикатора прогресса происходит периодически по сигналам таймера. Программная обработка комбинированных списков выполняется в двух классах — классах `FrameWnd` и `Dialog`, производных, соответственно, от классов `CFrameWnd` и `CDialog` библиотеки MFC.

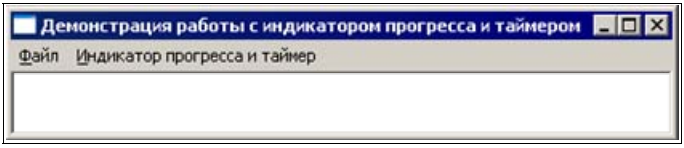


Рис. 10.18. Главное окно проекта UsgProgressTimer

В классе `FrameWnd` имеется обработчик команды меню **Индикатор прогресса и таймер | Диалог**, создающий модальное диалоговое окно с индикатором прогресса и кнопкой, управляющей им (см. рис. 10.17 и листинги 10.7, 10.8). Особенностью реализации модального диалогового окна, в демонстрационном примере, является то, что не требуется обмена информацией между классами `FrameWnd` и `Dialog`, т. к. не требуется инициализации диалогового окна и получения из него информации. По этой причине в классе `Dialog` не требуется перегрузки виртуальных методов `OnInitDialog` и `OnOK`.

В классе `Dialog` имеются: закрытые данные, связанные с индикатором прогресса и таймером (см. листинг 10.9), конструктор для передачи идентификатора диалогового окна в базовый класс и обработчики сообщений таймера и кнопки **Запуск индикатора** (см. листинги 10.9, 10.10). Обратите первоочередное внимание на члены классов `FrameWnd` и `Dialog`, а также используемые методы `CWnd::SetTimer`, `CWnd::KillTimer` и `CProgressCtrl::SetPos` — см. листинг 10.10, табл. 10.13 и *разд. 10.3.2*.

Примечание

Обратите внимание на то, что в демонстрационном проекте для задания диапазона индикатора прогресса не использован метод `CProgressCtrl::SetRange`, а использовано значение диапазона индикатора прогресса по умолчанию — от 0 до 100.

Далее рассмотрим шаблон ресурсов диалогового окна, индикатора прогресса и кнопки запуска индикатора в части создания и конфигурирования индикатора прогресса и кнопки запуска.

Для размещения индикатора прогресса воспользуйтесь панелью инструментов **Toolbox** (см. рис. 8.2) — выберите индикатор прогресса (Progress Control) и перетащите его в нужное место диалогового окна. Для демонстрационного примера размеры и расположение элементов интерфейса показаны на рис. 10.17.

Для настройки свойств любого управляющего элемента диалогового окна достаточно вызвать его контекстное меню, выбрать пункт меню **Properties** и, в появившейся одноименной вкладке, задать свойства. Свойства индикатора прогресса и кнопки для его запуска, для демонстрационного примера, показаны на рис. 10.19, 10.20.

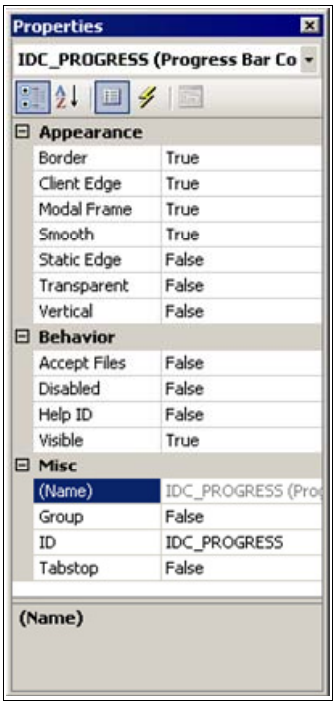


Рис. 10.19. Свойства индикатора прогресса проекта UsgProgressTimer

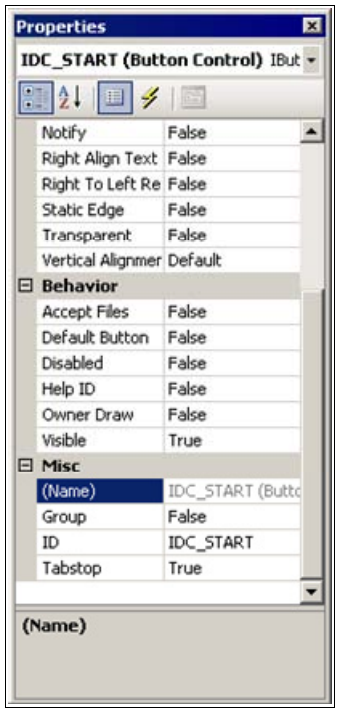


Рис. 10.20. Свойства кнопки запуска проекта UsgProgressTimer

Совет

Обратите внимание на следующие идентификаторы: диалогового окна (`IDD_DIALOG`), индикатора прогресса (`IDC_PROGRESS`), кнопки запуска (`IDC_START`), а также их использование в программном коде (см. листинги 10.8 и 10.10). Изучите свойства индикатора прогресса. Используя копию проекта UsgProgressTimer, поэкспериментируйте, изменяя значения свойств.

10.4. Вопросы и упражнения для самопроверки

Для самопроверки ответьте на следующие вопросы:

1. Что представляет собой список?
2. Перечислите имеющиеся виды списков.
3. Какой класс библиотеки MFC поддерживает работу со списками?
4. Укажите способы применения списков.
5. Перечислите элементарные стили списка и дайте им характеристику.
6. Укажите элементы таблицы сообщений от списков.
7. Перечислите и охарактеризуйте основные методы класса `CListBox` библиотеки MFC.
8. Что представляет собой комбинированный список?
9. Перечислите имеющиеся виды комбинированных списков и дайте им характеристику.
10. Какой класс библиотеки MFC поддерживает работу с комбинированными списками?
11. Укажите способы применения комбинированных списков.
12. Перечислите элементарные стили комбинированного списка и дайте им характеристику.
13. Укажите элементы таблицы сообщений от комбинированных списков.
14. Перечислите и охарактеризуйте основные методы класса `CComboBox` библиотеки MFC.
15. Что представляет собой индикатор прогресса?
16. Какой класс библиотеки MFC поддерживает работу с индикатором прогресса?
17. Перечислите элементарные стили индикатора прогресса и дайте им характеристику. Охарактеризуйте основные методы класса `CProgressCtrl` библиотеки MFC, используемые при работе с индикатором прогресса.
18. Что представляет собой таймер? Как запустить, остановить таймер или обработать его сообщение?

Ответы на вопросы можно проверить — они приведены в *разд. П1.10 приложения 1*.

При экспериментах с созданной копией демонстрационного проекта UsgListBoxes выполните следующие упражнения:

1. Попробуйте менять свойства списков и посмотрите, на что это влияет.
2. Добавьте в модальное диалоговое окно еще один список и организуйте перенос данных между ним и любым другим списком.

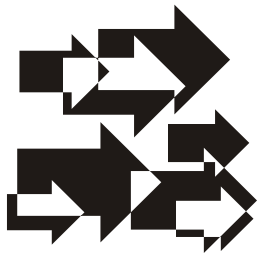
При экспериментах с созданной копией демонстрационного проекта UsgComboBoxes выполните следующие упражнения:

1. Поэкспериментируйте с комбинированными списками пользуясь мышью. Выполните раскрытие или закрытие раскрывающихся списков, выбор элемента списка, редактирование текста в дружественном окне редактирования.
2. Поэкспериментируйте с комбинированными списками пользуясь клавиатурой. Выполните раскрытие или закрытие раскрывающихся списков, выбор элемента списка, редактирование текста в дружественном окне редактирования. Изучите действия клавиш-стрелок, клавиши табуляции <Tab>, клавиш <Home>, <End> и <Del>, клавиатурных комбинаций <Ctrl>+<C>, <Ctrl>+<V> и <Ctrl>+<X>.
3. Попробуйте менять свойства комбинированных списков и посмотрите, на что это влияет.
4. По полученным результатам опишите действие свойств комбинированных списков.

При экспериментах с созданной копией демонстрационного проекта UsgProgressTimer выполните следующие упражнения:

1. Попробуйте менять свойства индикатора прогресса. Посмотрите, на что это влияет.
2. Любым способом измените время заполнения индикатора прогресса.
3. Добавьте в диалоговое окно еще один индикатор прогресса и добейтесь его функционирования.
4. Попробуйте заменить горизонтальный индикатор прогресса вертикальным и обеспечьте его заполнение со сглаживанием.

ГЛАВА 11



Динамически подключаемые библиотеки [6]

Динамически подключаемые библиотеки (Dynamic-Link Library, DLL) или, далее, *динамические библиотеки*, широко применяются в Windows и это не случайно. DLL могут использовать различные приложения, причем многократно. К DLL можно отнести также элементы управления ActiveX и, в некотором роде, драйверы.

Примечание

Всех ожидаемых преимуществ DLL получить не удалось вследствие так называемого "DLL Hell" ("ад DLL"), возникающего в том случае, когда несколько приложений одновременно требуют различные, но не полностью совместимые версии библиотек DLL, что приводит к сбоям в приложениях. Проще говоря, сущность проблемы заключается в конфликте разных версий DLL, призванных поддерживать определенные функции. Количество DLL огромно, но не все обладают надежностью и совместимостью. Поэтому конфликты возникают довольно часто, снижая общую надежность операционной системы. Последние версии Windows разрешают параллельное использование разных версий DLL, что, в принципе, свело на нет преимущества изначального принципа модульности. При этом, в процессе развития, идея модульности выросла в концепцию COM (Component Object Model, Объектная модель компонентов).

Примечание

Большая часть кода MFC находится в DLL и эти файлы имеют имена типа MFCxx.DLL.

Формат DLL придерживается тех же соглашений, что и формат исполняемых файлов, сочетая код, таблицы и ресурсы. Динамические библиотеки являются средством создания по-настоящему модульного программного обеспечения. Может показаться, что программы и так модульные, раз используют объектно-ориентированную технологию, а классы языка C++ — модульные. Но

классы обеспечивают модульность при разработке программы, а DLL — в период ее *выполнения*. Вместо "гигантских" исполняемых exe-файлов, которые пришлось бы перестраивать и тестировать при любом, даже самом незначительном, изменении кода, гораздо эффективнее создавать компактные DLL-модули и тестировать их по отдельности. Еще раз отметим, что в Windows поддержка основных функций строится на применении именно DLL.

Создавать DLL достаточно просто. Объясняется это тем, что со стороны мастера создания приложений и библиотеки классов MFC поддержка DLL существенно расширена и улучшена. Далее рассмотрим, как создавать на языке C++ DLL-модули и использовать их в клиентских приложениях.

Существует два типа динамически подключаемых библиотек, поддерживаемых библиотекой классов MFC — обычные DLL и DLL расширений.

Обычная (regular) DLL может быть связана (загружена) с приложением как статически, так и динамически и доступна любым приложениям. Но такая DLL может экспортировать только C-функции и неспособна экспортировать классы, а также обычные и переопределяемые методы классов. Это объясняется тем, что каждый компилятор языка C++ использует свой метод расширения имен функций. Но при этом, внутри такой DLL, можно использовать классы.

DLL расширений (extension DLL) могут быть динамически связаны только с библиотекой MFC и использоваться приложениями на базе MFC. DLL такого типа позволяют хранить в себе весь набор типов данных, используемых в MFC. Соответственно, что приложение может легко создавать новые объекты классов из такой DLL. DLL расширений имеет простой интерфейс. Интерфейс — это набор переменных, указателей, функций или классов, которые содержит в себе данная DLL и к которым приложение будет иметь доступ.

В объявление экспортируемого класса нужно лишь добавить макрос `AFX_EXT_CLASS`, а обо всем остальном позаботится библиотека MFC:

```
// Объявление класса: обратите внимание на макрос AFX_EXT_CLASS
//   для экспорта класса
class AFX_EXT_CLASS MainThread : public CWinApp
{
        // ...
}
```

Далее, для краткости, рассмотрим, как наиболее простое и удобное, только создание и тестирование DLL расширений для приложений на базе MFC, как содержащих, так и не содержащих ресурсы.

11.1. Точка входа DLL: функция *DllMain*

Код библиотеки времени выполнения языка C++ вызывает функцию `DllMain` каждый раз, когда процесс или поток подключается к DLL или отключается от DLL.

Примечание

Процесс — это абстрактное понятие, относящееся к программе. Часто процессом называют программу и все ее элементы: адресное пространство, глобальные переменные, регистры, стек, счетчик команд, состояние, открытые файлы, дочерние процессы и т. д. Стандарт ISO 9000:2000 Definitions определяет процесс как совокупность взаимосвязанных и взаимодействующих действий, преобразующих входы в выходы. Проще говоря, процесс — это программа, загруженная в память и готовая к исполнению. Процесс является независимым объектом со своей областью памяти, внутренним состоянием и ресурсами, доступными *потокам выполнения* этого процесса. Процесс имеет, по крайней мере, один поток, но их, при необходимости, может быть и больше: т. е. процесс всегда начинается с одного потока, а остальные создаются по мере необходимости.

Примечание

Поток выполнения — это базовый элемент, которому операционная система выделяет время процессора. Каждый поток пользуется набором структур, где хранит свой контекст в то время, пока дожидается очередного кванта времени процессора. Каждый из потоков, принадлежащих одному процессу, совместно использует адресное пространство процесса и может получать доступ к его глобальным системным ресурсам. Поток может выполнять любую часть кода процесса. Многопоточность — способ выполнения процесса, при котором вычисления разбиваются на несколько потоков, выполняющихся "параллельно", т. е. без предписанного порядка во времени. При выполнении некоторых задач такое разделение может обеспечить более эффективное использование ресурсов вычислительной машины.

Функция `DllMain` использует соглашение вызова `APIENTRY` и имеет три параметра:

```
BOOL WINAPI DllMain( HINSTANCE hinstDLL, DWORD fdwReason,
                     LPVOID lpvReserved );
```

Параметр `hinstDLL` — это дескриптор процесса или потока, `fdwReason` (табл. 11.1) показывает причину вызова функции `DllMain`, а параметр `lpvReserved` зарезервирован для использования операционной системой Windows.

Примечание

Если DLL не нуждается в выделении памяти или других ресурсов, можно полностью исключить функцию `DllMain` из своего исходного кода. Компилятор автоматически подставит функцию `DllMain`, используемую по умолчанию.

Таблица 11.1. *Возможные значения параметра* `fdwReason` *функции* `DllMain`

Значение	Описание
`DLL_PROCESS_ATTACH`	Новый процесс пытается получить доступ к DLL
`DLL_THREAD_ATTACH`	Новый поток существующего процесса пытается получить доступ к DLL
`DLL_PROCESS_DETACH`	Последний поток процесса отсоединяется от DLL
`DLL_THREAD_DETACH`	Один из потоков процесса отсоединяется от DLL

Реализация функции `DllMain` может иметь следующий вид:

```
BOOL WINAPI DllMain(
    HINSTANCE           hinstDLL,
    DWORD               fdwReason,
    LPVOID              lpvReserved )
{
    switch( fdwReason )
    {
        case DLL_PROCESS_ATTACH:
            // ...
        case DLL_THREAD_ATTACH:
            // ...
        case DLL_PROCESS_DETACH:
            // ...
        case DLL_THREAD_DETACH:
            // ...
    }
    return TRUE;
}
```

11.2. Создание и использование DLL расширений

В качестве исходного проекта, при создании и тестировании DLL расширений, используем демонстрационный проект FrameWnd_PCH, представляющий собой простое приложение на базе MFC, без ресурсов *(см. главу 4)*.

11.2.1. Создание DLL расширений для приложения на базе MFC. Демонстрационный проект FrameWndExDLL

Для создания DLL расширений в IDE на вкладке **Start Page** выберите **Create: Project...** В появившемся окне **New Project** укажите свойства проекта в соответствии с рис. 11.1 и нажмите кнопку **OK**.

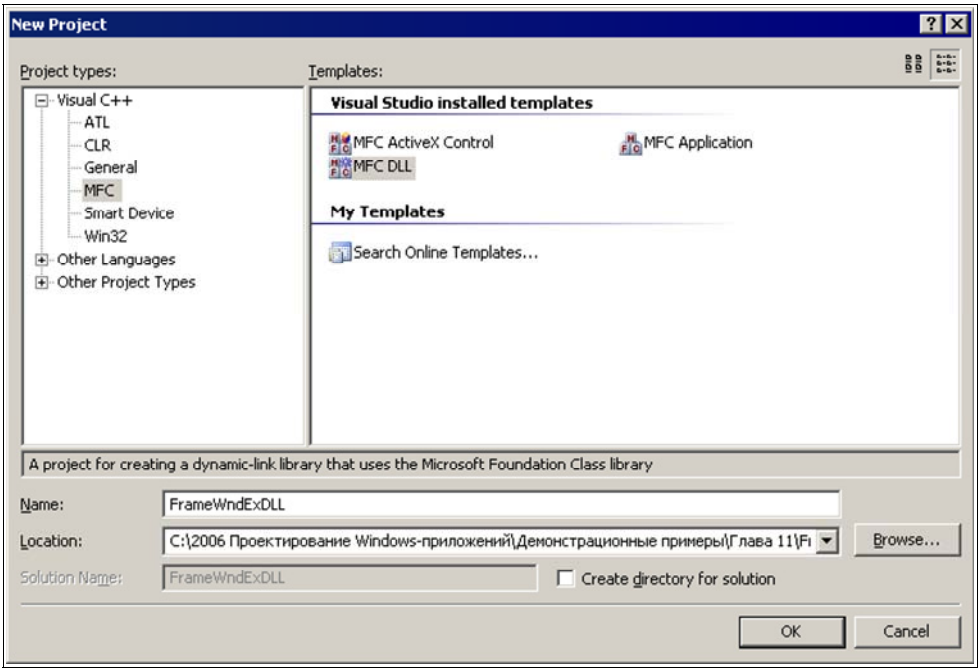


Рис. 11.1. Свойства проекта FrameWndExDLL

В результате появится окно **MFC DLL Wizard — FrameWndExDLL**, в котором следует выбрать вкладку **Application Settings**, настроить ее в соответствии с рис. 11.2 и нажать кнопку **Finish.**

В окне **Solution Explorer** IDE, для проекта FrameWndExDLL, вызовите контекстное меню, выполните пункт **Properties**, в появившемся окне **FrameWndExDLL Property Pages** выберите вкладку **General**, настройте ее в соответствии с рис. 11.3 и нажмите кнопку **OK**.

Итак, в нашем случае, DLL расширений строится для демонстрационного проекта FrameWmd_PCH из *главы 4*, из которого взяты файлы MainThread.hpp (добавлен макрос в заголовок класса), MainThread.cpp (без изменений), FrameWnd.hpp (без изменений), FrameWnd.cpp (без изменений). Перечислен-

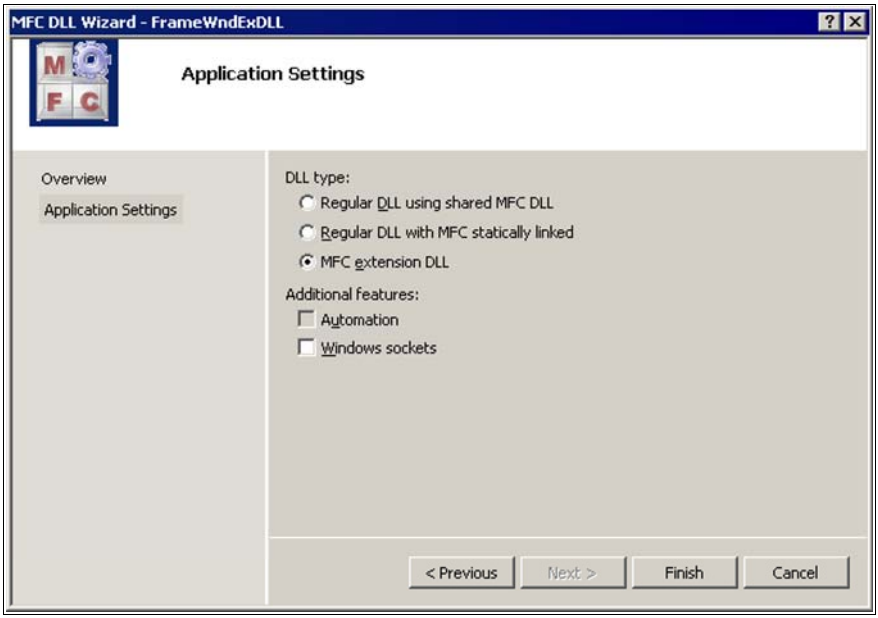


Рис. 11.2. Вид вкладки **Application Settings** проекта FrameWndExDLL

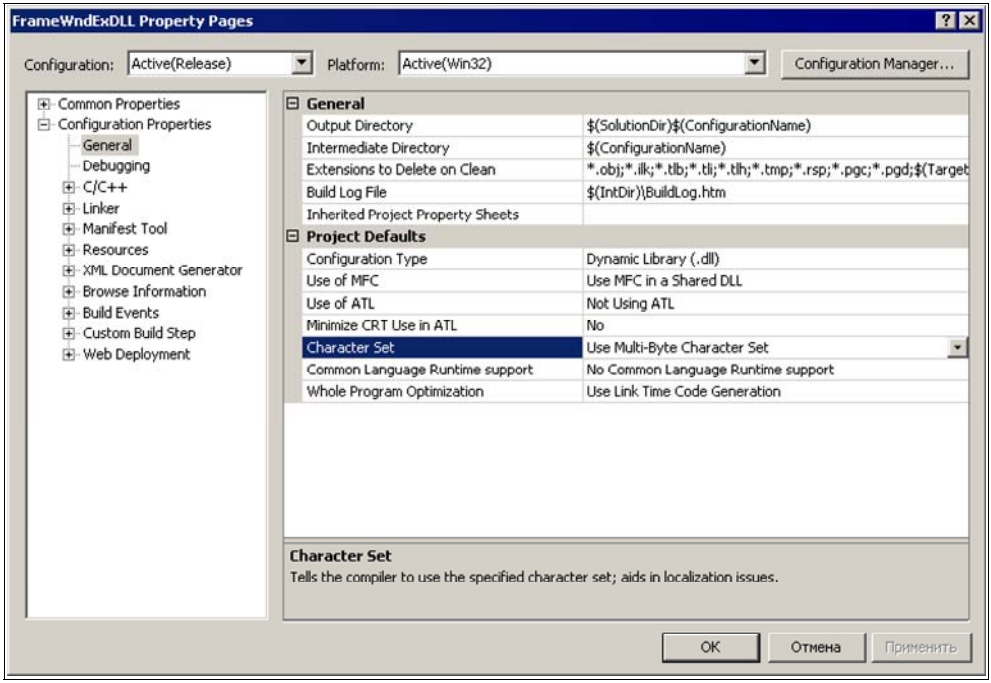


Рис. 11.3. Вид вкладки **General** проекта FrameWndExDLL

ные файлы следует скопировать в папку проекта FrameWndExDLL и включить в проект. Остальные файлы проекта сгенерированы мастером. Не забудьте задать прекомпиляцию стандартных заголовочных файлов так, как это было указано в *разд. 4.5*. Файловый состав проекта FrameWndExDLL показан на рис. 11.4.

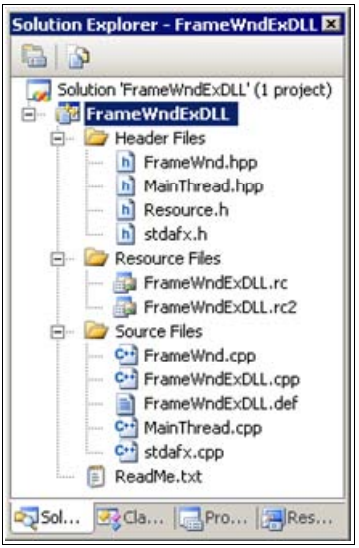


Рис. 11.4. Файловый состав проекта FrameWndExDLL

Для создания DLL расширений (файлы FrameWndExDLL.dll и FrameWndExDLL.lib папки **Release**) достаточно выполнить команду **Build**.

Демонстрационный проект имеется на прилагаемом компакт-диске в папке \Глава 11\FrameWndExDLL. Содержимое новых файлов проекта StdAfx.h, StdAfx.cpp, FrameWndExDLL.cpp, FrameWndExDLL.def, сгенерированных мастером, и модифицированного файла MainThread.hpp из демонстрационного проекта FrameWmd_PCH *(см. главу 4)* приведено в листингах 11.1—11.5 соответственно, а файлов MainThread.cpp, FrameWnd.hpp и FrameWnd.cpp — в ранее листингах 4.2—4.4 соответственно *(см. главу 4)*.

Листинг 11.1. Файл StdAfx.h

```
// Этот файл сгенерирован мастером и не редактируется

// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
```

```
#pragma once

#ifndef VC_EXTRALEAN
#define VC_EXTRALEAN        // Exclude rarely-used stuff from
                            //    Windows headers
#endif

// Modify the following defines if you have to target a platform
//   prior to the ones specified below.
// Refer to MSDN for the latest info on corresponding values for
//   different platforms.
#ifndef WINVER              // Allow use of features specific to
                            //   Windows XP or later.
#define WINVER 0x0501       // Change this to the appropriate value
                            //   to target other versions of
                            //    Windows.
#endif

#ifndef _WIN32_WINNT        // Allow use of features specific to
                            //    Windows XP or later.
#define _WIN32_WINNT 0x0501 // Change this to the appropriate value
                            //   to target other versions of
                            //   Windows.
#endif

#ifndef _WIN32_WINDOWS      // Allow use of features specific to
                            //   Windows 98 or later.
#define _WIN32_WINDOWS 0x0410   // Change this to the appropriate
                                //   value to target Windows Me or
                                //   later.
#endif

#ifndef _WIN32_IE           // Allow use of features specific to IE
                            //   6.0 or later.
#define _WIN32_IE 0x0600    // Change this to the appropriate value
                            //   to target other versions of IE.
#endif

#define _ATL_CSTRING_EXPLICIT_CONSTRUCTORS  // some Cstring
                                            //   constructors will
                                            //   be explicit

#include <afxwin.h>         // MFC core and standard components
#include <afxext.h>         // MFC extensions
```

```
#ifndef _AFX_NO_OLE_SUPPORT
#include <afxole.h>         // MFC OLE classes
#include <afxodlgs.h>       // MFC OLE dialog classes
#include <afxdisp.h>        // MFC Automation classes
#endif // _AFX_NO_OLE_SUPPORT

#ifndef _AFX_NO_DB_SUPPORT
#include <afxdb.h>          // MFC ODBC database classes
#endif // _AFX_NO_DB_SUPPORT

#ifndef _AFX_NO_DAO_SUPPORT
#include <afxdao.h>         // MFC DAO database classes
#endif // _AFX_NO_DAO_SUPPORT

#ifndef _AFX_NO_OLE_SUPPORT
#include <afxdtctl.h>       // MFC support for Internet Explorer 4
                            //   Common Controls
#endif
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>         // MFC support for Windows Common
                            //   Controls
#endif // _AFX_NO_AFXCMN_SUPPORT
```

Листинг 11.2. Файл StdAfx.cpp

```
// Этот файл сгенерирован мастером и не редактируется

// stdafx.cpp : source file that includes just the standard includes
// FrameWndExDLL.pch will be the pre-compiled header
// stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"
```

Листинг 11.3. Файл FrameWndExDLL.cpp

```
/*
   Этот файл сгенерирован мастером, но снабжен комментариями на
русском языке.
   DLL расширений строится для проекта FrameWmd_PCH, из которого
взяты файлы MainThread.hpp (добавлен макрос в заголовок класса),
MainThread.cpp (без изменений), FrameWnd.hpp (без изменений),
FrameWnd.cpp (без изменений). Остальные файлы проекта сгенерированы
мастером.
```

```
   Microsoft Visual Studio C++ .NET 2005
*/

// FrameWndExDLL.cpp : Defines the initialization routines for the
//   DLL.

#include "stdafx.h"                 // Прекомпилируемый файл
#include <afxdllx.h>                // Поддержка DLL расширений

#ifdef _MANAGED
#error Please read instructions in FrameWndExDLL.cpp to compile
// If you want to add /clr to your project you must do the
//   following:
//    1. Remove the above include for afxdllx.h
//    2. Add a .cpp file to your project that does not have /clr
//       thrown and has Precompiled headers disabled, with the
         following text:
//       #include <afxwin.h>
//       #include <afxdllx.h>
#endif

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


static AFX_EXTENSION_MODULE FrameWndExDLLDLL = { NULL, NULL };
/*
   Комментарий добавлен из справочной системы.
   The AFX_EXTENSION_MODULE is used during initialization of MFC
extension DLLs to hold the state of extension DLL module.

   struct AFX_EXTENSION_MODULE
   {
     BOOL bInitialized;
     HMODULE hModule;
     HMODULE hResource;
     CRuntimeClass* pFirstSharedClass;
     COleObjectFactory* pFirstSharedFactory;
   };
Parameters
   bInitialized — TRUE if the DLL module has been initialized with
AfxInitExtensionModule.
```

```
   hModule — specifies the handle of the DLL module.
   hResource — specifies the handle of the DLL custom resource
module.
   pFirstSharedClass — a pointer to information (the CRuntimeClass
structure) about the DLL module's first runtime class. Used to
provide the start of the runtime class list.
   pFirstSharedFactory — a pointer to the DLL module's first object
factory (a COleObjectFactory object). Used to provide the start of
the class factory list.

Remarks
   MFC extension DLLs need to do two things in their DllMain
function.
   Call AfxInitExtensionModule and check the return value.
   Create a CDynLinkLibrary object if the DLL will be exporting
CRuntimeClass objects or has its own custom resources.

   The AFX_EXTENSION_MODULE structure is used to hold a copy of the
extension DLL module state, including a copy of the runtime class
objects that have been initialized by the extension DLL as part of
normal static object construction executed before DllMain is
entered.
*/

#ifdef _MANAGED
#pragma managed(push, off)
#endif

extern "C" int APIENTRY
DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
{
    // Remove this if you use lpReserved
    UNREFERENCED_PARAMETER(lpReserved);

    if (dwReason == DLL_PROCESS_ATTACH)
    {
        TRACE0("FrameWndExDLL.DLL Initializing!\n");

        // Extension DLL one-time initialization
        if (!AfxInitExtensionModule(FrameWndExDLLDLL, hInstance))
            return 0;
```

```
        // Insert this DLL into the resource chain
        // NOTE: If this Extension DLL is being implicitly linked to
        //  by an MFC Regular DLL (such as an ActiveX Control)
        //  instead of an MFC application, then you will want to
        //  remove this line from DllMain and put it in a separate
        //  function exported from this Extension DLL. The Regular
        //   DLL that uses this Extension DLL should then explicitly
        //   call that function to initialize this Extension DLL.
        // Otherwise, the CDynLinkLibrary object will not be
        //  attached to the Regular DLL's resource chain, and
        //  serious problems will result.

        CDynLinkLibrary *pCDynLinkLibrary =
            new CDynLinkLibrary( FrameWndExDLLDLL );
        // Обработка ошибки размещения
        ASSERT_VALID( pCDynLinkLibrary );
    }
    else if (dwReason == DLL_PROCESS_DETACH)
    {
        TRACE0("FrameWndExDLL.DLL Terminating!\n");

        // Terminate the library before destructors are called
        AfxTermExtensionModule(FrameWndExDLLDLL);
    }
    return 1;                       // ok
}

#ifdef _MANAGED
#pragma managed(pop)
#endif
```

Листинг 11.4. Файл FrameWndExDLL.def

```
// Этот файл сгенерирован мастером и не редактируется

; FrameWndExDLL.def : Declares the module parameters for the DLL.

LIBRARY      "FrameWndExDLL"

EXPORTS
    ; Explicit exports can go here
```

Листинг 11.5. Файл MainThread.hpp

```
/*
   Файл                : MainThread.hpp

   Проект              : любой проект на основе библиотеки классов
                         MFC

   Назначение          : объявление класса "MainThread",
                         производного от класса "CWinApp" библиотеки
                         классов MFC (инициализация приложения перед
                         запуском)

   Microsoft Visual Studio C++ .NET 2005
*/

// Предотвращение возможности многократного подключения данного
//   файла
#ifndef _MainThread_hpp
    #define _MainThread_hpp

    // Объявление класса: обратите внимание на макрос AFX_EXT_CLASS
    //   для экспорта класса
    class AFX_EXT_CLASS MainThread : public CWinApp
    {
        // Методы

    public:

        // Инициализация приложения перед запуском
        virtual BOOL InitInstance( void );
    };

#endif
```

В DLL расширений (см. листинг 11.3, текст соответствующего файла сгенерирован мастером) сначала необходимо выполнить процедуры *инициализации*. При инициализации DLL расширений используется структура `AFX_EXTENSION_MODULE` библиотеки MFC. Функция `DLLMain` вызывает функцию `AfxInitExtensionModule` библиотеки MFC, которая и инициализирует DLL. Затем в памяти размещается новый объект класса `CDynLinkLibrary`, производного от класса `CCmdTarget`, и, тем самым, DLL включается в цепочку ресурсов.

11.2.2. Тестирование DLL расширений для приложения без ресурсов на базе MFC. Демонстрационный проект TestFrameWndExDLL

Для тестирования созданной DLL расширений выполните следующие действия: в IDE на вкладке **Start Page** выберите **Create: Project...**, в появившемся окне **New Project** укажите свойства проекта в соответствии с рис. 11.5 и нажмите кнопку **OK**.

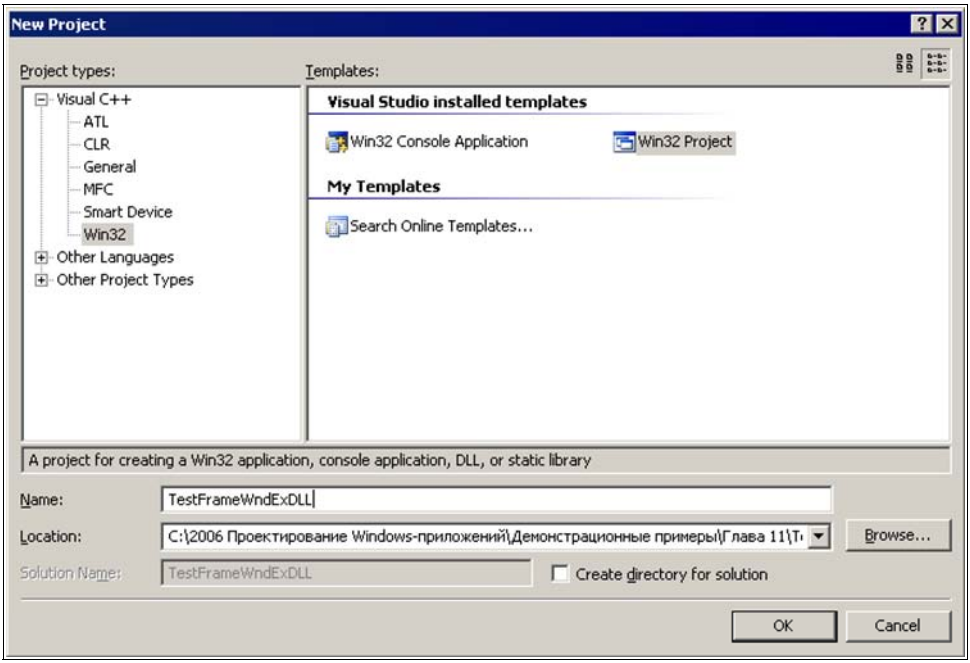


Рис. 11.5. Свойства проекта TestFrameWndExDLL

В результате появится окно **MFC Application Wizard — TestFrameWndExDLL**, в котором следует выбрать вкладку **Application Settings** и настроить ее так, как показано на рис. 11.6, и нажать кнопку **Finish.**

В окне **Solution Explorer**, для проекта TestFrameWndExDLL, вызовите контекстное меню, выберите **Properties** и в появившемся окне **TestFrameWndExDLL Property Pages** выберите вкладку **General**. Настройте ее в соответствии с рис. 11.7 и нажмите кнопку **OK**.

В папку проекта TestFrameWndExDLL из рассмотренного ранее проекта FrameWndExDLL копируются файлы stdafx.h, stdafx.cpp, MainThread.hpp (в этом случае макрос `AFX_EXT_CLASS` в заголовке объявления класса

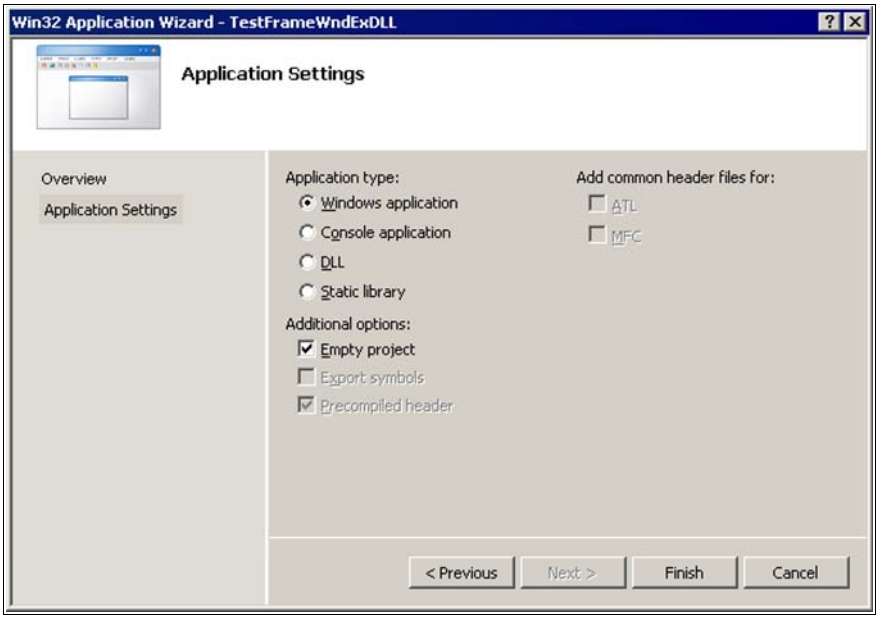


Рис. 11.6. Вид вкладки **Application Settings** проекта TestFrameWndExDLL

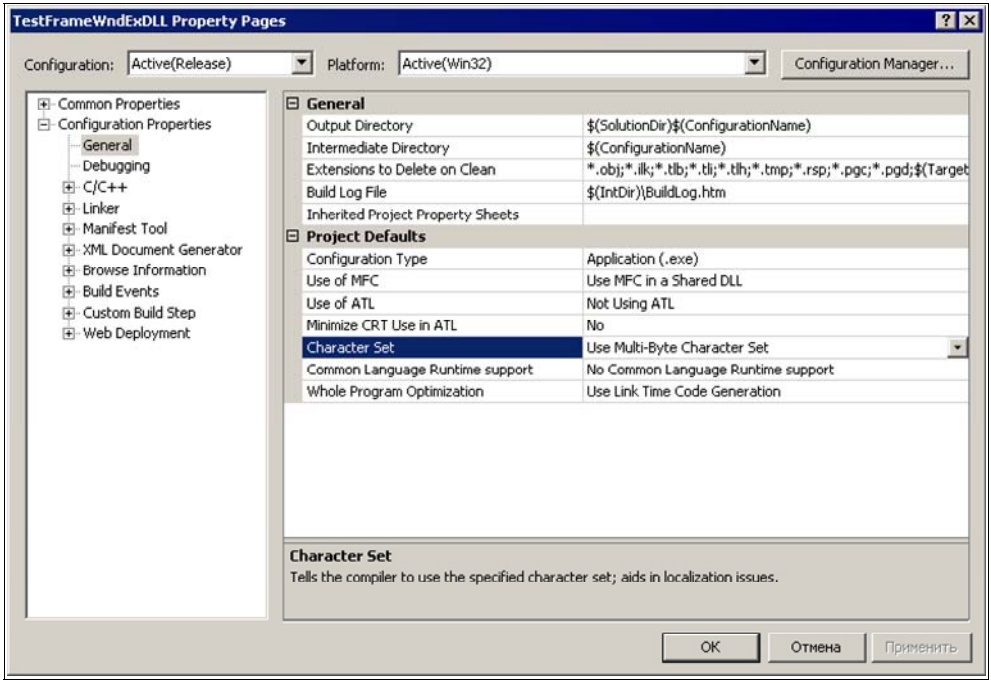


Рис. 11.7. Вид вкладки **General** проекта TestFrameWndExDLL

не нужен), FrameWndExDLL.lib и FrameWndExDLL.dll. Из проекта FrameWnd_PCH *(см. главу 4)* — файл Main.cpp с некоторыми изменениями (листинг 11.6). Все указанные файлы, кроме файла FrameWndExDLL.dll, включаются в проект TestFrameWndExDLL и задается прекомпиляция стандартных заголовочных файлов так, как это было указано в *разд. 4.5*.

Если в соответствии с действующими настройками при включении в проект файла FrameWndExDLL.lib появляется окно **Matching Custom Build Rule Not Found** с запросом, то ответьте нажатием кнопки **No**.

Листинг 11.6. Файл Main.hpp

```
/*
   Файл                : Main.cpp

   Проект              : тестирование DLL расширений
                         FrameWndExDLL.dll, написанной с
                         использованием MFC (создание файлов
                         FrameWndExDLL.dll и FrameWndExDLL.lib
                         выполняется в проекте FrameWndExDLL)

   Назначение          : объявление, создание и запуск приложения

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAfx.h"                  // Прекомпилируемый файл
// Объявление класса MainThread из FrameWndExDLL.dll — обратите
//   внимание на необходимость включения файла FrameWndExDLL.lib в
//   состав проекта
#include "MainThread.hpp"

// Инициализация и запуск приложения
MainThread              MainThread;
```

Файловый состав проекта TestFrameWndExDLL показан на рис. 11.8.

Для построения выполняемого файла для проекта TestFrameWndExDLL достаточно выполнить команду **Build TestFrameWndExDLL**, а для запуска проекта — команду **Start Without Debugging** или команду **Start Debugging**. Главное окно приложения показано на рис. 11.9.

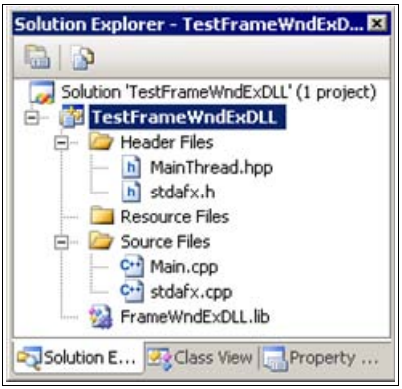


Рис. 11.8. Файловый состав проекта TestFrameWndExDLL

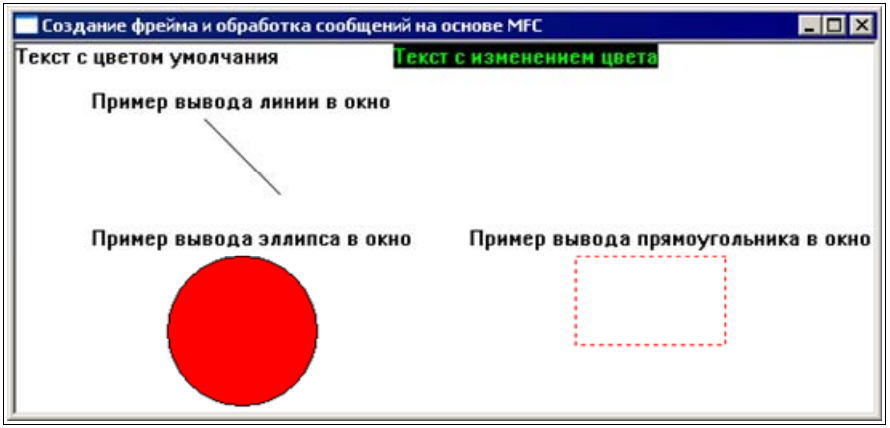


Рис. 11.9. Главное окно проекта TestFrameWndExDLL

11.2.3. Создание и тестирование DLL расширений для приложения с ресурсами на базе MFC. Демонстрационные проекты UsgMenuExDLLResource и TestUsgMenuExDLLResource

В качестве исходного проекта при создании и тестировании DLL расширений используем демонстрационный проект UsgMenu_MFC, представляющий собой приложение с ресурсами, рассмотренное в *главе 6*.

Начнем с проекта UsgMenuExDLLResource — создадим DLL расширений для приложения с ресурсами, выполняется аналогично указанному в *разд. 11.2.1*, в соответствии с рис. 11.1—11.3.

Но имеют место и некоторые различия. Для привязки DLL расширений к демонстрационному проекту FrameWmd_PCH из *главы 4*, содержащему ресурсы, сначала необходимо из сгенерированного мастером каркаса проекта и из его папки удалить файлы resource.h, UsgMenuExDLLResource.rc и UsgMenuExDLLResource.rc2. Далее, следует скопировать в папку проекта и включить в проект файлы MainThread.hpp (с добавлением макроса `AFX_EXT_CLASS` в заголовок класса), MainThread.cpp (без изменений), FrameWnd.hpp (без изменений), FrameWnd.cpp (без изменений), resource.h (без изменений), Resource.rc (без изменений) из проекта FrameWnd_PCH. Не забудьте задать прекомпиляцию стандартных заголовочных файлов так, как это было указано ранее в *разд. 4.5*. В результате получаем файловый состав проекта UsgMenuExDLLResource, показанный на рис. 11.10.

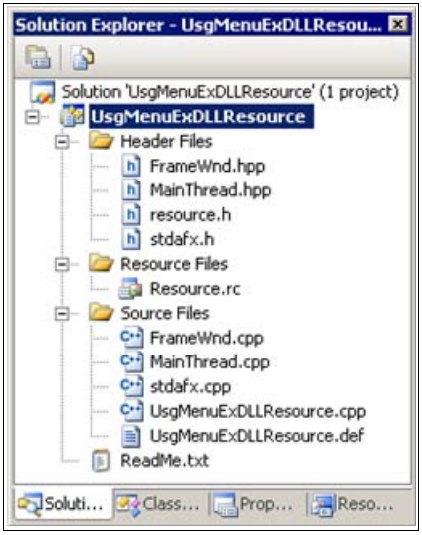


Рис. 11.10. Файловый состав проекта UsgMenuExDLLResource

Для создания DLL расширений (получения файлов UsgMenuExDLLResource.dll и UsgMenuExDLLResource.lib в папке **Release**) достаточно выполнить команду **Build**.

Демонстрационный проект UsgMenuExDLLResource находится на прилагаемом компакт-диске, в папке \Глава 11\UsgMenuExDLLResource.

Теперь перейдем к демонстрационному проекту TestUsgMenuExDLLResource. Тестирование ранее созданной DLL расширений с помощью демонстрационного проекта TestUsgMenuExDLLResource можно выполнить аналогично указанному в *разд. 11.2.2*, в соответствии с рис. 11.5—11.7. Далее, в папку проекта TestUsgMenuExDLLResource из проекта UsgMenuExDLLResource копи-

руются файлы stdafx.h, stdafx.cpp, MainThread.hpp (в этом случае макрос AFX_EXT_CLASS в заголовке объявления класса не нужен), UsgMenuExDLLResource.lib и UsgMenuExDLLResource.dll, из проекта FrameWnd_PCH из *главы 4* — файл Main.cpp с модификацией комментариев (листинг 11.7). Все указанные файлы, кроме файла FrameWndExDLL.dll, включаются в проект TestUsgMenuExDLLResource и задается прекомпиляция стандартных заголовочных файлов так, как это было указано ранее в *разд. 4.5*. Если, в соответствии с действующими настройками, при включении в проект файла UsgMenuExDLLResource.lib появляется окно **Matching Custom Build Rule Not Found** с запросом, то ответьте нажатием кнопки **No**.

Листинг 11.7. Файл Main.hpp

```
/*
   Файл               : Main.cpp

   Проект             : тестирование DLL расширений
                        UsgMenuExDLLResource.dll, написанной с
                        использованием MFC (создание файлов
                        UsgMenuExDLLResource.dll
                        UsgMenuExDLLResource.lib
                        выполняется в проекте UsgMenuExDLLResource)

   Назначение         : объявление, создание и запуск приложения

   Microsoft Visual Studio C++ .NET 2005
*/

#include "StdAfx.h"                  // Прекомпилируемый файл
// Объявление класса MainThread из FrameWndExDLL.dll — обратите
//   внимание на необходимость включения файла FrameWndExDLL.lib в
//   состав проекта
#include "MainThread.hpp"            // Класс MainThread

// Инициализация и запуск приложения
MainThread              MainThread;
```

Файловый состав проекта TestUsgMenuExDLLResource представлен на рис. 11.11.

Для построения проекта TestUsgMenuExDLLResource достаточно выполнить команду **Build UsgMenuExDLLResource**, а для запуска проекта — команду **Start Without Debugging** или команду **Start Debugging**. Главное окно приложения представлено на рис. 11.12.

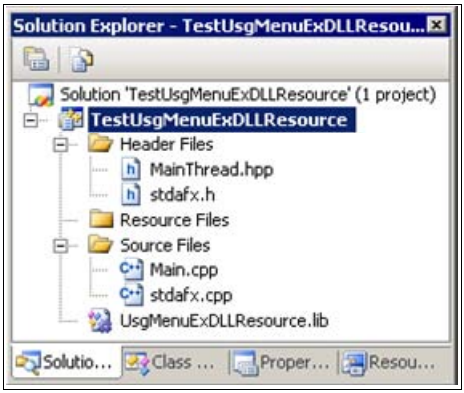


Рис. 11.11. Файловый состав проекта TestUsgMenuExDLLResource

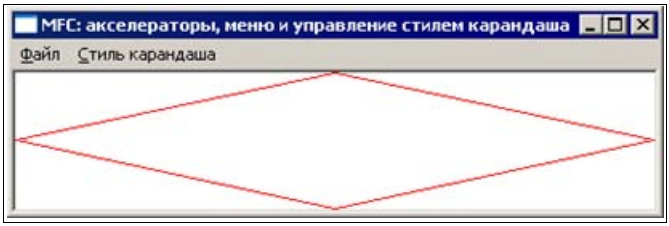


Рис. 11.12. Главное окно проекта TestUsgMenuExDLLResource

11.3. Вопросы и упражнения для самопроверки

Для самопроверки ответьте на следующие вопросы:

1. Что представляет собой динамически подключаемая библиотека?
2. Почему создавать DLL достаточно просто?
3. Какие типы DLL поддерживаются библиотекой классов MFC? Дайте их характеристику.
4. Как в DLL расширений обеспечивается экспорт класса?
5. Для чего используется функция `DLLMain`? Когда она вызывается?
6. Что представляют собой процесс и поток (потоки) выполнения?
7. Какой интерфейс имеет функция `DLLMain`?
8. Какие значения может принимать параметр функции `DLLMain`, показывающий причину ее вызова?

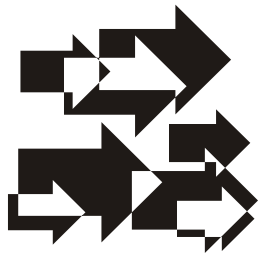
9. Как должен выглядеть каркас функции `DLLMain`? Всегда ли нужно писать ее в исходном коде?
10. Как создать DLL расширений с помощью мастера?

Ответы на вопросы можно проверить — они приведены в *разд. П1.11 приложения 1*.

Для закрепления учебного материала, изложенного в этой главе, выполните следующие упражнения:

1. Повторите создание и тестирование DLL расширений для демонстрационного проекта FrameWnd_PCH без ресурсов, рассмотренного в *главе 4*.
2. Повторите создание и тестирование DLL расширений для демонстрационного проекта UsgMenu_MFC с ресурсами, рассмотренного в *главе 6*.

ГЛАВА 12



Создание каркаса приложения на базе MFC с помощью мастера и его русификация [7]

IDE Microsoft Visual Studio 2005 содержит мощное средство создания приложений на базе MFC — мастер MFC Application Wizard, использующий библиотеку классов MFC. Рассмотрим как с его помощью создать каркас приложения и, попутно, дадим характеристику возможностей этого мастера.

12.1. Создание каркаса приложения на базе MFC. Демонстрационные проекты MDIApp и SDIApp

Для получения экземпляра каркаса многооконного MDI-приложения (MDI, Multilpe Document Interface — многооконный интерфейс) на базе MFC (далее, просто — приложения) с помощью MFC Application Wizard нужно выполнить следующие действия. В IDE, в окне **Recent Projects** (Недавние проекты) вкладки **Start Page** (Стартовая страница), выберите **Create:Project...** (Создать:проект). В появившемся окне **New Projects** (Новые проекты) задайте параметры проекта в соответствии с рис. 12.1 и нажмите кнопку **OK**.

В результате появится окно **MFC Application Wizard — MDIApp**, позволяющее выполнить требуемую настройку каркаса приложения (рис. 12.2).

В окнах мастера с помощью кнопки, с пиктограммой в виде знака вопроса **?**, или клавиши <F1> можно получить справочную информацию по любому из параметров настройки приложения. Для этого достаточно нажать кнопку **?** или нажать клавишу <F1>. В результате появится окно **MFC Application Wizard — Microsoft Visual Studio 2005 Documentation — Microsoft Document Explorer**, позволяющее получить справку по любому параметру настройки, представленному в этом окне. Есть два варианта задания параметров

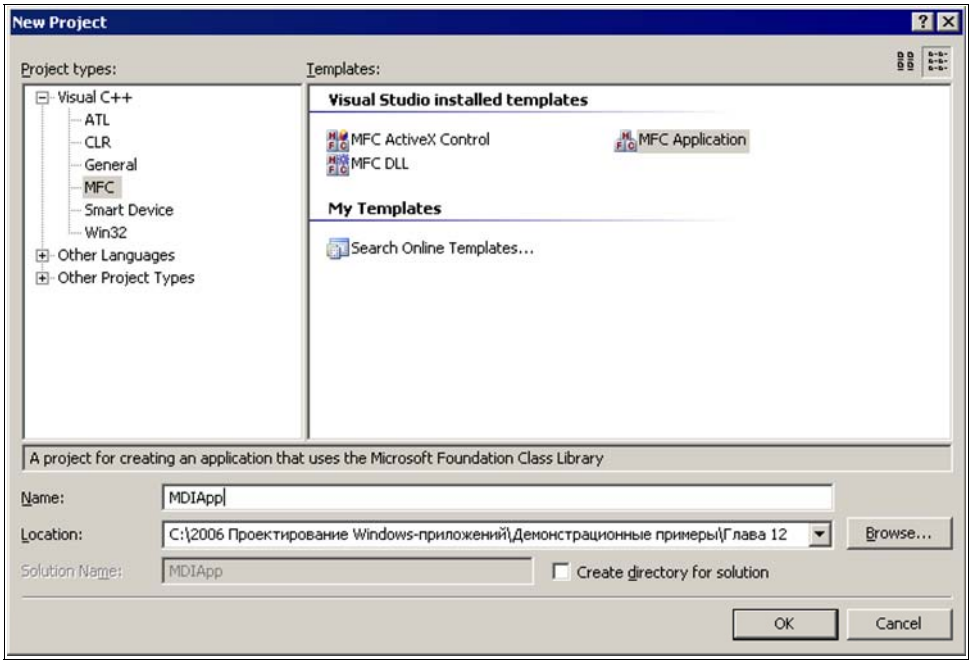


Рис. 12.1. Параметры создаваемого проекта

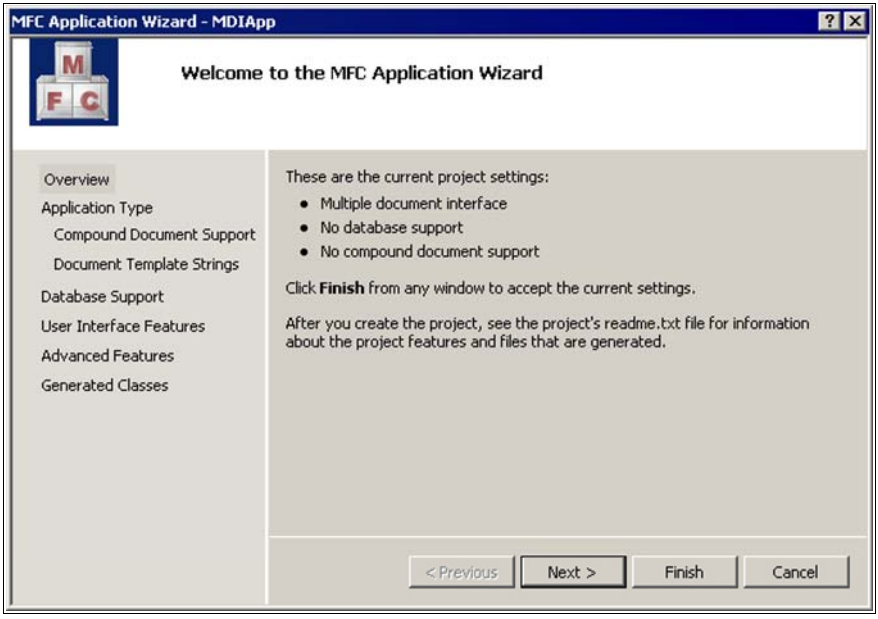


Рис. 12.2. Исходное окно настройки каркаса MFC-приложения

создаваемого каркаса — нажатием кнопки **Finish** (Закончить) принять настройки, предлагаемые мастером по умолчанию, или выполнить выборочную настройку каркаса приложения. Во втором случае следует либо выбрать вкладку **Application Type** (Тип приложения) (первую из семи оставшихся вкладок), либо нажать кнопку **Next** (Далее).

После нажатия кнопки **Next** в появившемся диалоговом окне **Application Type** (рис. 12.3) следует указать мастеру, должно ли приложение поддерживать работу с одним документом (SDI, Single Document Interface), с несколькими документами (MDI) или это будет диалоговое окно. SDI-приложение, поддерживающее работу с одним документом, создать проще всего, поскольку в этом случае не нужно учитывать взаимодействия между разными документами, одновременно открытыми приложением. Вместе с тем, MDI-приложение более универсально.

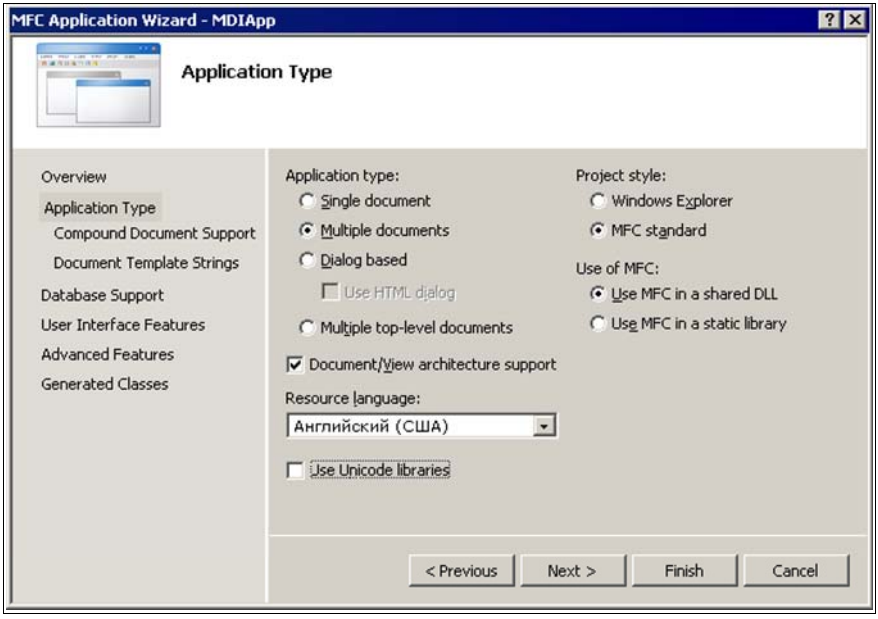


Рис. 12.3. Окно настройки **Application Type**

Поэтому согласитесь с вариантом, предлагаемым мастером. Обратите внимание на то, как меняется рисунок-пояснение в заголовке окна при изменении типа приложения. Убедитесь, что установлена предлагаемая по умолчанию опция поддержки архитектуры документ/представление **Document/View architecture support**. Чтобы использовать опцию **Use Multi-Byte Character Set** (Использовать расширенный набор символов), выключите выбираемую кнопку **Use Unicode libraries** (Использовать библиотеки Unicode). Для ос-

тальных опций согласитесь со значениями, предлагаемыми по умолчанию (английский язык для ресурсов, стандартный проект MFC, динамическое подключение библиотеки MFC). Далее, на этом и последующих шагах, есть четыре варианта. При нажатии кнопки **Finish** последующие шаги настройки приложения не выполняются и используются параметры приложения, предлагаемые мастером по умолчанию. При нажатии кнопки **Previous** (Предыдущие) происходит возврат к предыдущему окну настройки и вы можете скорректировать свои прежние решения. При нажатии кнопки **Cancel** построение приложения прекращается. И, наконец, при нажатии кнопки **Next** выполняется переход к следующему окну настройки. Щелкните по кнопке **Next**. На экране появится следующее окно настройки **Compound Document Support**, показанное на рис. 12.4.

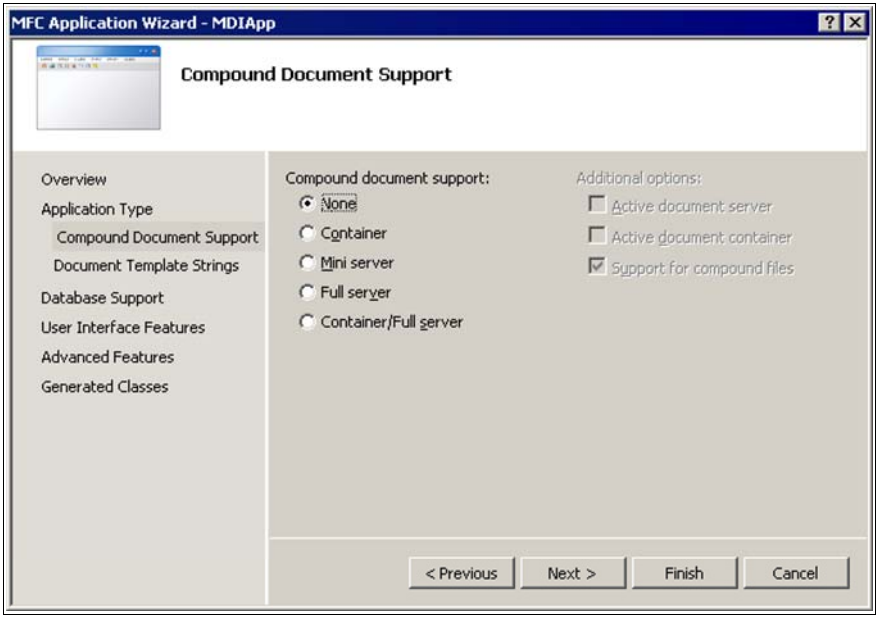


Рис. 12.4. Окно настройки **Compound Document Support**

Теперь можно выбрать вариант связывания документов (контейнер и/или сервер). Для получения об этом дополнительной информации вы можете воспользоваться справочной системой. Рассмотрение указанных вопросов выходит за рамки учебного пособия. Поэтому оставьте активной опцию **None** (Ничего), предлагаемую по умолчанию, и щелкните по кнопке **Next**, чтобы перейти к следующему окну настройки (рис. 12.5).

В окне настройки **Document Template Strings** (Параметры документов) можно задать параметры типа документа, связанного с приложением. Это позво-

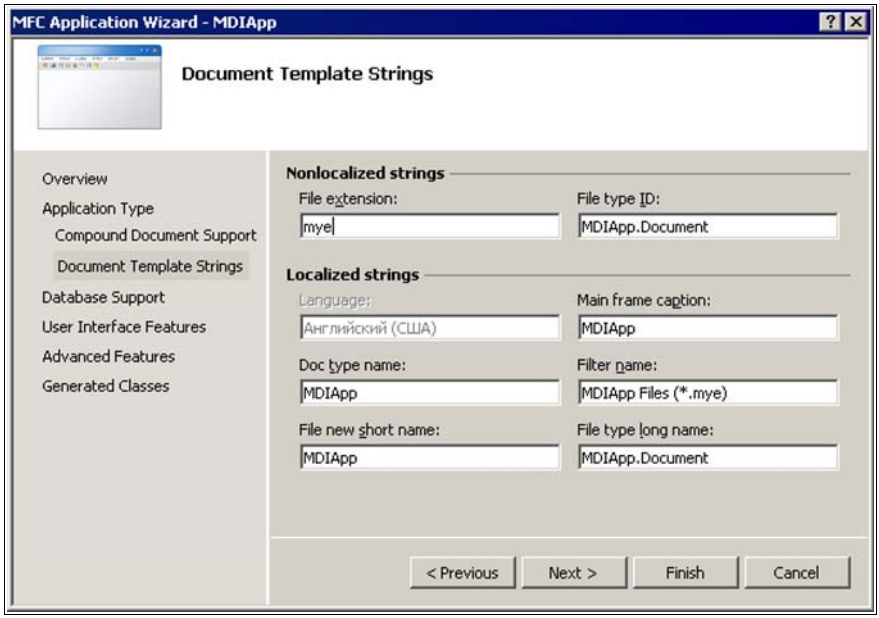


Рис. 12.5. Окно настройки **Document Template Strings**

лит Windows связать данный тип документа с вашим приложением. В дальнейшем, когда пользователь дважды щелкнет на имени файла, операционная система автоматически запустит приложение, связанное с ним. Подробнее о параметрах этого окна можно узнать, используя кнопку **?** или клавишу <F1>. В поле **File extension** (Расширение файла) введите расширение mye и нажмите кнопку **Next**, чтобы перейти к следующему окну настройки (рис. 12.6).

Установки в окне настройки **Database Support** (Поддержка баз данных) делаются только в том случае, если в приложении будет осуществляться работа с базами данных. В нашем примере следует оставить опцию **None**, предлагаемую по умолчанию. Щелкните на кнопке **Next** и вы перейдете к окну настройки **User Interface Features** (Пользовательский интерфейс) (рис. 12.7).

С помощью этого окна можно добавлять в программу различные средства, определяющие интерфейс приложения и параметры главного и дочерних окон приложения. К их числу относятся, например, опция главного окна **Tick frame**, позволяющая изменять размеры главного окна с помощью рамки, опции создания кнопок минимизации и максимизации (главное окно и дочерние окна) и опции создания в главном окне системного меню, панели инструментов, строки статуса и др.

Щелкните на кнопке **Next** и вы перейдете к окну настройки **Advanced Features** (Дополнительно) (рис. 12.8).

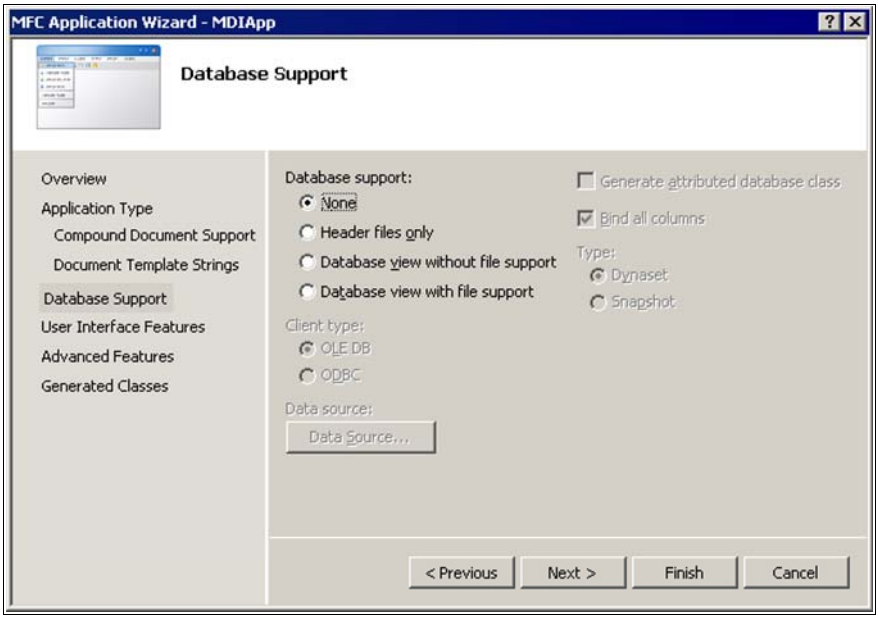


Рис. 12.6. Окно настройки **Database Support**

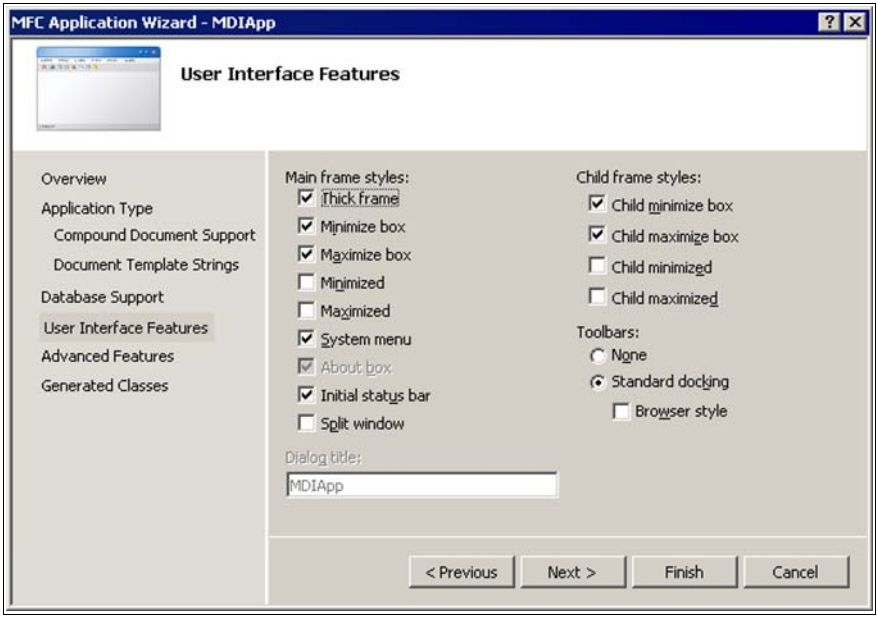


Рис. 12.7. Окно настройки **User Interface Features**

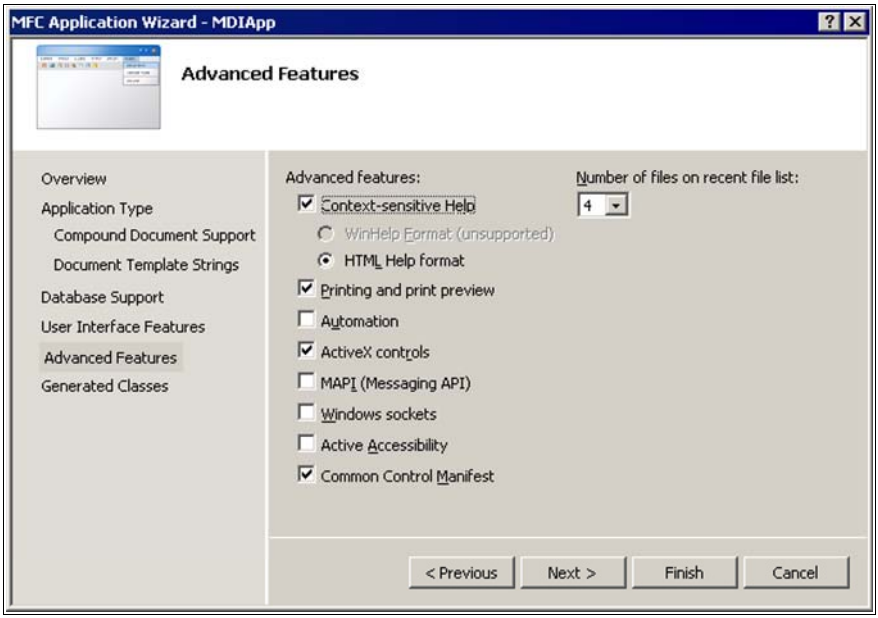


Рис. 12.8. Окно настройки **Advanced Features**

Окно **Advanced Features** позволяет задать контекстно-зависимую помощь, поддержку печати, вид представления управляющих элементов и т. п. Включите контекстно-зависимую помощь, а для других параметров примите значения, предлагаемые мастером по умолчанию.

Щелкните на кнопке **Next** и вы перейдете к последнему окну настройки **Generated Classes** (Сгенерированные классы) (рис. 12.9).

В последнем окне мастера отображается список классов, которые будут сгенерированы для нашего приложения (`CMDIAppView`, `CMDIAppApp`, `CMainFrame`, `CChildFrame` и `CMDIAppDoc`). Для класса `CMDIAppView`, реализующего функции области просмотра, в списке **Base class** (Базовый класс) можно выбрать базовый класс (`CEditView`, `CFormView`, `CHtmlEditView`, `CHtmlView`, `CListView`, `CRichEditView`, `CScrollView`, `CTreeView` или `CView`), от которого он будет порожден. Класс `CView`, потомок класса `CWnd`, является родительским для всех перечисленных классов. Если приложение поддерживает работу с одним документом (SDI-приложение), то будет использоваться одна область просмотра. Иначе, если приложение поддерживает работу с несколькими документами (MDI-приложение), то будут использованы несколько переключаемых областей просмотра. Область просмотра служит буфером между документом и пользователем и представляет собой дочернее окно главного окна приложения. Она содержит графический образ документа, выводимого на экран или

принтер, а также поддерживает возможность редактирования документа с помощью клавиатуры и мыши. Класс `CEditView` реализует функции текстового редактора и будет использован в нашем приложении. После щелчка по кнопке **Finish** будут сгенерированы файлы приложения, в том числе файл ReadMe.txt с итоговым отчетом, подготовленным мастером.

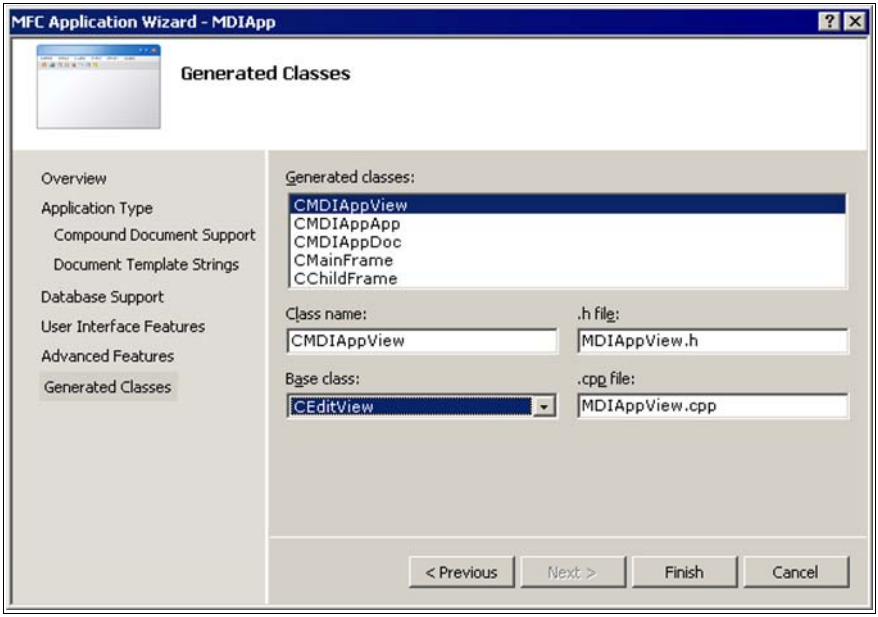


Рис. 12.9. Окно настройки **Generated Classes**

В процессе создания кода программы, в папку проекта MDIApp, будут добавлены разные подпапки и файлы, необходимые для функционирования приложения. Файловый состав проекта, сгенерированного MFC Application Wizard, показан на рис. 12.10, а главное окно приложения, после его запуска и выбора меню **File**, представлено на рис. 12.11.

Вид окна приложения после выбора других меню показан на рис. 12.12.

Если в главном окне MDI-приложения (рис. 12.11) вы закроете единственное дочернее окно редактирования **MDIApp1**, то вид окна и его строка меню изменятся (рис. 12.13).

Совет

Рекомендуем поэкспериментировать с этим проектом. Не пожалейте на это времени и изучите его возможности. Повыбирайте команды меню, понажимайте каждую из кнопок и уясните, что при этом происходит. Сделать это не трудно,

т. к., скорее всего, вы столкнетесь с уже знакомыми вещами. Для экспериментов можно использовать копию демонстрационного проекта MDIApp, имеющегося на прилагаемом компакт-диске, в папке \Глава 12\MDIApp.

Для закрепления материала аналогичным образом, с помощью MFC Application Wizard, создайте демонстрационный проект SDIApp, поддерживающий работу с единственным дочерним окном. Поэкспериментируйте с созданным проектом, сравните его с одноименным демонстрационным проектом, имеющемся на прилагаемом компакт-диске.

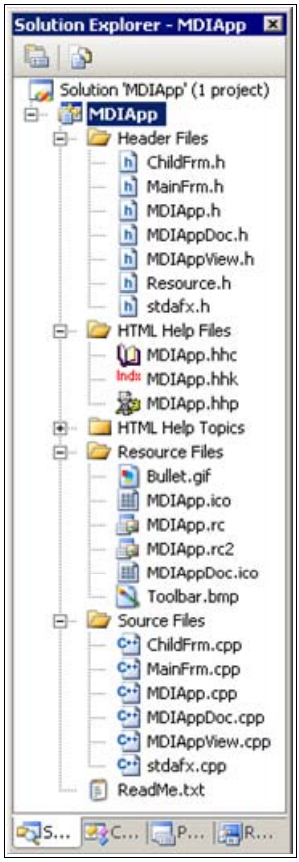


Рис. 12.10. Файловый состав демонстрационного проекта MDIApp, полученного с помощью MFC Application Wizard

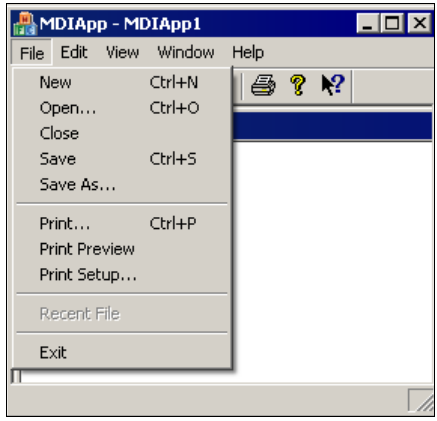


Рис. 12.11. Главное окно демонстрационного проекта MDIApp после выбора меню **File**

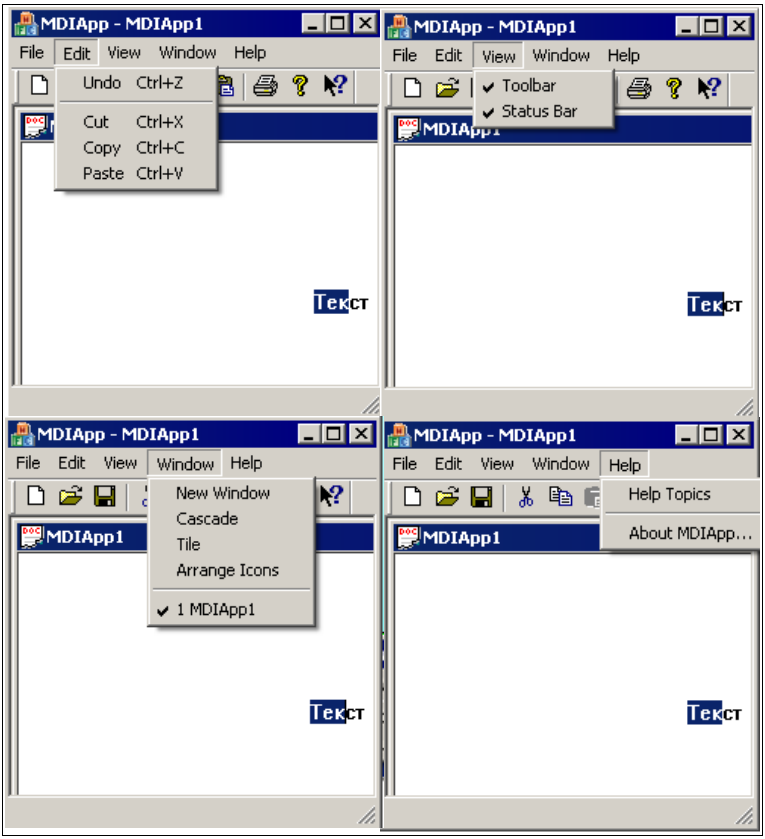


Рис. 12.12. Главное окно демонстрационного проекта MDIApp после выбора других меню

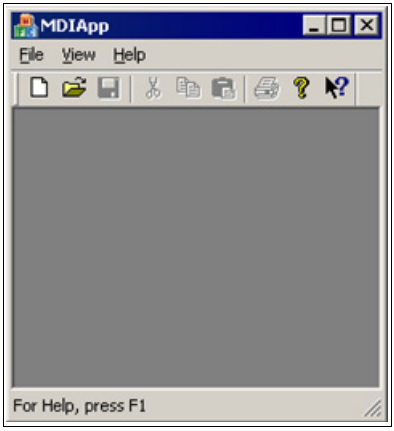


Рис. 12.13. Главное окно демонстрационного проекта MDIApp при отсутствии дочерних окон

12.2. Настройка ресурсов каркаса приложения на базе MFC, полученного с помощью MFC Application Wizard

В качестве настройки ресурсов далее рассматривается русификация ресурсов каркаса приложения на базе MFC. При русификации в качестве исходного проекта используем рассмотренный демонстрационный проект MDIApp, полученный с помощью MFC Application Wizard. Для этого создадим новую папку MDIApp1 и скопируем в нее содержимое папки \Глава 12\.MDIApp с компакт-диска. При этом имя проекта в новой папке осталось прежним, но чтобы отличить его от предыдущего проекта, будем называть новый проект демонстрационным проектом MDIApp1.

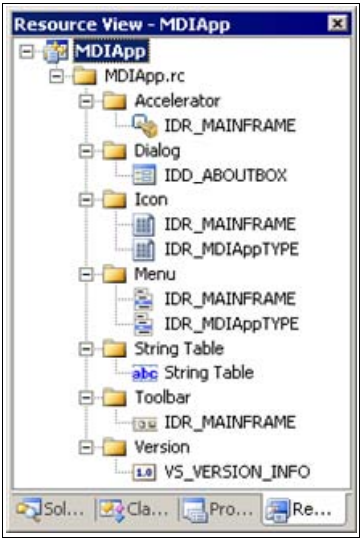


Рис. 12.14. Ресурсы демонстрационного проекта MDIApp1

Прежде всего, необходимо русифицировать все ресурсы демонстрационного проекта MDIApp1 (рис. 12.14). Для этого:

- выполните в IDE команду **Project | Settings...** (Настройки), в появившемся окне **MDIApp Property Pages** выберите вкладку **Resources** (Ресурсы) | **General** (Основное) (рис. 12.15), в поле **Culture** выберите язык **Русский (0x419)** и нажмите кнопку **OK**;
- если окно **Properties Window** отсутствует, то откройте его с помощью комбинации клавиш <Alt>+<Enter> или командой **View | Oher Window | Properties Window**. На вкладке **Resource View** выберите, например, ре-

сурс **Accelerator | IDR_MAINFRAME**, в его контекстном меню — **Properties** и в появившемся окне **Properties** выберите язык **Русский** (рис. 12.16). Аналогичным образом русифицируйте все остальные ресурсы.

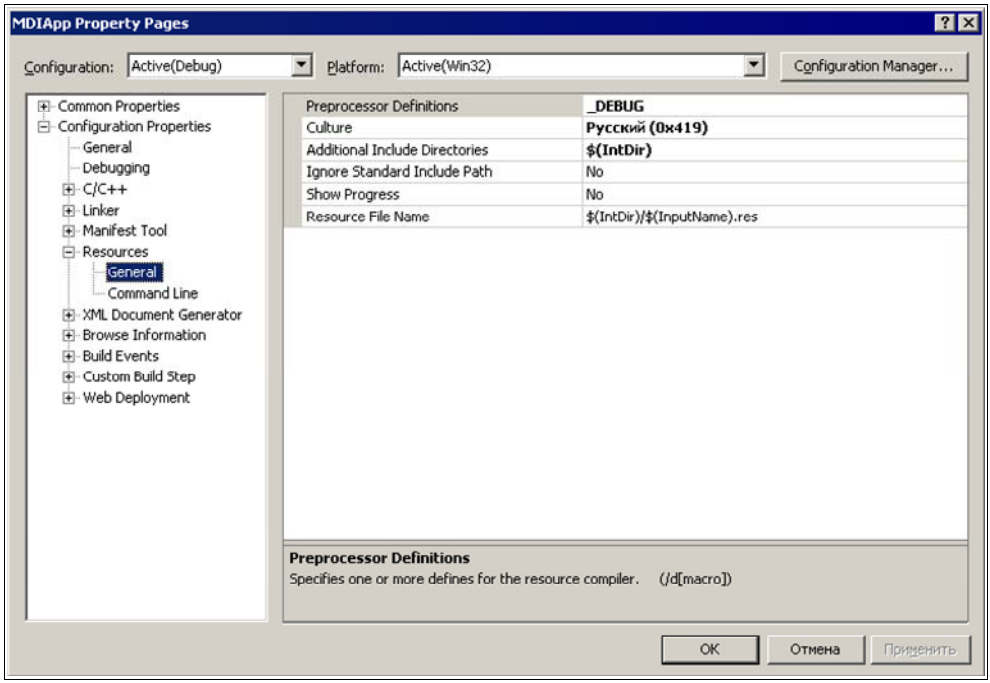


Рис. 12.15. Русификация ресурсов демонстрационного проекта MDIApp1

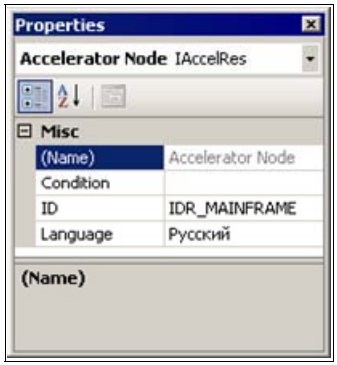


Рис. 12.16. Русификация ресурса **Accelerator | IDR_MAINFRAME** демонстрационного проекта MDIApp1

В результате этого ресурсы демонстрационного проекта MDIApp1 приобретут вид, показанный на рис. 12.14.

Далее выполним русификацию строки заголовка главного окна, диалогового окна **About MDIApp...**, команд меню и кнопок панели инструментов, а также таблицы строк. Русификация разделов встроенной в приложение справочной системы будет рассмотрена в *главе 13*.

Для модификации и русификации строки заголовка главного окна перейдите на вкладку **Resource View — MDIApp**. Последовательно раскройте ресурсы **MDIApp**, **MDIApp.rc**, **String Table**, двойным щелчком мыши по ресурсу **String Table** загрузите его в окно редактирования, и, во вкладке **MDIApp.rc (String Table)**, отредактируйте заголовок в соответствии с рис. 12.17 и нажмите клавишу <Enter>.

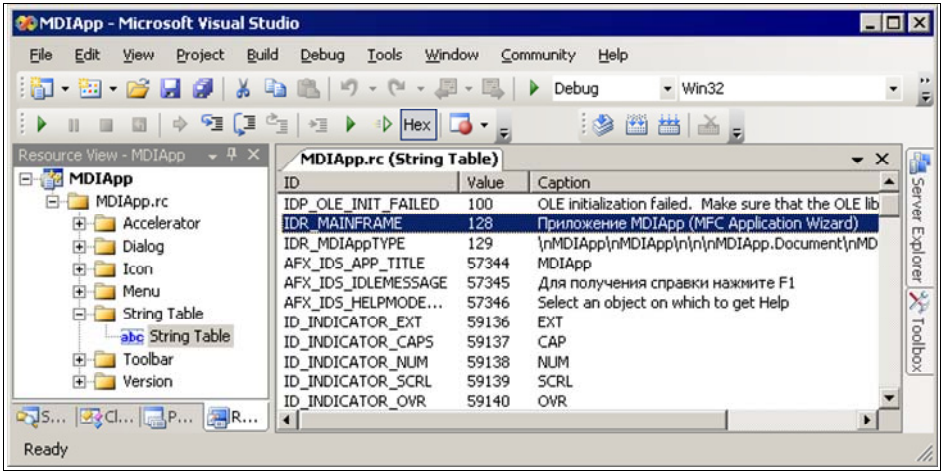


Рис. 12.17. Модификация и русификация строки заголовка демонстрационного проекта MDIApp1

На вкладке **Resource View — MDIApp** раскройте ресурс **Dialog**, двойным щелчком мыши по идентификатору **IDD_ABOUTBOX** загрузите ресурс в окно редактирования. На вкладке **MDIApp.rc (IDD_ABOUTBOX — Dialog)** настройте свойство **Caption** для шаблона диалогового окна и окон статического текста в соответствии с рис. 12.18. В результате окно приобретет вид, показанный на рис. 12.19.

Сначала рассмотрим русификацию меню на примере меню **File** главного окна приложения (см. рис. 12.11). На вкладке **Resource View — MDIApp** раскройте ресурс **Menu**, двойным щелчком мыши по идентификатору **IDR_MDIAppTYPE** загрузите ресурс меню в окно редактирования, во вкладке **MDIApp.rc (IDR_MDIAppTYPE — Menu)** выберите меню **File** и задайте его

свойство **Caption** в соответствии с рис. 12.20. Обратите внимание на то, как задается "горячая" клавиша в имени меню (на рис. 12.20 такой клавишей является <Ф>). Аналогичным образом русифицируются остальные меню каркаса MDI-приложения. При этом важно, чтобы "горячие" клавиши меню были уникальными.

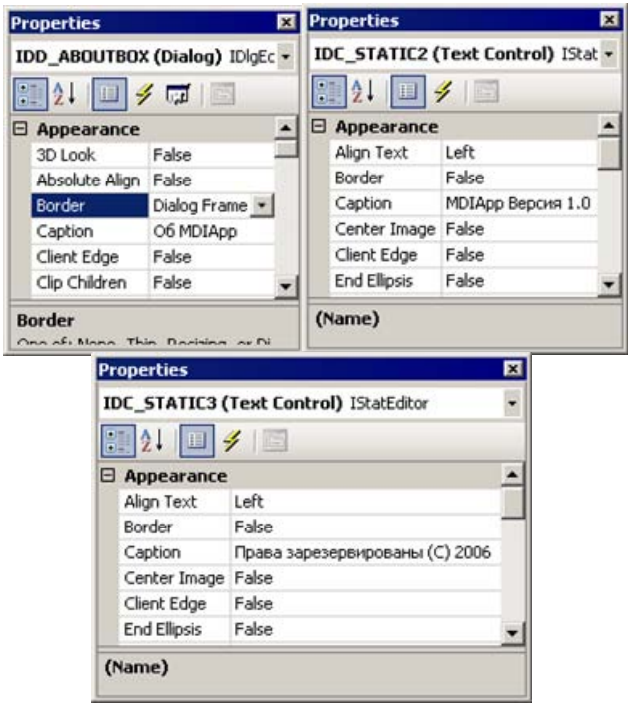


Рис. 12.18. Русификация окна **About MDIApp...** демонстрационного проекта MDIApp1

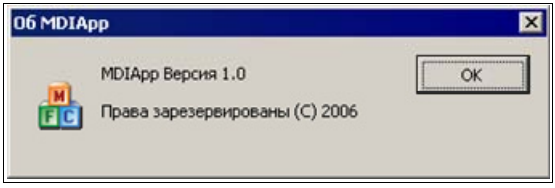


Рис. 12.19. Вид окна **About MDIApp...** демонстрационного проекта MDIApp1 после его русификации

Русификацию команд меню и кнопок панели инструментов рассмотрим на примере команды **New** меню **File** и кнопки **New** панели инструментов. В окне редактирования, во вкладке **MDIApp.rc (IDR_MDIAppTYPE — Menu)** выберите команду **New** меню **File** и настройте ее свойства в соответствии с

рис. 12.20. Обратите внимание на то, как задается "горячая" клавиша в имени команды меню (на рис. 12.20 такой клавишей является <H>). Обратите внимание на свойства **Caption**, **Prompt** и на то, что идентификаторы (свойство **ID**) у команды **New** меню **File** и кнопки **New** панели инструментов совпадают. Поэтому достаточно русифицировать только команду **New** меню **File** или кнопку **New** панели инструментов. Аналогичным образом русифицируются другие команды меню и кнопки на панели инструментов. При этом следует обеспечить уникальность "горячих" клавиш команд в рамках соответствующего меню ("горячие" клавиши команд разных меню могут совпадать).

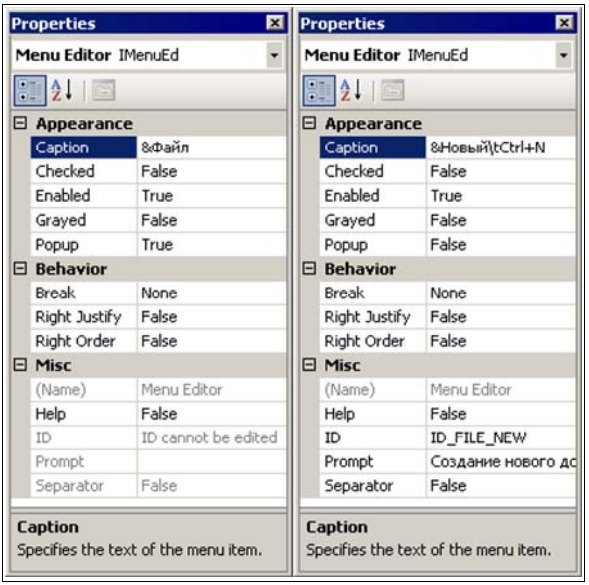


Рис. 12.20. Русификация меню **File**, команды **New** этого меню и кнопки **New** панели инструментов демонстрационного проекта MDIApp1

Для русификации некоторых элементов таблицы строк в окне редактирования выберите вкладку **MDIApp.rc (String Table)**, отредактируйте некоторые ее строки в соответствии с рис. 12.21.

Замечание

Не забудьте русифицировать второе меню каркаса MDI-приложения, используемое при отсутствии дочерних окон.

Вид главного окна демонстрационного проекта MDIApp1 после русификации, с раскрытым меню **File**, показан на рис. 12.22, а команды остальных меню продемонстрированы на рис. 12.23. Демонстрационный проект MDIApp1 содержится на прилагаемом компакт-диске, в папке \Глава 12\MDIApp1.

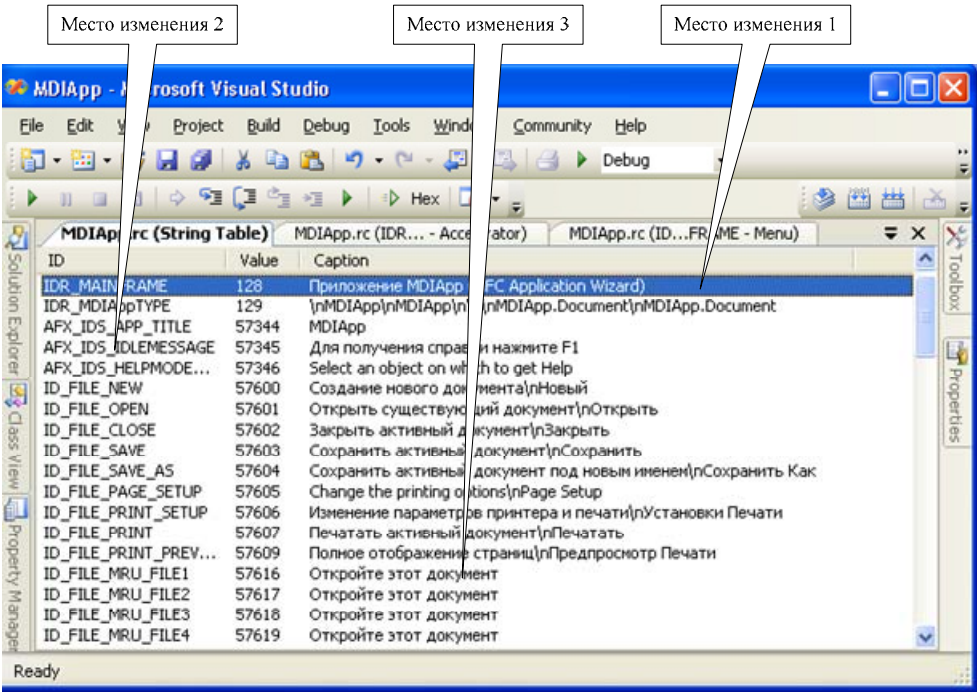


Рис. 12.21. Русификация ресурса таблицы строк демонстрационного проекта MDIApp1

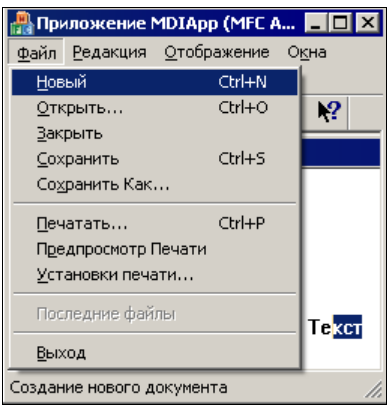


Рис. 12.22. Вид главного окна с раскрытым меню **File** демонстрационного проекта MDIApp1 после его русификации

Для закрепления материала русифицируйте демонстрационный проект SDIApp. Для этого создайте новую папку SDIApp1 и скопируйте в нее содержимое папки \Глава 12\SDIApp с прилагаемого компакт-диска. Русифицируйте демонстрационный проект SDIApp1 и сравните его с одноименным демон-

страционным проектом SDIApp1, имеющимся на прилагаемом компакт-диске, в папке \Глава 12\SDIApp.

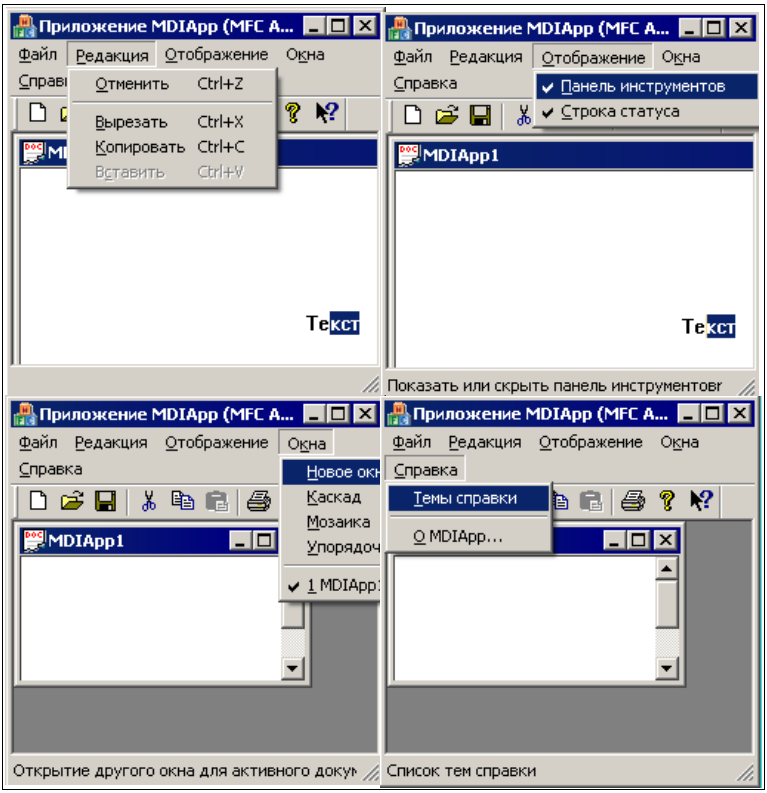


Рис. 12.23. Вид остальных меню демонстрационного проекта MDIApp1 после его русификации

12.3. Вопросы и упражнения для самопроверки

Для самопроверки ответьте на следующие вопросы:

1. Как запустить MFC Application Wizard?
2. Как получить справку по текущему окну настройки MFC Application Wizard?
3. Укажите назначение кнопок, имеющихся в текущем окне настройки MFC Application Wizard.
4. Укажите назначение и возможности исходного окна настройки MFC Application Wizard.

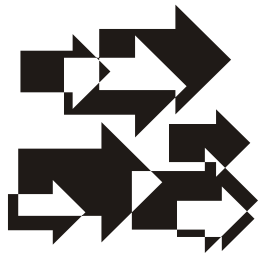
5. Укажите назначение и возможности окна настройки **Application Type** мастера MFC Application Wizard.
6. Укажите назначение и возможности окна настройки **Compound Document Support** мастера MFC Application Wizard.
7. Укажите назначение и возможности окна настройки **Document Template Strings** мастера MFC Application Wizard.
8. Укажите назначение и возможности окна настройки **Database Support** мастера MFC Application Wizard.
9. Укажите назначение и возможности окна настройки **User Interface Features** мастера MFC Application Wizard.
10. Укажите назначение и возможности окна настройки **Advanced Features** мастера MFC Application Wizard.
11. Укажите назначение и возможности окна настройки **Generated Classes** мастера MFC Application Wizard.
12. Как русифицировать ресурсы каркаса приложения, полученного с помощью MFC Application Wizard?
13. Как модифицировать и русифицировать строку заголовка главного окна каркаса приложения на базе MFC, полученного с помощью MFC Application Wizard?
14. Как русифицировать диалоговое окно **О программе...** каркаса приложения, полученного с помощью MFC Application Wizard?
15. Как русифицировать меню, команды меню и кнопки панели инструментов каркаса приложения, полученного с помощью MFC Application Wizard?
16. Как русифицировать ресурс таблицы строк каркаса приложения, полученного с помощью MFC Application Wizard?

Ответы на вопросы можно проверить — они приведены в *разд. П1.12 приложения 1*.

Для закрепления учебного материала, изложенного ранее в этой главе, выполните следующие упражнения:

1. Повторите создание каркаса MDI-приложения, используя MFC Application Wizard.
2. Используя MFC Application Wizard, создайте каркас SDI-приложения, обеспечивающего работу с единственным дочерним окном.
3. Русифицируйте копию демонстрационного проекта MDIApp, имеющегося на прилагаемом компакт-диске.
4. Русифицируйте копию демонстрационного проекта SDIApp, имеющегося на прилагаемом компакт-диске.

ГЛАВА 13



Модификация каркаса приложения на базе MFC, полученного с помощью мастера [7]

При модификации в качестве исходных проектов используем русифицированные демонстрационные проекты MDIApp1 и SDIApp1 *(см. главу 12)*. Далее, рассмотрим добавление нового меню с командами, кнопок на панель инструментов, создание и включение русифицированной справки для добавленных интерфейсных элементов.

13.1. Добавление нового меню и кнопок на панель инструментов. Демонстрационные проекты MDIApp2 и SDIApp2

Рассмотрим, как можно выполнить добавление в каркас русифицированного приложения, полученного с помощью мастера MFC Application Wizard, *нового меню*, на примере добавления меню **Дочернее окно** с двумя командами **Загрузка** и **Выгрузка**. С помощью этого меню можно будет загружать содержимое файла в активное дочернее окно редактирования или выгружать содержимое в файл на диске. Как и в предыдущей главе, в качестве исходных будем пользоваться копиями демонстрационных проектов MDIApp1 и SDIApp1, которые для удобства поместим соответственно в папки MDIApp2 и SDIApp2 и будем называть их MDIApp2 и SDIApp2.

Для добавления нового меню с командами в демонстрационном проекте MDIApp2 выберите вкладку ресурсов данного проекта **Resource View — MDIApp**, раскройте все ресурсы и двойным щелчком левой кнопки мыши по идентификатору `IDR_MDIAppTYPE` загрузите ресурс меню в окно редактирования. Добавьте новое меню **Дочернее окно** с командами **Загрузить**, **Выгрузить** и настройте их в соответствии с рис. 13.1—13.3. Сохраните выполненные изменения.

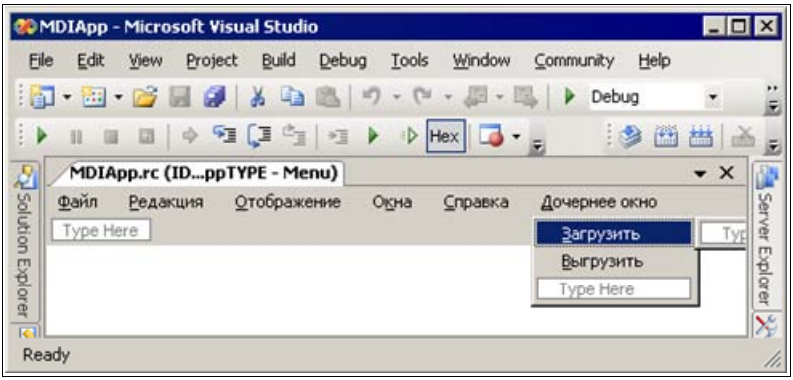


Рис. 13.1. Добавление меню **Дочернее окно** с двумя командами **Загрузить** и **Выгрузить** в демонстрационный проект MDIApp2

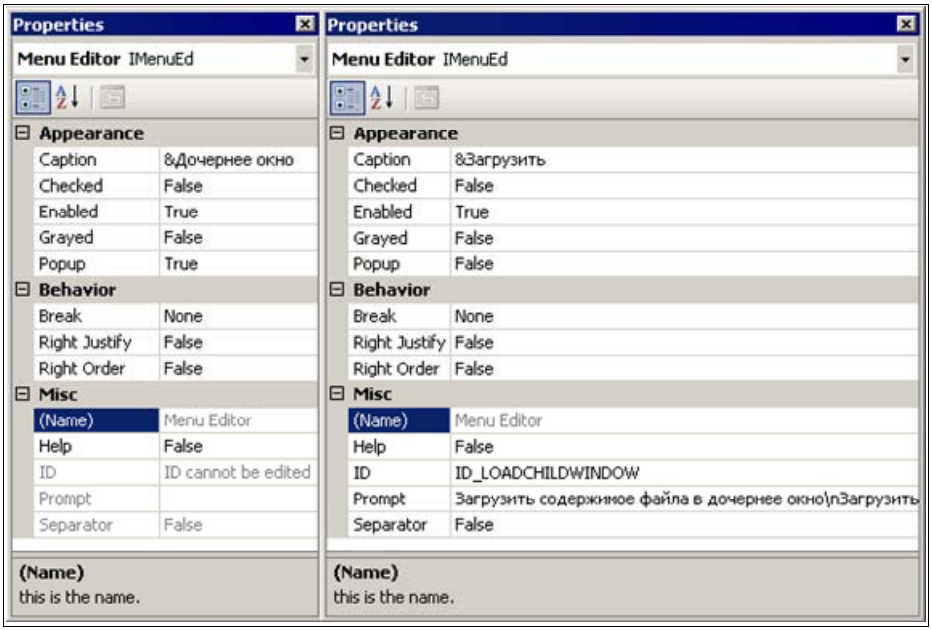


Рис. 13.2. Свойства меню **Дочернее окно** и команды **Загрузить** демонстрационного проекта MDIApp2

Для добавления в исходный код прототипа, макроса и определения *функции-обработчика* команды **Загрузить** достаточно выполнить команду **Add Event Handler...** контекстного меню, сконфигурировать программный код в соответствии с рис. 13.4 и нажать кнопку **Add and Edit**. Аналогичным образом добавьте код обработчика команды **Выгрузить** (рис. 13.5). Подробнее об

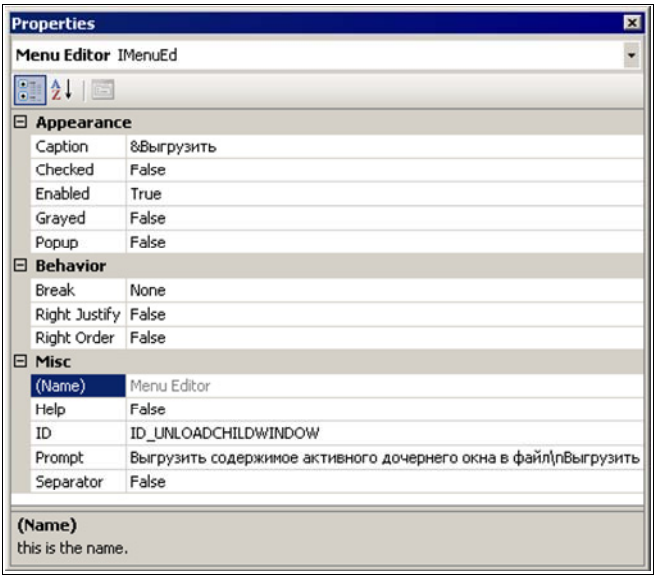


Рис. 13.3. Свойства команды **Выгрузить** демонстрационного проекта MDIApp2

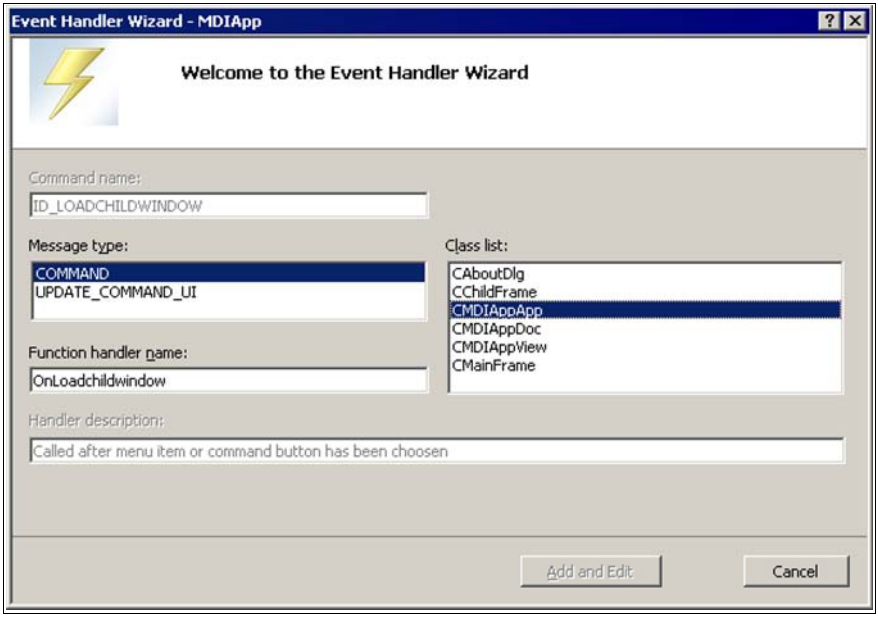


Рис. 13.4. Конфигурирование кода команды **Загрузить** демонстрационного проекта MDIApp2

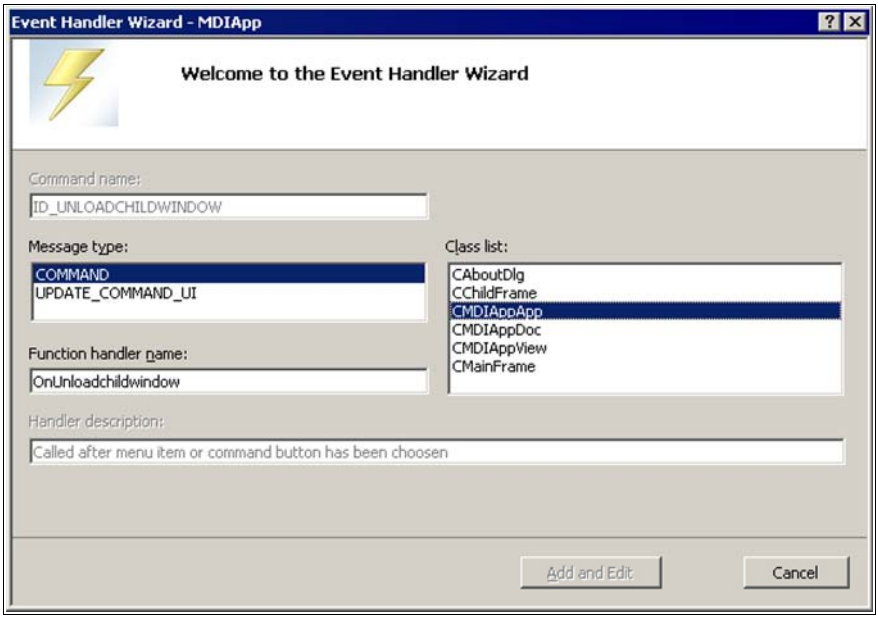


Рис. 13.5. Конфигурирование кода команды **Выгрузить** демонстрационного проекта MDIApp2

использовании мастера обработки событий **Add Event Handler...** для добавления в исходный код прототипа, макроса и определения функции-обработчика команды было указано ранее в *разд. 6.3* и рассмотрено в *разд. П3.2.6 приложения 3*.

Замечание

Обратите внимание на содержимое полей **Command name** на рис. 13.4—13.5. В этих полях указаны идентификаторы команд **Загрузить** и **Выгрузить**.

Модифицируйте измененные в результате этого файлы MDIApp.h и MDIApp.cpp в соответствии с листингами 13.1 и 13.2. В листинге 13.1 фрагмент текста, сгенерированного мастером Event Handler Wizard, содержится в конце листинга и имеет следующий вид:

```
public:
    afx_msg void OnLoadchildwindow();
public:
    afx_msg void OnUnloadchildwindow();
```

Аналогично, в листинге 13.2 фрагменты текста, сгенерированного мастером Event Handler Wizard, содержатся в таблице сообщений в начале и конце листинга ("заготовки" для обработчиков команд):

```
    ON_COMMAND(ID_LOADCHILDWINDOW, &CMDIAppApp::OnLoadchildwindow)
    ON_COMMAND(ID_UNLOADCHILDWINDOW,
               &CMDIAppApp::OnUnloadchildwindow)
...
void CMDIAppApp::OnLoadchildwindow()
{
    // TODO: Add your command handler code here
    return;
}
...
void CMDIAppApp::OnUnloadchildwindow()
{
    // TODO: Add your command handler code here
    return;
}
```

Листинг 13.1. Файл MDIApp.h

```
// MDIApp.h : main header file for the MDIApp application
//
#pragma once

#ifndef __AFXWIN_H__
    #error "include 'stdafx.h' before including this file for PCH"
#endif

#include "resource.h"       // main symbols

// CMDIAppApp:
// See MDIApp.cpp for the implementation of this class
//

class CMDIAppApp : public CWinApp
{
public:
    CMDIAppApp();

// Overrides
public:
    virtual BOOL InitInstance();

// Implementation
    afx_msg void OnAppAbout();
    DECLARE_MESSAGE_MAP()
```

```
// Прототипы обработчиков команд "Загрузить" и "Выгрузить" меню
//   "Дочернее окно"
public:
    afx_msg void OnLoadchildwindow();
public:
    afx_msg void OnUnloadchildwindow();
};

extern CMDIAppApp theApp;
```

Листинг 13.2. Фрагмент файла MDIApp.cpp

```
// MDIApp.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "MDIApp.h"
#include "MainFrm.h"

#include "ChildFrm.h"
#include "MDIAppDoc.h"
#include "MDIAppView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// CMDIAppApp
BEGIN_MESSAGE_MAP(CMDIAppApp, CWinApp)
    ON_COMMAND(ID_APP_ABOUT, &CMDIAppApp::OnAppAbout)
    // Standard file based document commands
    ON_COMMAND(ID_FILE_NEW, &CWinApp::OnFileNew)
    ON_COMMAND(ID_FILE_OPEN, &CWinApp::OnFileOpen)
    // Standard print setup command
    ON_COMMAND(ID_FILE_PRINT_SETUP, &CWinApp::OnFilePrintSetup)
    // Макросы обработчиков команд "Загрузить" и "Выгрузить"
    //   меню "Дочернее окно"
    ON_COMMAND(ID_LOADCHILDWINDOW, &CMDIAppApp::OnLoadchildwindow)
    ON_COMMAND(ID_UNLOADCHILDWINDOW,
               &CMDIAppApp::OnUnloadchildwindow)
END_MESSAGE_MAP()
```

```
// CMDIAppApp construction
CMDIAppApp::CMDIAppApp()
{
    EnableHtmlHelp();

    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}
// ...
// App command to run the dialog
void CMDIAppApp::OnAppAbout()
{
    CAboutDlg aboutDlg;
    aboutDlg.DoModal();
}

// CMDIAppApp message handlers

// Обработчик команды "Загрузить" меню "Дочернее окно"
void CMDIAppApp::OnLoadchildwindow()
{
    // TODO: Add your command handler code here
    // Загрузка содержимого файла "readme.txt" в дочернее окно
    m_pDocManager->OpenDocumentFile( "readme.txt" );

    return;
}

// Обработчик команды "Выгрузить" меню "Дочернее окно"
void CMDIAppApp::OnUnloadchildwindow()
{
    // TODO: Add your command handler code here
    // Сохранение всех изменений в активном дочернем окне
    m_pDocManager->SaveAllModified( );

    return;
}
```

Теперь добавим новые управляющие элементы на панель инструментов — две кнопки, действие которых будет аналогично действию команд **Загрузить** и **Выгрузить** меню **Дочернее окно**. Для добавления кнопок дважды щелкнем левой кнопкой мыши по ресурсу панели `IDR_MAINFRAME` и, тем самым, она будет загружена в окно редактирования.

Для отделения с помощью сепаратора новых кнопок, расположенных правее кнопки с пиктограммой в виде стрелки и вопросительного знака, достаточно ее перетащить посредством мыши вправо. Чтобы настроить параметры кнопки, следует щелкнуть по ней два раза левой кнопкой мыши, задать ее свойства в окне **Properties** и нарисовать пиктограмму кнопки в виде текста "Загр" (рис. 13.6).

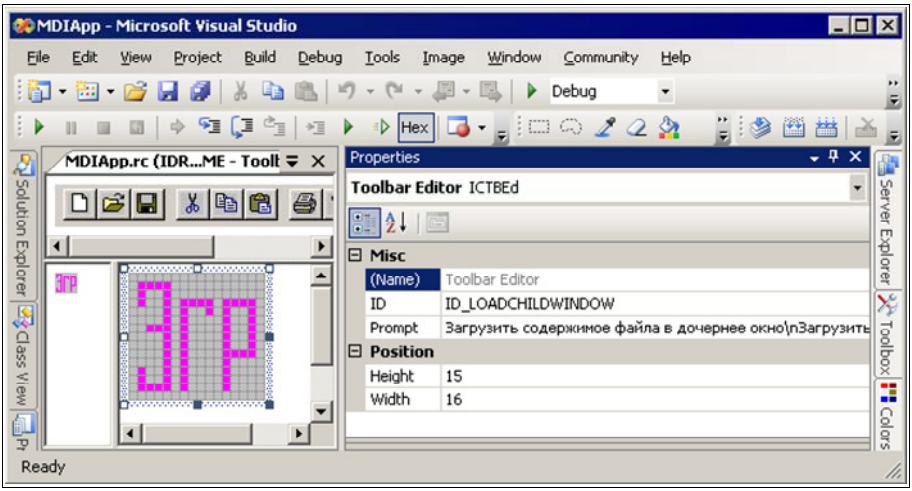


Рис. 13.6. Пиктограмма и свойства кнопки **Загрузить** демонстрационного проекта MDIApp2

Таким же образом создадим еще одну кнопку с настройками, показанными на рис. 13.7, и пиктограммой в виде текста "Вгр".

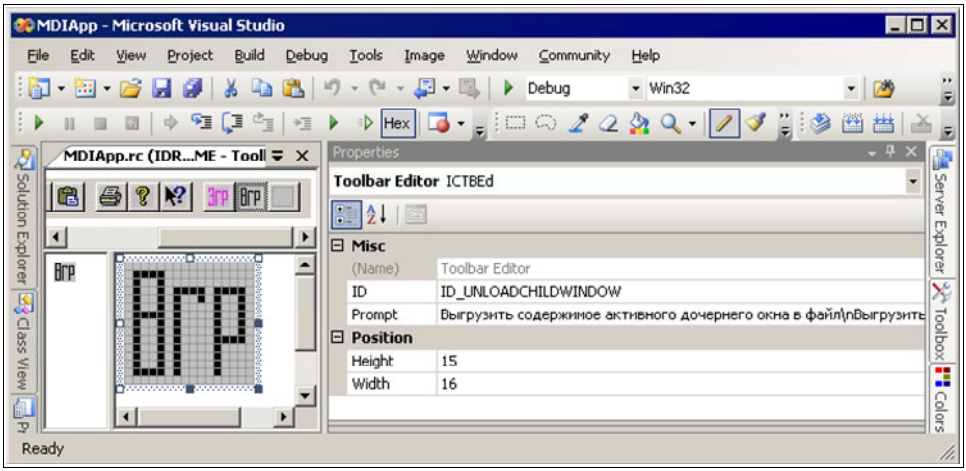


Рис. 13.7. Пиктограмма и свойства кнопки **Выгрузить** демонстрационного проекта MDIApp2

Замечание

Обратите внимание на то, что идентификаторы кнопок и соответствующих команд меню **Дочернее окно** совпадают. Это означает, что при активизации добавленных кнопок будут использованы те же обработчики, что и при активизации команд меню. Эти обработчики уже добавлены в демонстрационный проект.

Вид главного окна модифицированного таким образом демонстрационного проекта показан на рис. 13.8, а код проекта MDIApp2 имеется на прилагаемом компакт-диске, в папке \Глава 13\MDIApp2. Советуем вам самостоятельно повторить рассмотренную модификацию каркаса демонстрационного проекта — это очень важно для освоения материала. Поэкспериментируйте с добавленными меню, командами и кнопками. При всех затруднениях, которые могут у вас возникнуть, обращайтесь к демонстрационному примеру на компакт-диске.

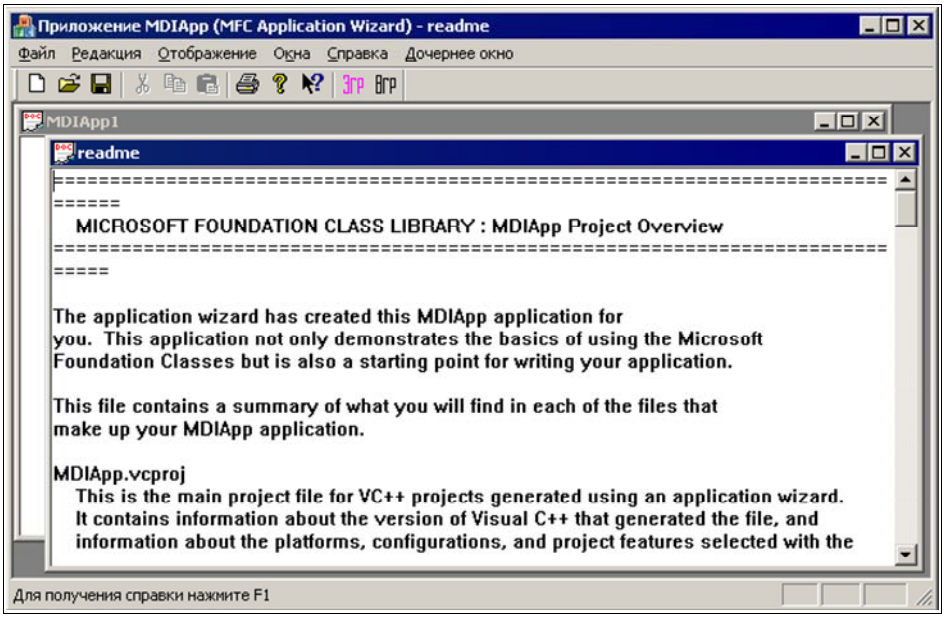


Рис. 13.8. Вид главного окна демонстрационного проекта MDIApp2 после добавления меню, команд и кнопок

Совет

Рекомендуем аналогичным образом добавить меню, команды и кнопки в демонстрационный проект SDIApp1. При всех затруднениях, которые могут возникнуть, обращайтесь к демонстрационному проекту SDIApp2, имеющемуся на компакт-диске, в папке \Глава 13\SDIApp2.

13.2. Создание и включение русифицированной справки для элементов интерфейса [7]

Практически в любом Windows-приложении вы встретите контекстно-зависимую справку (справочную систему) со сведениями о диалоговом окне, команде, кнопке панели инструментов или другом компоненте приложения. В ранних версиях IDE поддерживались две справочные системы — традиционная система WinHelp, основанная на файлах формата RTF (Rich Text Format) [7], и новая справочная система HTML Help, в основе которой лежат файлы в формате HTML (HyperText Markup Language). В IDE Microsoft Visual Studio 2005 поддерживается только более удобная и перспективная новая справочная система HTML Help, которая и будет рассмотрена далее.

Доступ к справочной системе приложения можно получить несколькими способами (все они поддерживаются в каркасе приложения, полученного с помощью MFC Application Wizard):

- чаще всего, как более удобный и оперативный, используется режим *интерактивной справки*. Режим "Что это такое?" включается при одновременном нажатии пользователем клавиш <Shift>+<F1> или кнопки на панели инструментов с пиктограммой со стрелкой и знаком вопроса. При этом указатель мыши принимает вид стрелки со знаком вопроса, для получения справки о команде, кнопке или области окна пользователь щелкает мышью на соответствующем элементе интерфейса. Режим интерактивной справки выключается после вывода справки на экран нажатием клавиши <Esc> или после переключения на другое приложение;
- с помощью меню **Help** (Справка), которое используется в большинстве оконных приложений;
- с помощью клавиши <F1> — если фокус ввода находится в активном окне приложения, диалоговом окне, на команде меню или кнопке панели инструментов, то при нажатии на клавишу <F1> вызывается раздел справочной системы, посвященный выбранному элементу интерфейса (например, команде меню).

Совет

Используя демонстрационный проект MDIApp2, попробуйте все перечисленные способы получения справочной информации.

Обычный документ, например обычная книга, имеет линейную структуру — один предмет изложения следует за другим в определенном порядке. В HTML-файле (HTML-документе) можно *мгновенно*, используя *гиперссыл-*

ки, перейти к любому его разделу, перемещаясь по документу в *произвольном* порядке.

Примечание

HTML (HyperText Markup Language, язык разметки гипертекста) — это стандартный язык разметки документов в сети Интернет. Язык HTML интерпретируется и отображается в виде документа, "удобного" для человека.

Далее, при создании и включении русифицированной справки для добавленных интерфейсных элементов в качестве исходного проекта используем рассмотренный демонстрационный проект MDIApp2, полученный с помощью MFC Application Wizard. Для этого создадим новую папку MDIApp3 и скопируем в нее содержимое папки MDIApp2 с прилагаемого компакт-диска. При этом, как и ранее, имя проекта в новой папке сохранилось прежним, чтобы отличить его от предыдущего проекта, будем называть новый проект демонстрационным проектом MDIApp3.

Для поддержки справочной системы добавим в каркас приложения демонстрационного проекта MDIApp2, полученного с помощью мастера, русифицированные темы справки для команд добавленного меню **Дочернее окно**. Такими командами являются команды **Загрузить** (идентификатор `ID_LOADCHILDWINDOW`) и **Выгрузить** (идентификатор `ID_UNLOADCHILDWINDOW`). Поскольку идентификаторы кнопок, добавленных на панель инструментов, совпадают с идентификаторами команд меню, то созданную справку можно использовать и для соответствующих кнопок.

Для каждой темы (Topic), которую нужно включить в справку, следует подготовить текстовое описание. *Текстовое описание* оформляется либо в виде HTML-файла, либо в виде набора таких файлов, объединенных гиперссылками. В последнем случае один из файлов будет *основным* (в нем, например, может быть оглавление), а остальные — *вспомогательными* (на них должны указывать гиперссылки из основного файла). В нашем примере имеет место последний случай. Один из файлов (назовем его ChildWindow.htm) будет соответствовать меню **Дочернее окно**, являться основным и содержать перечень команд меню — **Загрузить** и **Выгрузить**. Остальные файлы (назовем их соответственно Load.htm и Unload.htm) будут вспомогательными и на них будут указывать гиперссылки из файла ChildWindow.htm.

Для создания указанных HTML-файлов нужно выполнить следующее. В IDE перейдите в окно **Solution Explorer — MDIApp**, выберите в папке **HTML Help Files** (Файлы справки HTML) проекта файл MDIApp.hhp, вызовите контекстное меню, выполните в нем команду **Open With...** (Открыть с помощью...), в появившемся окне выберите команду **Microsoft® HTML Help Workshop (HTML Help)** и нажмите кнопку **OK**. В результате появится ин-

тегрированная среда разработки Microsoft Help Workshop, вид которой показан на рис. 13.9.

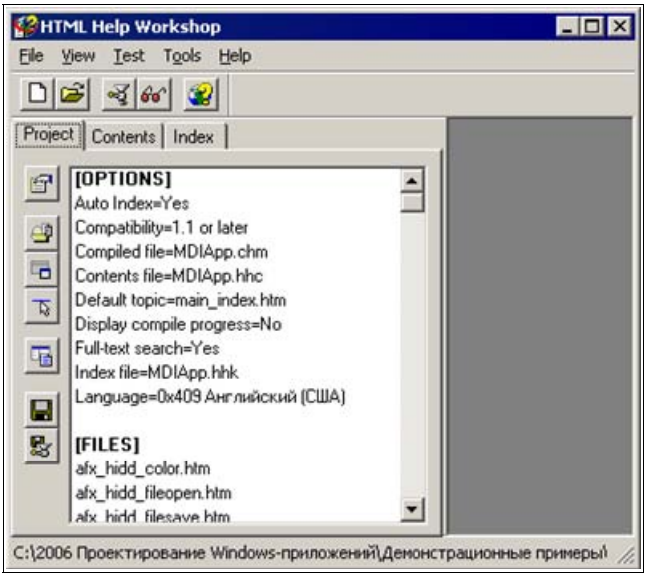


Рис. 13.9. Интегрированная среда Microsoft Help Workshop

Прежде всего, командой **Change project options** (Изменить параметры проекта) выполните русификацию проекта справки: в появившемся окне **Options** (Параметры), в поле **Language** (Язык) выберите **Русский** (рис. 13.10).

В среде проекта справки выполните команду **New** (Новый), в появившемся диалоговом окне **New** выберите **HTML File** (рис. 13.11) и нажмите кнопку **OK**.

В следующем диалоговом окне **HTML Title** (Заголовок HTML) (рис. 13.12) задайте заголовок HTML-файла (для файла ChildWindow.htm — ChildWindow) и нажмите кнопку **OK**. Заголовок будет использоваться при некоторых вариантах получения справочной информации. Для сохранения файла под нужным именем в среде проекта справки выполните команду **File | Save File As...**, укажите имя файла ChildWindow.htm и нажмите кнопку **Save** (Сохранить). В результате будет создан пустой файл. Таким же образом создайте пустые файлы Load.htm, Unload.htm с заголовками Load и Unload.

Заполнение созданных файлов требуемой информацией удобнее проводить в IDE. Так для заполнения файла ChildWindow.htm выполните команду **Open File** (Открыть файл), из папки **help** проекта выберите файл ChildWindow.htm и нажмите кнопку **Open** (Открыть). В окне редактирования появится пустой файл (рис. 13.13).

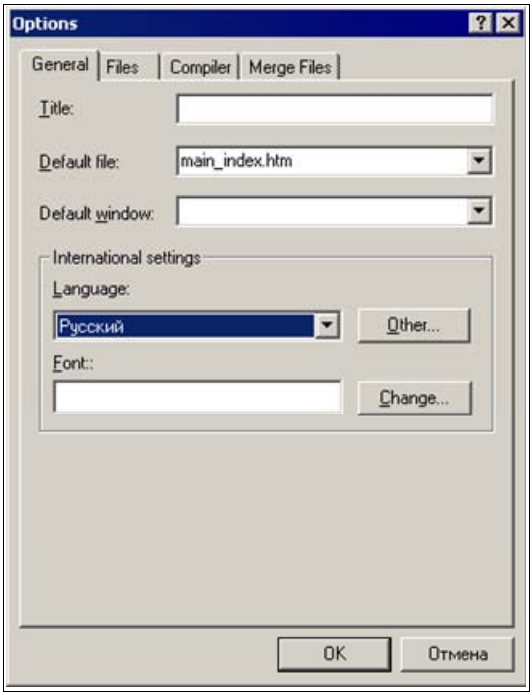


Рис. 13.10. Русификация проекта справки

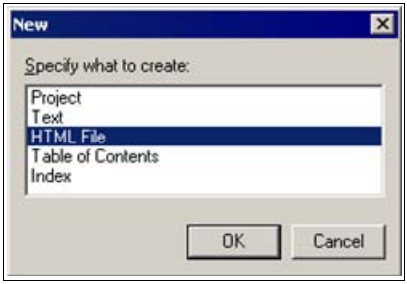


Рис. 13.11. Выбор типа создаваемого файла

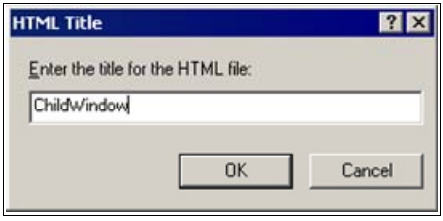


Рис. 13.12. Задание заголовка создаваемого файла

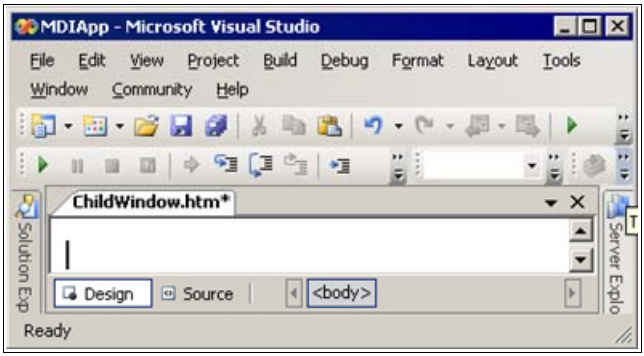


Рис. 13.13. Вид окна редактирования после открытия пустого файла ChildWindow.htm

Кнопка **Design** на рис. 13.13 предназначена для переключения в режим отображения текста файла справки в виде, наблюдаемом в справочной системе, а кнопка **Source** — для отображения текста файла на языке HTML. Для заполнения основного файла справки удобнее всего скопировать содержимое какого-либо существующего файла с гиперссылками (например, сгенерированного мастером файла menu_file.htm из папки HTML Help Topics проекта) и скорректировать его надлежащим образом. Вид этого файла после такой корректировки показан на рис. 13.14.

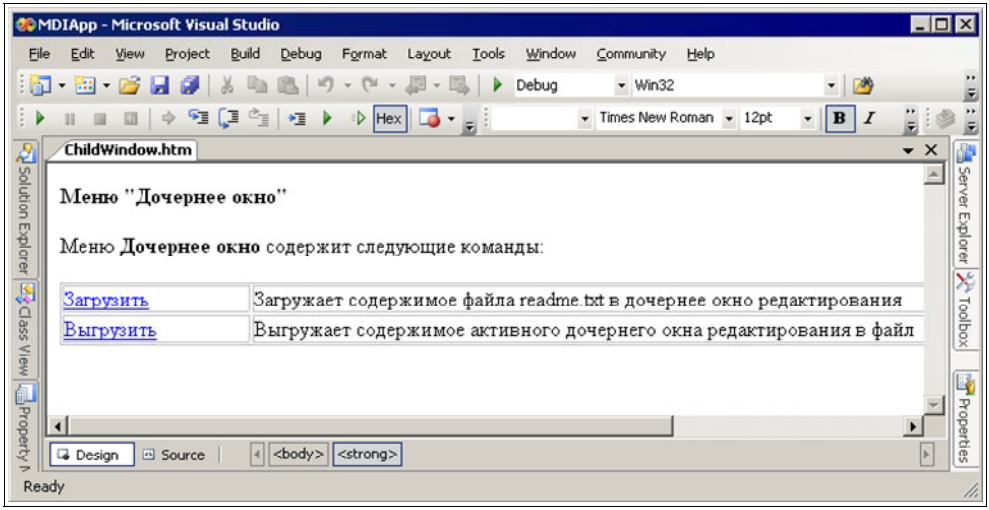


Рис. 13.14. Вид окна редактирования после заполнения файла ChildWindow.htm

Для *настройки гиперссылки*, выделенной синим цветом и подчеркиванием, следует вызвать ее контекстное меню и выполнить в нем команду **Properties**. В появившемся окне **Properties** в поле **HRef** указать имя файла, соответст-

вующего гиперссылке и нажать клавишу <Enter> (рис. 13.15). Для гиперссылки "Загрузить" таким файлом является Load.htm, а для гиперссылки "Выгрузить" — файл Unload.htm.

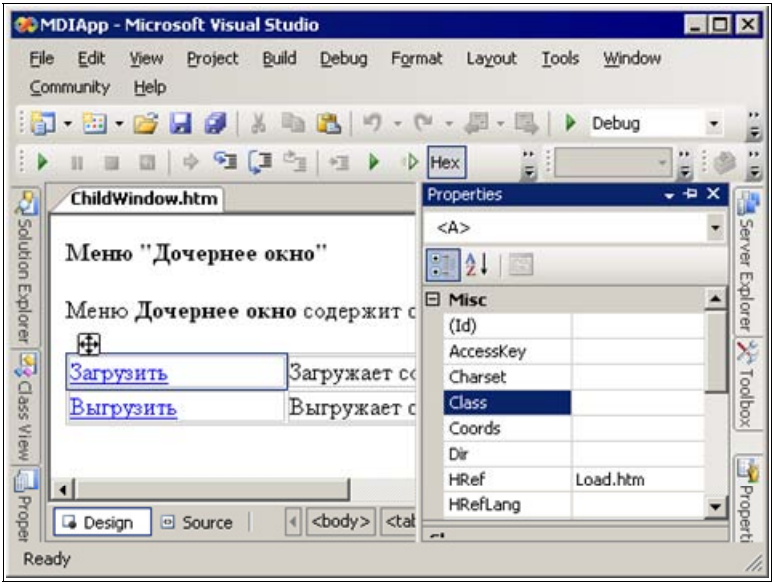


Рис. 13.15. Установление связи гиперссылки "Загрузить" и HTML-файла

Аналогичным образом заполните файлы Load.htm и Unload.htm. Эти файлы не содержат гиперссылок и их заполнять проще (рис. 13.16 и 13.17).

Для включения в файл справки графического изображения достаточно выбрать на рис. 13.16 вкладку **Source** (Исходный код) и вставить в текст файла следующие строки:

```
<p>
    Панель инструментов:
    <IMG alt="" src="file:///C:\Toolbar.bmp"></p>
```

В результате исходный код файла справки примет следующий вид:

```
<HEAD>
<meta name="GENERATOR" content="Microsoft&reg;
    HTML Help Workshop 4.1">
<Title>Load</Title>
</HEAD>
<BODY>
    <strong>Команда "Загрузить" (меню "Дочернее окно")</strong><p>
```

```
        Загружает содержимое файла readme.txt в дочернее окно
        редактирования с заголовкам readme</p>
    <p>
        Акселератор отсутствует</p>
    <p>
        Панель инструментов:
        <IMG alt="" src="file:///C:\Toolbar.bmp"></p>
    <p>
        Кнопка "Загрузить" на панели инструментов вторая
        справа.</p>
</BODY>
</HTML>
```

Примечание

Путь C:\Toolbar.bmp указывает на файл с графическим изображением.

С помощью **HTML Help Workshop** в раздел **FILES** (ФАЙЛЫ) добавьте все подготовленные HTML-файлы. С этой целью выберите заголовок сегмента **[FILES]**, выполните команду **Add/Remove topic files** (Добавить/удалить файлы тем) и в появившемся окне **Topic Files** (Файлы тем) нажмите кнопку **Add** (Добавить). В следующем диалоговом окне, удерживая нажатой клавишу <Ctrl>, выберите файлы ChildWindow.htm, Load.htm, Unload.htm и последовательно нажмите кнопки **Open** и **OK**. Нажатием на кнопку **Save Project, contens and index files** (Сохранить проект, содержимое и файлы) сохраните сделанные изменения.

В разделе **ALIASES** (ПСЕВДОНИМЫ) поставьте в соответствие идентификаторы элементов управления HTML-файлам, в которых эти элементы описаны. Это позволит получать *контекстную справку* по этим элементам. Идентификаторы элементов определены в файле hlp\HTMLDefines.h (команда **Загрузить** имеет идентификатор `HID_LOADCHILDWINDOW`, а команда **Выгрузить** — идентификатор `HID_UNLOADCHILDWINDOW`). Как правило, используются HTML-файлы из соответствующих тем. Чтобы выполнить это в среде **HTML Help Workshop**, выберите **[ALIAS]** ([ПСЕВДОНИМ]), активизируйте команду **HtmHelp API information**, в появившемся диалоговом окне выберите вкладку **Alias** (Псевдоним), нажмите кнопку **Add** и настройте окно **Alias** в соответствии с рис. 13.18. Аналогичным образом установите соответствие `HID_UNLOADCHILDWINDOW` и Unload.htm. В завершение нажмите кнопку **OK**, а для сохранения указанных изменений, как и ранее, выполните команду **Save Project, contens and index files**.

С помощью **HTML Help Workshop** на вкладке **Contents** добавьте созданную тему в оглавление. Для созданной темы задайте название и HTML-файл

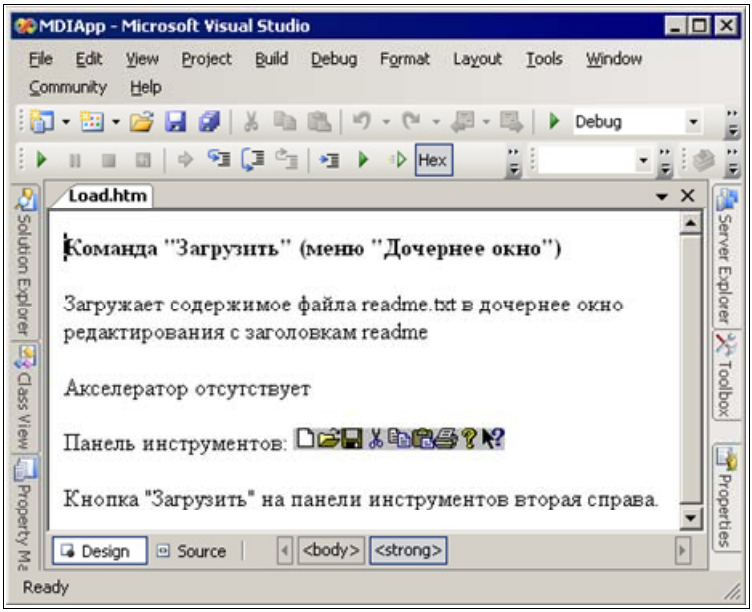


Рис. 13.16. Вид окна редактирования после заполнения файла Load.htm

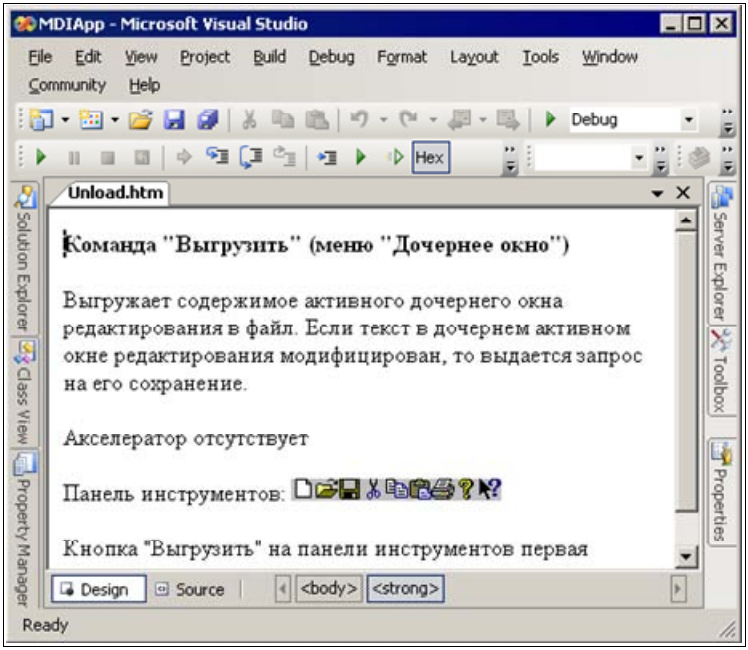


Рис. 13.17. Вид окна редактирования после заполнения файла Unload.htm

Рис. 13.18. Установление связи идентификатора команды и файла справки

(основной в том случае, если файлов несколько — наш случай). С этой целью в контекстном меню для **Menus** (Меню) выполните команду **Insert Topic...** (Добавить тему...), в появившемся окне **Table of Contens Entry** (Таблицы содержимого) в поле **Entry title** (Заголовок) введите "Дочернее окно", нажмите кнопку **Add**. Далее с помощью кнопки **Browse** (Пролистать) выберите файл ChildWindow.htm. Для завершения и сохранения последовательно нажмите кнопку **Open**, дважды нажмите кнопку **OK** и на вкладке **Project** выполните команду **Save Project, contens and index files**.

Для обеспечения всех возможностей получения справки добавьте *ключевые слова* на вкладке **Index**. При этом указывайте ключевое слово и заголовок HTML-файла, в котором идет речь о соответствующем понятии. Для этого, в среде **HTML Help Workshop** выберите вкладку **Index**, нажмите кнопку, снабженную всплывающей подсказкой **Insert a Keyword** (Добавить ключевое слово), а появившееся диалоговое окно **Index Entry** сконфигурируйте в соответствии с рис. 13.19. Для этого в поле **Keyword** (Ключевое слово) введите "Дочернее окно", нажмите кнопку **Add** и появится окно **Path or URL** (Путь или URL). В списке **HTML titles** (Заголовки HTML) выберите заголовок ChildWindow и дважды нажмите кнопку **OK**. Аналогичным образом установите еще две связи:

- Загрузить — Load — Load.htm;
- Выгрузить — Unload — Unload.htm.

Сохраните все сделанные изменения с помощью кнопки **Save Project, contens and index files** на вкладке **Project** и закройте среду **HTML Help Workshop**.

Выполните тестирование справочной системы полученного оконного приложения.

Код демонстрационного проекта MDIApp3 имеется на прилагаемом компакт-диске, в папке \Глава 13\MDIApp3. Самостоятельно повторите рассмотрен-

ную модификацию каркаса демонстрационного проекта — это очень важно для освоения материала. При всех затруднениях, которые могут у вас возникнуть, обращайтесь к демонстрационному примеру на компакт-диске.

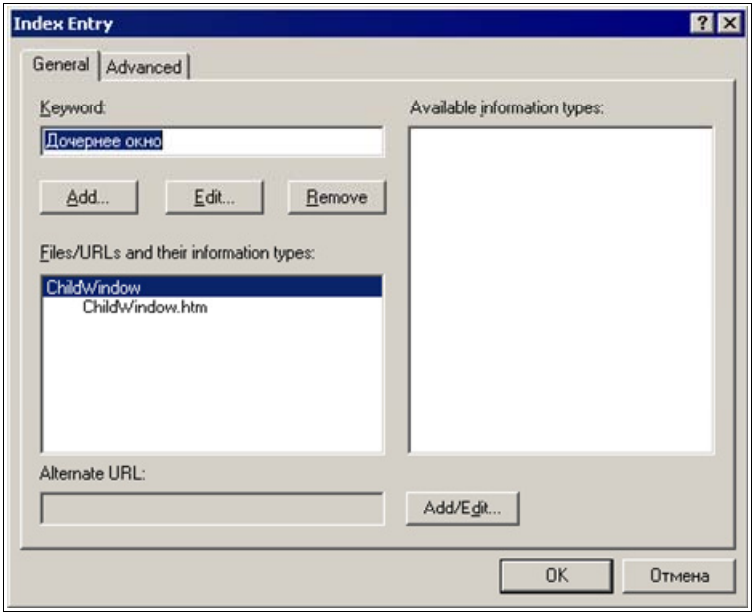


Рис. 13.19. Установление связи "Дочернее окно — ChildWindow — ChildWindow.htm"

Совет

Рекомендуем аналогичным образом добавить в справочную систему демонстрационного проекта SDIApp2, полученного с помощью мастера, справки для меню **Дочернее окно**, команд **Загрузить, Выгрузить** и одноименных кнопок на панели инструментов. При всех затруднениях, которые могут возникнуть, обращайтесь к демонстрационному проекту SDIApp3, имеющемуся на прилагаемом компакт-диске, в папке \Глава 13\SDIApp3.

13.3. Вопросы и упражнения для самопроверки

Для самопроверки ответьте на следующие вопросы:

1. Как добавить новое меню с командами в каркас приложения, полученный с помощью MFC Application Wizard?
2. Как добавить в исходный код прототип, макрос и определение функции-обработчика команды меню?

3. Как добавить кнопку на панель инструментов каркаса приложения, полученного с помощью MFC Application Wizard?
4. Какие разновидности справочных систем поддерживаются в IDE Microsoft Visual Studio?
5. Какими способами можно получить доступ к справочной системе приложения на базе MFC?
6. Что такое HTML-документ (HTML-файл)?
7. Что представляет собой текстовое описание темы справки?
8. Как запустить Microsoft Help Workshop?
9. Как выполнить русификацию проекта справки?
10. Как создать текстовое описание темы справки (поясните на примере демонстрационного проекта MDIApp3)?
11. Как включить в файл справки графическое изображение?
12. Как добавить в раздел **FILES** проекта справки подготовленные файлы справки?
13. Как обеспечить получение контекстной справки для добавленных управляющих элементов (поясните на примере демонстрационного проекта MDIApp3)?
14. Как добавить в оглавление созданную тему (поясните на примере демонстрационного проекта MDIApp3)?
15. Как добавить в справочную систему ключевые слова для новой темы (поясните на примере демонстрационного проекта MDIApp3)?

Ответы на вопросы можно проверить — они приведены в *разд. П1.13 приложения 1*.

Для закрепления учебного материала, изложенного в этой главе, выполните следующие упражнения:

1. Повторите создание демонстрационного проекта MDIApp2.
2. Повторите создание демонстрационного проекта SDIApp2.
3. Используя копию демонстрационного проекта MDIApp2, имеющегося на прилагаемом компакт-диске, проверьте получение справочной информации всеми способами.
4. Используя копию демонстрационного проекта SDIApp2, имеющегося на прилагаемом компакт-диске, проверьте получение справочной информации всеми способами.
5. Повторите создание демонстрационного проекта MDIApp3.
6. Повторите создание демонстрационного проекта SDIApp3.

ПРИЛОЖЕНИЕ 1

Ответы и решения к вопросам и упражнениям для самопроверки

Приводимые далее ответы и решения сгруппированы по главам. Для удобства читателя каждый приводимый ответ или решение предваряются соответствующим вопросом, который выделяется **полужирным шрифтом**.

П1.1. Глава 1

Что представляет собой окно?

См. *разд. 1.1*.

Перечислите и охарактеризуйте основные компоненты окна.

Основные компоненты окна показаны на рис. 1.1 и 1.2. Наряду с перечисленными, на указанных рисунках, компонентами окна имеются и другие, такие как панели инструментов с кнопками, строка статуса, управляющие элементы (всевозможные кнопки, элементы редактирования, различные списки, индикаторы прогресса и т. п.).

Что такое класс окна, зачем он нужен?

См. *разд. 1.3*.

Перечислите и охарактеризуйте графические объекты, используемые в окнах.

Основными графическими объектами окон являются: значки, указатели мыши, текстовые курсоры, окна сообщения, диалоговые окна, шрифты, точечные рисунки, перья, кисти и др. Более подробное описание имеется в *разд. 1.4.1—1.4.9*.

Что такое событийно-управляемая ОС?

ОС Windows является событийно-управляемой. Этот термин означает, что отдельные части ОС взаимодействуют между собой, а также с прикладными

программами посредством сообщений. Сообщение — это форма регистрации событий в операционной системе (или просто в системе). Система использует сообщения для того, чтобы сигнализировать о совершении событий.

Дайте общую характеристику формата сообщения, используемого в Windows.

Независимо от типа, все сообщения для 32-разрядных версий Windows характеризуются четырьмя общими параметрами (см. рис. 1.5):

- дескриптором окна, которому адресуется данное сообщение. Дескрипторы (или описатели) широко используются в приложениях Windows и представляют собой уникальный "номер" (беззнаковое 32-разрядное целое значение), который присваивается всем системным объектам (элементам управления, меню, значкам, перьям, кистям, областям памяти, устройствам вывода и т. д.);
- типом сообщения — второй параметр, задается идентификатором, который определен в заголовочном файле WINDOWS.H. Как правило, идентификаторы начинаются с двухсимвольного префикса, за которым следует символ подчеркивания. Так, оконные сообщения начинаются с префикса `WM_`: `WM_CREATE`, `WM_DESTROY`, `WM_SIZE`, `WM_PAINT`, `WM_QUIT`, `WM_COMMAND` и др. Сообщения кнопок имеют префикс `BM_`, полей — `EM_` и т. д.;
- еще двумя 32-разрядными параметрами, которые несут дополнительную информацию. Их содержание может изменяться в зависимости от типа сообщения. Например, посредством этих параметров может передаваться информация о том, какая клавиша была нажата, какая команда меню активизирована и т. д.

Опишите сообщение *WM_CREATE*.

Система посылает сообщение `WM_CREATE` (см. рис. 1.6) окну Windows *после* его создания, но *до* появления образа окна на экране. Свой первый параметр сообщение `WM_CREATE` не использует. Вторым параметром является указатель на структуру типа `CREATESTRUCT`, которая содержит информацию о создаваемом окне. Если приложение обрабатывает `WM_CREATE`, то при успешной работе оно должно возвращать значение 0. При этом создание окна продолжается. Если же приложение возвращает (–1), то функция создания окна (`CreateWindow` или `CreateWindowEx`) возвращает пустой манипулятор (со значением 0).

Опишите сообщение *WM_DESTROY*.

Система посылает сообщение `WM_DESTROY` (см. рис. 1.7) окну Windows перед его уничтожением *после* того, как образ окна удален с экрана. Сообщение

`WM_DESTROY` не использует оба параметра. Если приложение обрабатывает `WM_DESTROY`, то оно должно возвращать значение 0.

Опишите сообщение *WM_SIZE*.

Система посылает сообщение `WM_SIZE` (см. рис. 1.8) окну Windows после того, как размер окна был изменен. Значения новых размеров клиентской области окна можно получить с помощью макросов: `LOWORD(lParam)` — ширина, а `HIWORD(lParam)` — высота. Если приложение обрабатывает `WM_SIZE`, то оно должно возвращать значение 0.

Опишите сообщение *WM_QUIT*.

Сообщение `WM_QUIT` (см. рис. 1.9) означает запрос на завершение работы приложения. Оно приводит к тому, что специальная функция `GetMessage` возвращает значение 0. Первый параметр задает значение кода возврата, который приложение передает ОС при своем завершении.

Опишите сообщение *WM_PAINT*.

Сообщение `WM_PAINT` (см. рис. 1.10) посылается окну Windows в том случае, если система или приложение выдают запрос на перерисовку окна (или его части) на экране монитора. В частности, `WM_PAINT` посылается в результате вызова функций `UpdateWindow` и `RedrawWindow`. Сообщение `WM_PAINT` не использует оба параметра. Если приложение обрабатывает `WM_PAINT`, то оно должно возвращать значение 0.

Перечислите основные источники сообщений ОС Windows.

Существует четыре основных *источника*, от которых приложение может получить сообщение:

- пользователь;
- ОС Windows;
- само приложение;
- другие приложения.

Как в общем виде организован цикл обработки сообщений?

См. *разд. 1.5.4*.

Укажите назначение и особенности стандартного заголовочного файла Windows.h.

Включаемый файл WINDOWS.H имеет большой размер и открывает доступ к тысячам констант, структур, типов данных и прототипов функций. Он включается в большинство приложений Windows и содержит директивы подклю-

чения множества других заголовочных файлов. Это является основной причиной того, почему Windows-приложения компилируются так долго (в *главе 2* рассмотрен механизм прекомпиляции, устраняющий этот недостаток). В случае использования библиотеки MFC файл WINDOWS.H подключается к программе косвенно, через файл AFXWIN.H.

П1.2. Глава 2

С какой целью рассматриваются проекты Windows-приложений, использующих Windows API?

Для того, чтобы знать как устроены и уметь хорошо проектировать Windows-приложения с использованием библиотеки MFC, весьма полезно рассмотреть анатомию простого Windows-приложения, построенного на основе низкоуровневых средств программирования, таких как Windows API.

Укажите возможные способы создания копии проекта Windows-приложения FrameWnd, использующего Windows API.

Для создания копии проекта FrameWnd, использующего Windows API, можно использовать *три различных способа*, отличающихся трудоемкостью. Начало первых двух способов совпадает и заключается в создании пустого проекта. Для создания пустого проекта запустите IDE Microsoft Visual Studio 2005. В поле **Recent Projects** (Последние проекты) вкладки **Start Page** (Стартовая страница) (эту вкладку, если ее нет, можно сделать видимой, выполнив команду **View** (Вид) | **Other Windows** (Другие окна) | **Start Page**) выполните **Create:Project...** (Создать:Проект...). В результате на экране появится диалоговое окно **New Project** (Новый проект). Здесь вы должны выбрать тип создаваемого приложения — в поле **Project Types** (Тип проекта) выберите **Win32**, в поле **Templates** (Шаблоны) выберите **Win32 Project** (Проект Win32). Укажите путь для нового проекта — информация о расположении новой рабочей области проекта (путь) вводится в поле **Location** (Местоположение). Это можно сделать, набрав путь вручную, или воспользоваться расположенной справа кнопкой **Browse...** (Просмотр). Соответствующий подкаталог должен быть предварительно создан, если он отсутствует. Укажите в поле **Name** (Имя) имя проекта FrameWnd. Одновременно, с вводом имени проекта в поле **Name**, это имя автоматически добавляется в качестве подкаталога в поле **Location**. После выполнения указанных действий для создания проекта нажмите кнопку **OK**, в результате чего на экране появится диалоговое окно мастера создания приложения **Win32 Application Wizard — FrameWnd**. В этом окне выберите вкладку **Application Settings** (Настройки приложения), затем опцию **Empty Project** (Пустой проект) и нажмите кнопку **Finish** (Завершить). В окне **Solution Explorer — FrameWnd** для проекта FrameWnd

правой кнопкой мыши вызовите контекстное меню и выполните команду **Properties** (Свойства). В поле **Character Set** (Таблица символов) вкладки **General** (Общие) выберите **Use Multi-Byte Character Set** (Использовать многобайтовый набор) и нажмите кнопку **OK**. Чтобы наполнить созданный проект, необходимо добавить в него файлы, содержащие текст программы.

В рамках *первого способа*, в окне **Solution Explorer — FrameWnd** для проекта FrameWnd, правой кнопкой мыши, вызовите контекстное меню и выполните команду **Add** (Добавить) | **New Item...** (Новый элемент...). В появившемся окне **Add | New Item — FrameWnd** в поле **Categories** (Категория) выберите **Code** (Код), в поле **Templates** выберите **C++ File (.cpp)**, в поле **Name** задайте имя файла FrameWnd и нажмите кнопку **Add**. Аналогичным образом добавьте в проект пустые файлы StdAfx.cpp и StdAfx.h. В эти пустые файлы добавьте исходный код, в соответствии с листингами 2.1—2.7 (или скопируйте в них содержимое соответствующих файлов проекта FrameWnd, имеющегося на прилагаемом компакт-диске).

В рамках *второго способа*, после создания пустого проекта, скопируйте в папку проекта готовые файлы проекта FrameWnd, имеющегося на прилагаемом компакт-диске, и включите их в проект. Для включения файлов в проект в окне **Solution Explorer — FrameWnd** для проекта FrameWnd правой кнопкой мыши вызовите контекстное меню и выполните команду **Add | Existing Item...** (Существующий элемент). В появившемся окне **Add | Existing Item — FrameWnd** выберите файлы FrameWnd.cpp, StdAfx.cpp, StdAfx.h и нажмите кнопку **Add**. Чтобы обеспечить прекомпиляцию стандартных заголовочных файлов в копии проекта, полученной первым или вторым способом, выполните следующее. В окне **Solution Explorer — FrameWnd** для проекта FrameWnd правой кнопкой мыши вызовите контекстное меню и выполните команду **Properties**. В появившемся окне **FrameWnd Property Pages** откройте вкладку **C/C++**, выберите категорию **Precompiled Headers** (Прекомпилируемые заголовки) и в поле **Create/Use Precompiled Headers** (Создать/использовать прекомпилируемые заголовки) выберите **Use Precompiled Headers (/Yu)** (Использовать прекомпилируемые заголовки) и нажмите кнопку **OK**. Аналогично, в окне **Solution Explorer — FrameWnd** для файла StdAfx.cpp правой кнопкой мыши вызовите контекстное меню и выполните команду **Properties**. В появившемся окне **StdAfx.cpp Property Pages** откройте вкладку **C/C++**, выберите категорию **Precompiled Headers** и в поле **Create/Use Precompiled Headers** выберите **Create Precompiled Headers (/Yc)** (Создать прекомпилируемые заголовки) и нажмите кнопку **OK**.

В рамках *третьего способа* папка проекта просто копируется с компакт-диска. В полученной копии проекта выполните последовательную компиляцию файлов StdAfx.cpp и FrameWnd.cpp, командой **Build FrameWnd** по-

стройте проект и запустите его командой **Start Without Debugging** (Запуск без отладки) (см. рис. 2.1).

Замечание

Более подробно все указанные способы создания копии проекта Windows-приложения, использующего Windows API, описаны в *приложении 3*.

Опишите наиболее часто используемые типы данных ОС Windows.

Наиболее часто используемые типы данных представлены в табл. 2.1.

Перечислите наиболее часто используемые структуры данных ОС Windows и укажите их назначение.

Наиболее часто используемые структуры данных и их назначение приведены в табл. 2.2.

Перечислите и дайте краткую характеристику функциональных частей функции *WinMain* Windows-приложения, использующего Windows API.

Функция `WinMain` является обязательным компонентом любого низкоуровневого Windows-приложения. С нее начинается выполнение программы и ею же обычно заканчивается. Функция `WinMain` состоит из трех функциональных частей, в которых соответственно производятся:

- начальная инициализация приложения (подготовка данных класса окна и его регистрация);
- создание главного окна приложения;
- запуск цикла обработки сообщений, извлекаемых из очереди сообщений приложения (цикл обработки сообщений и приложение завершаются после получения сообщения `WM_QUIT`).

Опишите работу стандартного цикла обработки сообщений.

Следующий цикл работает в течение всего времени выполнения программы:

```
MSG                     msg;        // Для очередного сообщения
// Стандартный цикл обработки сообщений
while ( GetMessage( &msg, NULL, 0, 0 ) )
{
    TranslateMessage( &msg );
    DispatchMessage( &msg );
}
```

Каждая итерация представляет собой прием одного сообщения из очереди сообщений приложения. За это отвечает системная функция `GetMessage`, вто-

рой аргумент которой (NULL) говорит о том, что читаться должны сообщения, адресованные любому окну приложения. Третий и четвертый аргументы формируют фильтр сообщений, который ограничивает получаемые сообщения определенным диапазоном. В нашем случае эти аргументы нулевые и будут приниматься любые сообщения, адресованные данному приложению. Когда сообщение выбирается из очереди, оно помещается в специальную структуру типа MSG, на которую указывает первый аргумент и которая выглядит следующим образом:

```
typedef struct tagMSG
{
    // Дескриптор (логический номер) окна, чья оконная
    //   процедура получает сообщение
    HWND                hwnd;
    UINT                message;    // Номер сообщения
    WPARAM              wParam;     // Дополнительная информация
                                    //   о сообщении, зависящая
    LPARAM              lParam;     //   от типа сообщения
                                    //   (wParam, lParam)
    DWORD               time;       // Время возникновения
                                    //   сообщения
    // Позиция указателя мыши на момент возникновения сообщения
    POINT               pt;
} MSG;
```

После того как сообщение взято из очереди сообщений, оно передается в функцию TranslateMessage, которая вызывает драйвер клавиатуры Windows для преобразования виртуальных кодов клавиш в ASCII-значения, которые ставятся в очередь программных событий в виде сообщения WM_CHAR. Это позволяет программе отличить, например, "A" от "a" без анализа состояния клавиши регистра. Эта функция необходима только тем приложениям, которые обрабатывают ввод данных с клавиатуры. Важность функции TranslateMessage заключается в том, что она позволяет пользователям выбирать команды меню не только шелчками мыши, но и нажатием клавиш. Последняя функция цикла — DispatchMessage берет данные о сообщении из структуры типа MSG и передает их в соответствующую оконную процедуру для обработки. После того как сообщение передано, снова вызывается функция GetMessage, чтобы получить следующее сообщение из очереди (если таковое имеется). За исключением WM_QUIT, каждое сообщение заставляет GetMessage возвратить значение TRUE. Принимая сообщение WM_QUIT, программа выходит из цикла обработки сообщений и завершает работу. Важно понимать, что этот, или подобный ему, цикл без конца повторяется в течение всей жизни программы, написанной для Windows.

Опишите основные свойства оконного класса (поля структуры *WNDCLASS*).

Любое окно Windows принадлежит одному из существующих в данный момент в системе классов, который должен быть определен до того, как окно будет отображено на экране. Класс окна задает наиболее общие свойства окон, например, форму курсора при перемещении его в область окна, или имя меню, определенного для окон этого класса. Другими словами, класс окна — это шаблон, в котором определяются выбранные стили, шрифты, заголовки, пиктограммы, размер, расположение окна и т. д. Поскольку каждый класс имеет свою структуру параметров окна, доступную всем, не происходит ненужного дублирования данных. При этом следует учитывать, что имя класса должно быть уникальным, чтобы не возникало конфликтов с классами окон других приложений. Кроме того, два окна одного и того же класса используют общую функцию окна (оконную процедуру) со всеми ее вспомогательными функциями.

Объявление структуры `WNDCLASS` имеет вид:

```
typedef struct tagWNDCLASSA
{
    // Определяет свойства окна, которые могут комбинироваться
    //   при помощи операции | (ИЛИ). Если этому полю присвоить
    //   значение NULL, то ОС Windows автоматически установит
    //   значения по умолчанию
    UINT                style;
    WNDPROC             lpfnWndProc;// Адрес функции окна,
                                    //   которая обрабатывает сообщения
                                    //   для окон данного класса
    // Задает число байт, которое необходимо дополнительно
    //   запросить у ОС Windows под эту структуру для хранения
    //   собственных данных, присоединенных к классу. Это поле
    //   может иметь значение NULL
    int                 cbClsExtra;
    // Задает число байт, которое необходимо дополнительно
    //   запросить у ОС Windows для размещения всех структур,
    //   создаваемых совместно с данным классом для хранения
    //   собственных данных, присоединенных к окну. Это поле
    //   также может иметь значение NULL
    int                 cbWndExtra;
    // Кто создает определение класса. Это необходимо для
    //   служебных действий внутри ОС Windows. Когда
    //   завершается последний экземпляр программы, ОС Windows
    //   удаляет все связанные определения классов
    HINSTANCE           hInstance;
```

```
// Определяет пиктограмму, которая будет использоваться для
//   изображения приложения, например, на панели задач. Вы
//   можете создать пиктограмму сами или использовать одну
//   из предопределенных пиктограмм. Это поле может иметь
//   значение NULL
HICON               hIcon;
// Идентифицирует курсор мыши по умолчанию, используемый в
//   данном окне. Если нет желания создавать свой курсор,
//   то можно воспользоваться одним из предоставляемых
//   системой
HCURSOR             hCursor;
// Задает цвет фона для окна. Для его определения можно
//   воспользоваться любой из 20 системных констант цвета,
//   которые определены в заголовочном файле winuser.h.
// При использовании этих констант к их значению
//   обязательно нужно добавлять единицу, т. к. первое
//   значение для них равно нулю, а нуль является
//   недействительным значением для логического номера
//   кисти. Кроме того, необходимо выполнить приведение к
//   типу HBRUSH — иначе компилятор выдаст сообщение об
//   ошибке.
// Если значение этого поля равно нулю, то приложение
//   должно самостоятельно управлять заливкой фона окна при
//   поступлении соответствующего сообщения
HBRUSH              hbrBackground;
// Указатель на имя меню окна, определенное в файле
//   ресурсов. Это поле может иметь значение NULL
LPCSTR              lpszMenuName;
// Указатель на строку, содержащую имя класса (имя класса
//   должно быть уникальным)
LPCSTR              lpszClassName;
} WNDCLASS;
```

В чем заключается регистрация класса окна? Можно ли для одного и того же класса окна создавать несколько экземпляров окон?

Как уже было сказано, любое окно Windows принадлежит одному из существующих в данный момент в системе классов, который должен быть определен до того, как окно будет отображено на экране. Функция `RegisterClass` определяет класс окна, которое выводится на экран. Все окна Windows — объекты, каждый из которых имеет набор характерных черт. Заполняя структуру `WNDCLASS`, вы сообщаете Windows, какими хотите видеть конкретные объекты. Конечно же, для одного и того же класса окна можно создавать несколько экземпляров окон.

Что нужно сделать для создания конкретного окна? Что является параметрами конкретного окна?

Для создания конкретного окна нужно последовательно вызвать функции `CreateWindow` (выполняющую собственно создание окна), `ShowWindow` (выполняющую отображение окна на экране) и `UpdateWindow` (производящую перерисовку окна):

```
// Создание окна
hWnd = CreateWindow(
    // Указатель на строку зарегистрированного имени класса
    szClassName,
    // Указатель на строку заголовка окна
    szTitle,
    WS_OVERLAPPEDWINDOW,         // Стиль окна
    // Горизонтальная координата левого верхнего угла окна
    //   (используется умалчиваемое значение)
    CW_USEDEFAULT,
    // Вертикальная координата левого верхнего угла окна
    //   (используется умалчиваемое значение)
    CW_USEDEFAULT,
    CW_USEDEFAULT,               // Ширина окна (используется
                                 //   умалчиваемое значение)
    CW_USEDEFAULT,               // Высота окна (используется
                                 //   умалчиваемое значение)
    NULL,                        // Дескриптор родительского
                                 //   окна (его нет)
    NULL,                        // Дескриптор меню окна (его
                                 //   нет)
    hInstance,                   // Дескриптор экземпляра
                                 //   приложения
    NULL );                      // Указатель на дополнительные
                                 //   данные окна (их нет)
if ( !hWnd )
    return FALSE;
```

Аргументов в вызове `CreateWindow` много, но назначение каждого из них очевидно. Первый аргумент — имя зарегистрированного оконного класса. Второй — заголовок окна. Аргументы с четвертого по седьмой задают начальное положение и размеры окна. Константа `CW_USEDEFAULT` предписывает Windows самой выбрать эти параметры. Это значение выбирается наиболее часто, но если необходимы некоторые конкретные величины, то можно указать их здесь. Следующие три аргумента определяют дескрипторы родительского окна, меню и самого приложения соответственно. Последний (один-

надцатый) аргумент позволяет ассоциировать с окном некоторые дополнительные данные. Функция `CreateWindow` возвращает дескриптор созданного окна Windows. После возврата из функции `CreateWindow` Windows записывает в свою внутреннюю базу данных информацию, необходимую для сопровождения данного конкретного окна. Но при этом окно на экране не появляется — требуется вызов оставшихся двух функций, благодаря которым полностью оформленное окно появляется на экране.

```
// Показать окно с дескриптором hWnd: передает в ОС Windows
//   информацию (nCmdShow) о том, в каком виде необходимо
//   отобразить окно
if ( ShowWindow( hWnd, nCmdShow ) )
    return FALSE;
Обратите внимание, что второй аргумент, указанный в вызове функции
ShowWindow( ), можно переопределить, задавая различные варианты
отображения окна
// Перерисовать окно: для перерисовки окна функция предписывает
//   ОС Windows послать окну сообщение WM_PAINT
if ( !UpdateWindow( hWnd ) )
    return FALSE;
```

Перечислите и дайте краткую характеристику основных типов окон ОС Windows.

Основными типами окон являются:

- *перекрывающиеся окна* (overlapped windows). Это основной, наиболее универсальный тип окон Windows. Для их создания чаще всего используется комбинированный стиль `WS_OVERLAPPEDWINDOW`. Главное окно приложения, как правило, имеет именно этот тип;
- *всплывающие окна* (popup windows). Чаще всего этот тип окон создается с использованием стиля `WS_POPUP`. Обычно они используются для отображения какой-либо информации на короткий промежуток времени. Наиболее часто окна этого типа применяются для отображения диалоговых или окон сообщений. Основное отличие этих окон заключается в том, что даже если они имеют родительское окно, то все равно всегда отображаются поверх всех окон на экране, выскакивая, как поплавки, наверх даже тогда, когда пользователь делает активным другое окно. Для окон этого типа, как правило, организуется своя оконная процедура. Они могут и не иметь родителя. Если такое окно не имеет родительского, то оно является совершенно независимым от создавшего его окна и, по своим свойствам, практически не отличимо от перекрывающихся окон. Поведение всплывающего окна, имеющего родителя, зависит от того, что происходит с родительским окном. Когда главное окно минимизируется, всплывающее окно "без родителя" скрывается, а "с родителем" остается (на экране). Подчеркнем еще

один важный момент. Когда мы говорим, что вспомогательное окно имеет родителя, то это совсем не означает, что оно является дочерним;

- *дочерние окна* (child windows). Дочерние окна создаются тогда, когда у приложения уже есть главное окно. Такие окна связаны некоторыми характеристиками (как бы подчинены) с тем окном, из которого они были созданы — отсюда и такое название. Назначение таких окон может быть самым разнообразным, начиная от простого деления родительского окна на области до многодокументного интерфейса. Все элементы управления также являются дочерними окнами. Дочерние окна никогда не отображаются вне своего родительского окна ни в раскрытом виде, ни в виде пиктограммы — они как бы целиком принадлежат родителю. Располагаются они в родительском окне относительно верхнего левого угла его рабочей (клиентской) области. Более того, при перемещении родительского окна по экрану его дочерние окна перемещаются вместе с ним. И, наконец, дочернее окно никогда не может стать активным.

Что такое оконная процедура? Укажите, как она устроена и работает.

Чтобы программа могла взаимодействовать с "внешним миром", необходимо это каким-либо образом обеспечить. В любом Windows-приложении такое взаимодействие обеспечивает *оконная процедура*. Она регистрируется в системе одновременно с регистрацией оконного класса и вызывается всякий раз, когда Windows выполняет какую-либо операцию над окном приложения. При каждом вызове оконной процедуры ей передается для обработки очередное сообщение, поступающее из очереди сообщений приложения. Система Windows "знает", что имя этой процедуры указано в поле `lpfnWndProc` структуры `WNDCLASS`, описывающей класс окна. Все окна, созданные на базе этого класса, будут управляться указанной оконной процедурой. В Windows существует более двухсот различных оконных сообщений, которые могут посылаться окнам. Все они имеют префикс `WM_` (window message — оконное сообщение), например, `WM_CREATE`, `WM_SIZE`, `WM_PAINT` и др. Первым параметром оконной процедуры является дескриптор окна, которому высылается сообщение. Поскольку одна процедура может обрабатывать сообщения сразу нескольких окон, этот дескриптор позволяет определить, какому именно окну адресовано сообщение. Второй параметр содержит идентификатор (номер) сообщения. Два оставшихся параметра, типов `WPARAM` и `LPARAM`, несут дополнительную информацию, зависящую от конкретного сообщения. Обработка самих сообщений осуществляется с помощью стандартной конструкции `switch`, которая может в конкретном случае быть довольно большой по размеру (ее размер зависит от числа обрабатываемых пользователем сообщений). В соответствии с конструкцией переключателя в оконной процедуре часть сообщений в ней может обрабатываться явно, а все остальные сообще-

ния передаются в функцию `DefWindowProc`, управляющую поведением окна по умолчанию. Она умеет обрабатывать, по умолчанию, почти все виды сообщений, ассоциированные с конкретным типом окна. Например, она "знает" как обрабатывать действия мыши, изменяющие форму и местоположение окна.

Что такое контекст экрана? Перечислите и охарактеризуйте разновидности контекста экрана.

Одна из основных особенностей Windows API — независимость вывода от конкретной модели устройства. Программное обеспечение, которое поддерживает эту независимость, содержится в двух библиотеках динамической компоновки. Первая, GDI.DLL, обеспечивает графический интерфейс устройства (GDI, Graphic Device Interface), вторая — является драйвером устройства. Использование того или иного драйвера зависит, естественно, от конкретной разновидности устройства вывода. Перед операцией вывода на некоторое устройство приложение должно запросить GDI о загрузке соответствующего драйвера (обычно это происходит автоматически и не требует от программиста дополнительных усилий по программированию). После загрузки драйвера приложение настраивает все параметры вывода. Windows обеспечивает хранение всех необходимых параметров в специальной структуре, называемой *контекстом устройства*. Эта структура определяет набор графических объектов, связанных с ними атрибутов и графических режимов, которые собственно и воздействуют на вывод. Одним из контекстов устройств Windows API является *контекст экрана*. Используя контекст устройства, приложение может осуществлять следующий набор операций:

- перечисление графических объектов;
- выбор новых графических объектов;
- удаление графических объектов;
- сохранение графических объектов, их атрибутов и графических режимов;
- восстановление графических объектов, их атрибутов и графических режимов.

Windows API предоставляет для экрана три типа контекстов:

- *контекст класса* поддерживается только для совместимости с предыдущими версиями Windows;
- *частные контексты экрана* следует использовать в прикладных программах, которые выполняют многочисленные операции рисования. Это обеспечит повышение их быстродействия. Приложение создает частный контекст устройства, определяя стиль `CS_OWNDC` для класса окна в момент инициализации структуры `WNDCLASS` с последующей регистрацией класса окна с помощью функции `RegisterClass`;

- *общие контексты экрана* используются обычно в приложениях, которые лишь время от времени выполняют операции рисования. Приложение получает общий или частный контекст экрана, вызывая одну из функций: `BeginPaint`, `GetDC` или `GetDCEx`. Windows инициализирует общие контексты экрана значениями по умолчанию, которые можно менять по мере необходимости при помощи соответствующих функций Windows API.

Количество общих контекстов экрана ограничено и, следовательно, по завершении операций вывода необходимо освободить их, сообщив об этом системе вызовом функции `EndPaint` или `ReleaseDC`. В результате вызова одной из этих функций изменения параметров контекста, которые были произведены приложением, будут отменены. Таким образом, параметры отображения необходимо устанавливать каждый раз. Частные контексты устройства отличаются от общих тем, что сохраняют любые изменения для заданных по умолчанию данных даже после вызова функции `EndPaint` или `ReleaseDC`. Частные контексты устройства лишь однократно инициализируются значениями по умолчанию. Они автоматически удаляются после разрушения последнего окна класса.

Когда нужно работать с контекстом экрана? Как следует работать с контекстом экрана?

В каких случаях Windows-приложению необходимо осуществлять вывод? Конечно же, при изменении данных, чтобы пользователь мог это увидеть. Вывод должен осуществляться и в следующем ряде случаев:

- в момент создания окна;
- при изменении размеров или местоположения окна;
- при появлении окна или его части из-за другого окна;
- при минимизации или максимизации окна;
- при отображении данных из только что открытого файла;
- при прокрутке, изменении или выборе данных, находящихся в окне, и т. д.

Во многих случаях Windows берет на себя инициирование процесса отображения, отмечая область перерисовки и посылая соответствующему окну сообщение `WM_PAINT`. Это сообщение обрабатывает само приложение, осуществляя вывод данных в окно. В любом случае, Windows-приложению для начала вывода необходимо получить *дескриптор контекста* устройства вывода. Одним из контекстов устройств Windows API является *контекст экрана*. Для получения дескриптора контекста экрана чаще всего используется функция `BeginPaint`. Используя полученный дескриптор контекста экрана, приложение может осуществлять следующий набор операций:

- перечисление графических объектов;
- выбор новых графических объектов;
- удаление графических объектов;
- сохранение графических объектов, их атрибутов и графических режимов;
- восстановление графических объектов, их атрибутов и графических режимов.

По завершении требуемых операций с экраном контекст экрана следует освободить. Чаще всего это делается вызовом функции `EndPaint`.

Перечислите стандартные функции для рисования на экране графических примитивов.

Основными функциями для рисования на экране графических примитивов являются `TextOut` (вывод текста), `MoveToEx` и `LineTo` (рисование линии), `Ellipse` (рисование эллипса и окружности), `Rectangle` (рисование прямоугольника) и `Polygon` (рисование многоугольника).

Зачем нужна прекомпиляция? Как задать прекомпиляцию в проекте Windows-приложения, использующего Windows API?

См. *разд. 2.7*.

П1.3. Глава 3

Для чего нужна библиотека классов MFC, что она обеспечивает?

Создание даже простых Windows-приложений с использованием стандартных функций API является довольно сложной задачей, для решения которой требуется много времени и труда. Для *снижения* указанных *затрат* IDE Microsoft Visual Studio предлагает мощную библиотеку классов MFC (Microsoft Foundation Classes), предназначенную для разработки оконных приложений, отвечающих всем стандартам и требованиям сегодняшнего дня. Библиотека классов MFC предоставляет программистам удобные классы объектов, в которых инкапсулированы все наиболее важные структуры и функции API. Библиотека MFC значительно *проще* в использовании, чем функции API.

Какие основные преимущества обеспечивает использование библиотеки классов MFC?

Использование библиотеки классов MFC обеспечивает следующие *основные преимущества*:

- устраняются конфликты, связанные с совпадением имен стандартных и обычных функций и переменных;

- код и данные инкапсулируются в классах, доступ к ним можно ограничить;
- обеспечивается наследование возможностей базовых классов.

Каких основных принципов придерживались разработчики библиотеки классов MFC?

Создатели библиотеки классов MFC при разработке данного продукта придерживались строгих правил и стандартов, базирующихся на следующих *основных принципах*:

- возможность *комбинирования* вызовов стандартных функций с использованием методов классов. Это означает, что в одной программе могут применяться как библиотека классов MFC, так и функции API;
- *баланс между производительностью и эффективностью* базовых и производных классов. Создатели библиотеки классов MFC осознавали, что от совершенства ее программного кода зависит эффективность работы основанного на ее использовании приложения. Четко соблюдались требования к размеру классов и быстроте выполнения их методов. В результате по скорости работы классы библиотеки MFC незначительно уступают библиотечным функциям языка C/C++;
- преемственность в переходе от использования API-функций к библиотеке классов MFC. Одна из поставленных перед разработчиками MFC задач состояла в том, чтобы программисты, уже знакомые с функциями API, не заучивали новый, далеко не малый, набор имен функций и констант. Это требование строго соблюдалось при именовании членов классов. Это выгодно отличает библиотеку классов MFC от других библиотек;
- простота переноса библиотеки классов MFC на разные операционные платформы — от Windows 95 до Windows 2000 и Windows XP;
- органичная взаимосвязь с традиционными средствами программирования на языке C++, позволяющая избежать чрезмерного усложнения программного кода.

Еще одно *важное свойство*, на которое обращали внимание разработчики Microsoft, — это возможность непосредственного использования базовых классов в программных проектах Windows-приложений. Существовавшие до MFC библиотеки классов были чересчур абстрактными. В Microsoft их называли "тяжелыми классами", поскольку основанные на них приложения отличались большим размером программного кода и недостаточно быстро выполнялись. Разработчикам MFC удалось найти золотую середину между разумным уровнем абстрактности и размером кода.

Перечислите основные особенности библиотеки классов MFC.

Библиотека классов MFC обладает следующими *основными особенностями*.

- *Расширенная система обработки исключительных ситуаций*, благодаря которой приложения менее чувствительны к ошибкам и сбоям.
- *Улучшенная система диагностики*, позволяющая записывать в файл информацию об используемых объектах. Сюда также можно отнести возможность контроля за содержимым данных-членов классов.
- *Полная поддержка* всех функций API, элементов управления, сообщений, GDI, графических примитивов, меню и диалоговых окон.
- *Возможность определить тип объекта во время выполнения программы*. Это позволяет осуществлять динамическое управление данными-членами классов, в случае использования различных экземпляров класса.
- Отпала необходимость в использовании многочисленных громоздких структур `switch...case`, которые часто являются источниками ошибок в процедурном программировании. Все сообщения связываются с обработчиками внутри классов.
- Относительно небольшой размер и быстрота выполнения программного кода классов.
- Поддержка *модели компонентных объектов* COM (Component Object Model).
- *Использование тех же соглашений* об именовании методов классов, которые применялись при подборе имен функций Windows API. Это существенно облегчает идентификацию действий, выполняемых классами.

Как можно получить самые точные и полные сведения о составе и иерархии классов библиотеки MFC?

Самые полные сведения о составе и иерархии классов библиотеки MFC можно получить из справочной системы IDE Microsoft Visual Studio 2005. Для этого достаточно выполнить команду **Help | Index** и в поле **Look for:** появившегося окна **MFC Reference — Microsoft Visual Studio 2005 Documentation — Microsoft Document Explorer** набрать **MFC, hierarhy** и посмотреть вкладки **Hierarhy Chart** и **Hierarhy Chart Categories**.

Какие соглашения об именах приняты в библиотеке классов MFC?

В качестве *префикса*, обозначающего имя класса, библиотека классов MFC использует заглавную букву C (от слова *Class* — класс), за которой идет имя, характеризующее назначение класса. Например, `CWinApp` — класс, определяющий приложение, `CWnd` — базовый класс всех оконных объектов, `CDialog` — класс диалоговых окон и т. д.

Для *имен методов классов* используются три способа. При *первом способе* имя объединяет глагол и существительное, например, `LoadIcon` (загрузить пиктограмму) или `DrawText` (нарисовать текст). При *втором способе* имя метода состоит только из существительных, например, `DialogBox` (блок диалогового окна). Для методов, предназначенных для преобразования одного типа в другой, обычными являются такие имена, как `XtoY` (из X в Y). Для данных-членов классов библиотеки MFC принят следующий *способ назначения имен*: обязательный префикс `m_` (от *class member* — член класса), за которым идет префикс, характеризующий тип данного, и завершается все именем данного. Например, `m_pMainWnd`, где `p` — префикс, указывающий на то, что эта переменная является указателем.

Укажите особенности работы со стандартными включаемыми файлами библиотеки классов MFC.

В любом из файлов Windows-приложения на базе библиотеки классов MFC одной из первых является строка:

```
#include <stdafx.h>
```

Этот заголовочный (подключаемый, включаемый) файл служит для подключения необходимых библиотечных файлов к программе и включает в себя все наиболее часто используемые заголовочные файлы. Это и файл afxwin.h, который содержит описание основных классов библиотеки MFC и сводит воедино все включаемые файлы, необходимые для базового функционирования MFC (ядро MFC и стандартные компоненты). Это и файлы: afxcmn.h (общие управляющие элементы MFC), afxext.h (содержит описание классов общего назначения, макросы, базовые типы данных MFC, классы расширений MFC), afxmt.h (многопоточность MFC) и afxres.h (ресурсы MFC). При разработке больших приложений нам обязательно потребуется использование заранее откомпилированных заголовочных файлов (прекомпиляция), что позволит существенно сократить время компиляции. Файл stdafx.h как раз и выступает в такой роли. Здесь же отметим изменения, которые претерпел основной заголовочный файл Windows — файл windows.h. Как и раньше, каждое Windows-приложение должно содержать этот файл. Отличие же заключается в том, что, в своем последнем виде, он уже не представляет собой огромного "монстра", а разбит на 35 сравнительно небольших файлов, в том числе:

- windef.h — содержит определения базовых типов;
- winbase.h — описывает базовые функции API;
- wingdi.h и winuser.h — включают определения процедур для модулей GDI и USER соответственно, а также определения констант и макросов для этих модулей системы.

П1.4. Глава 4

С какой целью рассматривается один и тот же проект простого Windows-приложения с использованием двух технологий создания — на базе Windows API и библиотеки классов MFC?

Такое решение сделает возможным сопоставление двух *технологий создания оконных приложений* — на базе Windows API и с использованием библиотеки классов MFC, что позволит лучше понять анатомию высокоуровневых оконных приложений.

В чем заключаются особенности оформления исходных текстов приложения FrameWnd на базе MFC?

Особенностью оформления исходного кода проекта данного оконного приложения является включение в них, а не в текст учебного пособия в виде комментариев важной информации. Это делает возможным при работе с проектом получать ответы на возникающие вопросы более оперативно, не прибегая к чтению текста книги. Исходный код проекта демонстрирует также *профессиональное оформление*.

Укажите место класса *CWinApp* в иерархии классов библиотеки MFC и его назначение.

Место класса `CWinApp` в иерархии классов библиотеки MFC показано на рис. 4.1. `CWinApp` является базовым классом, с помощью которого мы создаем объект — Windows-приложение. Основными задачами объекта этого класса являются инициализация и создание главного окна, а затем опрос системных сообщений.

Что происходит при создании объекта с типом *CWinApp* или с типом производного от него класса?

В этом случае функция `WinMain` при создании объекта класса `CWinApp` или производного от него класса операционной системой не вызывается. Вместо этого происходит обращение к стартовой функции `_WinMainCRTStartup` библиотеки времени выполнения. В числе прочих операций, выполняемых этой функцией, имеется и вызов функции `WinMain` библиотеки MFC. Эта функция, как и обычная функция `WinMain`, выполняет инициализацию первого экземпляра приложения (метод `InitApplication`), инициализацию текущего экземпляра приложения (виртуальный метод `InitInstance` класса `CWinApp` или производного от него класса), запуск цикла обработки сообщений (виртуальный метод `Run` класса `CWinApp`) и корректное завершение работы приложения (виртуальный метод `Run` класса `CWinApp`).

Какую роль в классе *CWinApp* играет виртуальный метод *InitInstance* и с какой целью его следует переопределять?

В классе `CWinApp` важную роль играет переопределяемый (виртуальный) метод `InitInstance`. В своем первозданном виде этот метод не делает "ничего" и наполнение его содержанием целиком является задачей программиста. Обычно это то место, где конструируется объект главного окна приложения. Это единственный метод класса `CWinApp`, который вы должны обязательно переопределить для того, чтобы Windows-приложение что-нибудь делало. С этой целью обычно используется класс, производный от класса `CWinApp`, в котором и переопределяется метод `InitInstance`.

Укажите назначение методов *Create*, *MoveWindow*, *ShowWindow* и *UpdateWindow*, наследованных из класса *CWnd*.

Метод `Create` обеспечивает создание окна. Местоположение и размеры окна задаются с помощью метода `MoveWindow`. Для отображения и перерисовки окна используются соответственно методы `ShowWindow` и `UpdateWindow`.

Можно ли обойтись только методом *Create*, не используя методов *MoveWindow*, *ShowWindow* и *UpdateWindow*?

Да, это возможно при надлежащем выборе значений аргументов при вызове метода `Create`.

Как сделать созданный объект окна главным?

Указатель на объект созданного окна нужно присвоить переменной `MainThread::m_pMainWnd`. Это позволит корректно завершить приложение, когда будет закрыто его главное окно.

Укажите место класса `CFrameWnd` в иерархии классов библиотеки MFC и его назначение.

Место класса `CFrameWmd` в иерархии классов библиотеки MFC показано на рис. 4.2. Класс `CWnd` — это базовый класс для всех окон, созданных на базе библиотеки MFC. Помимо предоставления большого числа методов для работы с окнами для производных от него классов, он служит для создания разнообразных дочерних окон. В отличие от своего базового, класс `CFrameWnd` служит для создания перекрывающихся или всплывающих окон и спроектирован так, чтобы взять на себя выполнение всех основных функций Windows-приложения.

Для чего предназначен класс, производный от класса *CFrameWnd*?

Этот класс предназначен для обработки поступающих для него сообщений.

Что такое таблица сообщений? Укажите ее назначение.

Библиотека классов MFC предоставляет альтернативу конструкциям с использованием оператора `switch`, используемых в оконных процедурах традиционных низкоуровневых Windows-приложений для обработки сообщений, посылаемых окну. Взаимосвязь сообщений и их обработчиков может быть определена таким образом, что, когда сообщение поступает в оконную процедуру, автоматически вызывается соответствующий обработчик. Реализация такой взаимосвязи основана на понятии *карты* (или *таблицы*) *сообщений* (message map). Таблица сообщений представляет собой механизм пересылки сообщений и команд Windows в окна, представления и другие объекты приложения, реализованного на базе библиотеки классов MFC. Такие таблицы преобразуют сообщения системы Windows, извещения элементов управления, а также команды меню, кнопок панелей инструментов, акселераторов клавиатуры в вызовы функций соответствующих классов, которые их обрабатывают. Эта отличительная черта таблицы сообщений реализована по аналогии с виртуальными функциями языка C++, но имеет дополнительные преимущества, недоступные для них. Каждый класс, который может получить сообщение, должен иметь свою таблицу сообщений, чтобы иметь возможность соответствующим образом обрабатывать сообщения. При этом следует иметь в виду, что таблица сообщений должна определяться вне какой-либо функции или объявления класса. Она также не может размещаться внутри блока.

Укажите структуру таблицы сообщений.

Для определения таблицы сообщений используются три макроса:

- в файле реализации класса макросы `BEGIN_MESSAGE_MAP` и `END_MESSAGE_MAP`;
- в файле объявления класса макрос `DECLARE_MESSAGE_MAP`.

Макрос `DECLARE_MESSAGE_MAP` располагается в конце объявления класса, использующего карту сообщений. Непосредственно перед ним располагаются прототипы обрабатывающих функций. В прототипах обрабатывающих функций используется макрос `afx_msg`, который разработчики библиотеки классов MFC предусмотрели для будущего применения. Однако до сих пор он реально не используется. Макрос `afx_msg` в определении не раскрывается и употребляется только как метка-заполнитель, чтобы идентифицировать обработчики сообщений в объявлении класса. Два остальных макроса, определяющих карту сообщений, располагаются в реализации класса на внешнем уровне:

```
// Определение таблицы сообщений
BEGIN_MESSAGE_MAP( FrameWnd, CFrameWnd )
    ON_WM_PAINT( )
END_MESSAGE_MAP( )
```

Структура карты сообщений достаточно проста и представляет собой набор макросов (в данном примере набор содержит один макрос), заключенных в специальные "операторные скобки-макросы". Начинается карта сообщений с вызова макроса BEGIN_MESSAGE_MAP(FrameWnd, CFrameWnd), который имеет следующие аргументы: FrameWnd — задает имя класса, который является владельцем карты сообщений; CFrameWnd — определяет имя базового класса. Заканчивается карта сообщений вызовом макроса END_MESSAGE_MAP. Между этими двумя вызовами располагаются специальные макросы, называемые компонентами карты сообщений. Вот они-то, собственно, и позволяют сопоставить сообщение с конкретным обработчиком. Для каждого стандартного сообщения Windows определен свой макрос в форме ON_WM_XXX, где XXX — имя сообщения, например, ON_WM_PAINT. Имена обработчиков определяются при распаковке параметров каждого сообщения системы Windows на основе простого соглашения. Имена обработчиков всегда начинаются с префикса On, за которым следует имя соответствующего сообщения Windows (без префикса WM_), записанное строчными буквами (при этом первая буква — прописная). Например, для сообщения WM_PAINT в классе CWnd определен следующий обработчик:

```
afx_msg void OnPaint( void );
```

Описания всех обработчиков стандартных сообщений Windows можно найти в файле afxwin.h в объявлении класса CWnd. Отметим также, что для командных сообщений системы Windows от меню, акселераторов и кнопок панелей инструментов, для команд обновления, для извещений, посылаемых дочерними окнами своим родителям, для кнопок, элементов редактирования, списков и комбинированных списков используются макросы другого формата. Определение обработчика сообщения WM_PAINT помещается в файл реализации класса и имеет следующую структуру:

```
// Обработчик сообщения WM_PAINT
afx_msg void FrameWnd :: OnPaint( void )
{
    // Создаем объект для рисования и вывода
    CPaintDC            PaintDC( this );

    // Вывод заданного текста (третий аргумент) в определенное
    //   место окна (первый и второй аргументы)
    PaintDC.TextOut( 0, 0, "Текст с цветом умолчания" );

    return afx_msg void( );
}
```

Охарактеризуйте объект класса *CPaintDC*.

Рисовать удобнее всего с помощью класса `CPaintDC` библиотеки классов MFC. Объект этого класса представляет поверхность окна. Основными этапами использования такого объекта являются: создание объекта типа `CPaintDC`, рисование на объекте и разрушение объекта. Конструктор `CPaintDC` вызывает функцию `BeginPaint` и возвращает графический контекст для рисования, а деструктор вызывает `EndPaint`. Пример реализации метода `OnPaint`, обеспечивающего вывод в окно, приведен в ответе на предыдущий вопрос. Обратите внимание, что методы класса `CPaintDC` стали иметь более удобный интерфейс. Так, в методе `TextOut` количество параметров уменьшилось на два — стал ненужным дескриптор контекста устройства (он стал членом класса `CPaintDC`), а длина выводимого текста автоматически определяется по третьему аргументу в вызове метода `TextOut`.

Перечислите основные методы класса *CPaintDC* для рисования на экране графических примитивов.

Основными функциями для рисования на экране графических примитивов являются: `TextOut` (вывод текста), `MoveTo` и `LineTo` (рисование линии), `Ellipse` (рисование эллипса/окружности), `Rectangle` (рисование прямоугольника/квадрата) и `Polygon` (рисование многоугольника).

Укажите общую схему рисования на экране с использованием класса *CPaintDC*.

Общая схема рисования:

- создание новых инструментов с требуемыми свойствами (перо, кисть и т. д.);
- включение созданных инструментов в контекст с запоминанием информации об аналогичных инструментах, используемых в контексте по умолчанию;
- рисование новыми инструментами;
- по окончании рисования разрушение созданных инструментов;
- включение в контекст старых инструментов, используемых по умолчанию.

Какие недостатки имеются в использовании методов *Create*, *TextOut*, *CreateSolidBrush*, *Ellipse*, *DeleteObject*, *CreatePen* и *Rectangle*, в демонстрационных примерах главы 4? Как их можно исправить?

При использовании этих методов не контролируются и не обрабатываются результаты их завершения. В случае ошибки они возвращают нулевое значение. Следовательно, ошибочное завершение перечисленных методов можно выполнить с помощью макроса `ASSERT_VALID` (см. листинг 4.1).

П1.5. Глава 5

Следует ли определять в низкоуровневых приложениях свойства и регистрировать класс, используемый при создании кнопок?

Нет, не следует. В этом случае используется предопределенный класс "BUTTON".

Укажите назначение и смысл параметров API-функции MessageBox.

API-функция MessageBox предназначена для выдачи сообщений на экран. Ее интерфейс проиллюстрирован на рис. 5.3. Более подробные сведения можно получить с помощью справочной системы.

Укажите, какие элементарные стили можно использовать при создании кнопок.

В качестве элементарных стилей при создании кнопок можно комбинировать стили обычных кнопок с префиксом WS_ и элементарные стили для кнопок с префиксом BS_.

Как по сообщению *WM_COMMAND* определить его источник?

Сообщение WM_COMMAND генерируется очень большим количеством управляющих элементов (всевозможные кнопки, команды меню и т. п.). Для того чтобы можно было правильно обработать конкретное сообщение WM_COMMAND, нужно знать, какой же из управляющих элементов создал сообщение. Информация об этом (идентификатор управляющего элемента) хранится в младшем слове параметра wParam и получить ее можно с помощью макроса с параметром LOWORD(wParam).

Укажите назначение макроса *TRACE0*, используемого в приложениях, базирующихся на MFC.

Содержимое строки, указанной в круглых скобках макроса TRACE0, выводится в окно **Debug** IDE. В него же выводится код завершения приложения. Чтобы это происходило, нужно запускать приложение в режиме отладки (нажав клавишу <F5>).

Какой метод используется в приложениях на базе MFC для создания кнопок (дочерних управляющих элементов)?

Это перегружаемый виртуальный метод virtual void CreateChildControls (void).

Чем отличаются методы *MessageBox* и *AfxMessageBox*, используемые в приложениях на базе MFC?

Перечисленные методы отличаются от соответствующей стандартной функции API более удобным интерфейсом — не требуется дескриптор и используются значения параметров по умолчанию.

Укажите назначение вкладки *Object Browser* IDE.

Вкладка **Object Browser** IDE (ее можно сделать видимой командой **View | Object Browser**) является удобным средством работы с исходным кодом программного проекта, позволяющим наглядно представить структуру проекта в целом, пользовательских классов проекта и их состав. Вид вкладки для программного проекта WndBtn_MFC после выбора глобального объекта `MainThread` показан на рис. 5.7 (обратите внимание на настройку вкладки). Из рисунка видно, что проект использует два пользовательских класса: класс `MainThread`, производный от класса `CWinApp`, класс `FrameWnd`, производный от класса `CFrameWnd`, и один глобальный объект `MainThread` типа `MainThread`. Класс `MainThread` содержит единственный метод `InitInstance` со спецификатором доступа `public` (см. рис. 5.8). Класс `FrameWnd` содержит переменные `m_pBtnAbout` и `m_pBtnExit` — члены класса со спецификатором доступа `private`, методы `CreateChildControls`, конструктор и деструктор, со спецификатором доступа `public`, методы `OnBtnAboutClick`, `OnBtnExitClick` и `OnPaint`, со спецификатором доступа `protected` (см. рис. 5.9). Обратите внимание, что в правом окне вкладки **Object Browser** перечисляются сначала методы, а потом переменные класса. И те, и другие следуют в определенном порядке — сначала общедоступные, потом защищенные и в конце закрытые. Посмотрите и запомните, какими пиктограммами снабжаются члены класса в списке. Если во вкладке **Object Browser** "щелкнуть" мышью по названию класса, то в активном окне редактирования появится файл с объявлением соответствующего класса, причем текущей строкой будет строка заголовка класса. Аналогично, используя эту вкладку, можно в активное окно редактирования поместить файл с текущей строкой, соответствующей началу определения члена класса, который был выбран во вкладке **Object Browser**.

Укажите порядок создания кнопки.

Для создания кнопок используется перегружаемый виртуальный метод `virtual void CreateChildControls( void )`. В этом методе каждая из кнопок сначала размещается в динамической памяти, а затем создается с помощью метода `Create`. Первый аргумент в вызове этого метода задает название кнопки, второй — стиль кнопки, третий — местоположение и размеры кнопки, а последний — ее идентификатор. Обратите внимание, что стиль кнопки комбинирует стили окна и стили кнопок. Использование в стиле кнопки элементарного стиля `WS_VISIBLE` делает кнопки видимыми сразу после их создания. Для получения об этом методе более подробной справки можно поступить обычным для IDE способом — поместить курсор на название метода `Create` и нажать клавишу <F1>.

Как в приложениях на базе MFC задаются функции, обрабатывающие нажатия кнопок?

Функции, обрабатывающие нажатие кнопок, задаются в проекте с помощью таблицы сообщений.

Приложение, созданное с помощью функций API, имеет несколько окон. Сколько в нем нужно определить и зарегистрировать классов и сколько использовать оконных процедур?

Количество классов, которые нужно определить и зарегистрировать, и количество оконных процедур в приложении, созданном с помощью функций API, должны совпадать с числом используемых в нем окон.

Приложение на базе MFC имеет несколько окон с различной обработкой сообщений. Сколько в нем нужно использовать классов с таблицами сообщений, производных от класса *CFrameWnd* библиотеки MFC?

Количество таких классов должно соответствовать числу окон приложения на базе MFC.

Укажите порядок создания дочернего окна приложения на базе MFC.

При создании дочернего окна приложения на базе MFC используется стандартная последовательность действий, аналогичная используемой при создании главного окна приложения. Сначала в динамической памяти размещается объект типа `FrameWnd`, затем с помощью метода `Create` создается окно и посредством метода `ShowWindow` выводится на экран.

П1.6. Глава 6

Что такое ресурсы? Перечислите ресурсы, доступные в IDE.

Ресурсы являются важной составной частью Windows-приложений и очень удобным инструментом программиста. С ресурсами удобно работать, используя встроенный в IDE *редактор ресурсов*. Стандартные ресурсы IDE содержат ускорители (Accelerator), растровые изображения (Bitmap), курсоры (Cursor), диалоговые окна (Dialog), пиктограммы (Icon), меню (Menu), таблицы строк (String Table), панели инструментов (Toolbar), версию (Version) и др. Ресурсы представляют собой *кластеры бинарных данных*, "пристегнутые", во время компоновки, к концу выполняемого файла приложения. Когда Windows запускает приложение, она загружает в память только то, что ей необходимо в данный момент. Ресурсы обычно находятся на жестком диске в конце исполняемого файла до тех пор, пока Windows не затребует их. Большинство ресурсов — это выгружаемые данные только для чтения, которые

Windows может загрузить в память или выгрузить оттуда в любой момент, если изменятся требования к системной памяти.

Охарактеризуйте меню, команды, акселераторы и таблицы строк.

Взаимодействие пользователя с приложением может осуществляться в форме выбора той или иной команды *меню* или с помощью клавиатурных ускорителей (акселераторов). *Ресурсы ускорителей* позволяют пользователю вызывать те или иные команды более оперативно, с помощью сочетаний клавиш. *Таблицы строк* хранят строковые ресурсы отдельно от основного исходного кода на языке C++. Их легко модифицировать и переводить на другие языки, что обеспечивает легкий способ создания многоязыковых приложений, имеющих один и тот же программный код.

Как русифицировать добавляемые ресурсы?

Для *русификации добавляемых ресурсов* во вкладке **Solution Explorer** вызовите для проекта правой кнопкой мыши контекстное меню, выполните в нем команду **Properties**, выберите вкладку **Resourses** и в поле **Culture** задайте язык **Русский (0x419)**.

Как добавить в проект необходимые ресурсы?

Для добавления в проект необходимых ресурсов во вкладке **Solution Explorer** вызовите для проекта правой кнопкой мыши контекстное меню и выполните в нем команду **Add | Resource...**. В появившемся окне **Add Resource** (см. рис. 6.1) последовательно выберите нужные ресурсы, каждый раз нажимая кнопку **New**.

Как настроить ресурсы?

Во вкладке **Resource View** последовательно выберите настраиваемые ресурсы и в появляющейся каждый раз вкладке **Properties** задайте требуемые свойства ресурсов.

Как сделать видимыми вкладки *Resource View и Properties*? Как вернуться к умалчиваемому расположению окон IDE?

Вкладки **Resource View** и **Properties** можно сделать видимыми соответственно командами **View | Resource View** и **View | Other Window | Properties Window**. Стандартное (умалчиваемое) расположение окон IDE можно задать командой **Window | Reset Window Layout**.

Как задать состав и настроить меню, команды, акселераторы и "горячие" клавиши?

Это можно сделать с помощью редактора ресурсов. Подробнее см. в *разд. 6.2.1*.

Укажите типовые свойства меню.

Типовые свойства меню демонстрирует рис. 6.7. Для меню, в свойстве **Caption**, символ &, предшествующей букве, делает эту букву в названии меню активной (она будет подчеркнута). Свойство **Enabled** должно иметь значение True, иначе меню будет выключено. Свойство **Popup** обычно имеет значение True. Это означает, что при активизации меню появится всплывающее окно, содержащее команды меню. Свойство **Help** обычно имеет значение False. Это означает, что меню будут располагаться так, как это показано на рис. 6.7. В противном случае соответствующее меню и все меню справа от него будут прижаты к правой границе.

Как посмотреть или модифицировать свойства команды меню? Укажите типовые свойства команды меню.

Чтобы посмотреть или изменить настройку какой-либо команды меню, достаточно в окне редактирования выбрать эту команду и в появившемся окне дважды щелкнуть левой кнопкой мыши по команде. На вкладке **Properties** можно посмотреть или задать свойства команды. Свойства **Caption**, **Popup** и **Help** обсуждались при рассмотрении меню. Свойство **Enabled** не влияет на поведение команды меню. Свойство **ID** задает идентификатор команды, который будет использован при программной обработке действия команды. Свойство **Separator** обычно имеет значение False, в противном случае соответствующая команда будет отображаться в виде разделительной линии.

Как можно создать и настроить новое меню или команду?

Для *создания нового меню* достаточно дважды щелкнуть по пустому меню, расположенному рядом с последним заданным меню. В появившемся окне **Properties** можно задать параметры меню. Аналогично, для *создания новой команды меню* достаточно дважды щелкнуть по пустому полю команды меню, расположенному внизу. В появившемся окне настройки можно задать параметры команды меню.

Как расположить существующие меню или их команды в другом порядке?

Меню или команды меню можно расположить в любом нужном порядке путем их "перетаскивания" с помощью левой кнопки мыши.

Как посмотреть или модифицировать свойства акселераторов?

С этой целью, на вкладке **Resource View**, выберем нужный нам ресурс акселераторов (щелкнув по его идентификатору). В результате этого ресурс загрузится в окно редактирования и посредством встроенного редактора ресурсов его можно модифицировать. С помощью левой кнопки мыши можно вы-

брать любой из акселераторов — он будет подсвечен. Чтобы посмотреть или изменить *настройку* какого-либо акселератора, достаточно его выбрать и дважды щелкнуть по нему левой кнопкой мыши, появится вкладка **Properties**. С ее помощью можно посмотреть или задать свойства акселератора. Обратите внимание на то, какой идентификатор использует данный акселератор — это идентификатор, который применяется для соответствующей команды меню. Это означает, что при вводе клавиатурной комбинации акселератора будет использована та же обработка, что и при выборе команды меню. Обратите также внимание на то, что все доступные стандартные и, созданные к этому времени, нестандартные идентификаторы можно посмотреть и выбрать для использования в окне **ID** с помощью кнопки, расположенной в его правой части. Задать свойства акселератора можно и непосредственно в окне редактирования — в полях строки соответствующего акселератора.

Охарактеризуйте типы акселераторов. Укажите, как создать новый акселератор.

Акселераторы могут быть типа **VIRTKEY** или **ASCII**. В первом случае модификатор может отсутствовать или использоваться единственный модификатор <Alt>. Во втором случае можно использовать любые комбинации модификаторов — пустой, <Alt>, <Ctrl>, <Shift>, <Ctrl>+<Alt>, <Ctrl>+<Shift>, <Alt>+<Shift> и <Ctrl>+<Alt>+<Shift>. Для создания нового акселератора достаточно в окне редактирования дважды щелкнуть по нижнему (пустому) акселератору. В появившемся окне **Properties** можно задать параметры акселератора.

Какие файлы проекта, использующего ресурсы, поддерживаются редактором ресурсов и непосредственно не редактируются?

Два файла проекта — Resource.rc и resource.h — поддерживаются редактором ресурсов и непосредственно не редактируются. Они обновляются автоматически при каждой модификации ресурсов проекта, выполняемой с помощью редактора ресурсов. Файлы Resource.rc и resource.h, хотя и наблюдаемы в IDE как текстовые файлы, но их не рекомендуется редактировать.

Как выполняется программная поддержка ресурсов акселераторов?

Прежде всего, для программной поддержки ресурсов акселераторов модифицирован стандартный цикл обработки сообщений в функции `WinMain`:

```
// Загрузка акселераторов
HACCEL              hAccelTable;
hAccelTable = LoadAccelerators( hInstance,
    reinterpret_cast< LPCTSTR >( IDR_MAINFRAME ) );
MSG                 msg;          // Для очередного сообщения
```

```
// Стандартный цикл обработки сообщений
while ( GetMessage( &msg, NULL, 0, 0 ) )
{
    // TranslateAccelerator( ) нужна только при использовании
    //   акселераторов
    if ( !TranslateAccelerator( msg.hwnd, hAccelTable, &msg ) )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
}
```

Непосредственно перед стандартным циклом обработки сообщений вызывается стандартная API функция `LoadAccelerators`, которая загружает таблицу акселераторов, указанную вторым аргументом в ее вызове. Обратите внимание, что в качестве идентификатора ресурса акселераторов используется идентификатор, который применяется и редактором ресурсов. Далее, в теле стандартного цикла обработки сообщений вызывается `TranslateAccelerator`, которая формирует сообщение для оконной процедуры.

Как в программе подключить ресурс меню?

Для подключения ресурса меню в функции регистрации класса окна `InitApplication( )` изменена инициализация одного из полей структуры со сведениями о регистрируемом классе:

```
wc.lpszMenuName = MAKEINTRESOURCE( IDR_MAINFRAME );
```

Здесь макрос с параметром `MAKEINTRESOURCE( IDR_MAINFRAME )` используется для корректного приведения типа. Обратите также внимание на то, что в качестве идентификатора ресурса меню используется `IDR_MAINFRAME`, который применялся и редактором ресурсов.

Как в программе обрабатываются команды меню (акселераторы)?

В оконной процедуре `WndProc` выполняется обработка команд меню и акселераторов (см. варианты `ID_FILE_EXIT`, `ID_PENSTYLE_SOLID`, `ID_PENSTYLE_DASH` и `ID_PENSTYLE_DOT` в листинге 6.1). Обратите внимание на то, что в перечисленных вариантах использованы те же идентификаторы, что и в ресурсах команд меню и акселераторов. При обработке команд меню и акселераторов использована стандартная API-функция `InvalidateRect`, которая обеспечивает перерисовку заданной области окна. Нулевое значение второго аргумента в вызове этой функции задает перерисовку всей клиентской области окна, а нулевое значение последнего аргумента сохраняет прежний фон окна.

Перечислите различные способы активизации команд меню и дайте их краткое описание.

Команду меню можно активизировать с помощью мыши, акселератора (одновременное нажатие комбинации клавиш) и с помощью "горячих" клавиш. При настройке меню и команд в их названиях мы использовали символ &, что делало одну из букв в названии особенной (мы назвали такие символы "горячими", они подчеркнуты). Активизировать нужное меню или его команду можно путем соответствующего *последовательного* нажатия "горячих" клавиш. Например, для активизации команды **Точечный Alt+T** меню **Стиль карандаша** достаточно нажать клавиши в следующей последовательности (см. рис. 6.3): <Alt> — активизируются все "горячие" клавиши, <C> и <T>.

Как в приложение на базе MFC добавить с помощью мастера обработки событий таблицу сообщений, прототипы и определения функций, обрабатывающих команды меню и акселераторы?

Для добавления в соответствующий класс проекта таблицы сообщений, прототипов и определений функций, обрабатывающих команды меню, при модификации файлов с определением класса пользуйтесь мастером обработки событий. Для запуска мастера откройте вкладку **Resource View**, раскройте ресурс меню и загрузите в окно редактирования файл ресурса Resource.rc. Для соответствующей команды вызовите контекстное меню и выполните команду **Add Event Handler...**. Добавьте в класс таблицу сообщений и обработчик команды (например, см. рис. 6.12) и нажмите кнопку **Add and Edit**.

Как подключить ресурсы к приложению на базе MFC?

Подробный ответ на этот вопрос содержится в *разд. 6.3*.

Как в приложении на базе MFC получить размеры клиентской области окна?

Получение размеров клиентской области окна обеспечивает метод `GetClientRect` класса `CWnd` библиотеки MFC.

Какие недостатки имеются в использовании методов классов *CPen* и *CPaintDC* в приложении на базе MFC из главы 6?

Не контролируется правильность работы методов `CreatePen` и `LineTo`.

П1.7. Глава 7

Что такое панель инструментов?

Панель инструментов — это панель управления, содержащая ряд кнопок, которые являются альтернативой командам меню. На ней могут находиться

кнопки, имеющие различные стили. Панели инструментов обычно выравниваются по верхнему краю окна-рамки. Однако они могут "плавать" — как отдельные окна, имеющие свои размеры, но остающимися при этом на переднем плане, так и пристыковывающиеся к любому краю окна-рамки. Многочисленные примеры таких панелей можно видеть в IDE. С помощью редактора ресурсов можно легко создать панели инструментов с всплывающими подсказками и подключить их к приложению тоже не сложно.

Что такое строка статуса?

Строка статуса — это полоса (строка), имеющая ряд окон-индикаторов, в которых отображается некоторый текст. Этот текст может содержать пояснение к выбранному пункту меню, выбранной кнопке или данные о текущем состоянии приложения. Строка статуса обычно расположена внизу главного окна приложения.

Что такое всплывающие подсказки?

Всплывающие подсказки — это маленькие всплывающие окна, отображающие единственную строку текста (обычно короткую), которая описывает назначение элемента интерфейса пользователя. Всплывающие подсказки остаются невидимыми до тех пор, пока пользователь не поместит курсор мыши над каким-либо элементом, например, над кнопкой панели инструментов. И тогда они появляются там, где находится курсор. На рис. 7.1 показан пример всплывающей подсказки.

Как создать панель инструментов?

Для создания панели инструментов проще всего выполнить команду **Project | Add Resource...**, в появившемся окне **Add Resource** (см. приведенный ранее рис. 6.1) выбрать тип ресурса **Toolbar** и нажать кнопку **New**. В результате этого в проект включается ресурс панели инструментов.

Как настроить панель инструментов?

Для *настройки панели инструментов* необходимо во вкладке **Resource View** открыть ресурс **ToolBar**, для идентификатора ресурса правой кнопкой мыши вызвать контекстное меню и выполнить команду **Properties.** В появившемся окне **Properties** задайте параметры панели (см. пример на рис. 7.4). Обратите внимание на то, что идентификатор ресурса панели инструментов должен совпадать с идентификаторами других ресурсов программного проекта.

Как включить кнопку в созданную панель инструментов и настроить ее?

Для включения кнопки в созданную панель инструментов достаточно на вкладке **Resource View** дважды щелкнуть левой кнопкой мыши по идентифи-

катору ресурса панели инструментов, а затем в появившемся в окне редактирования ресурсе панели инструментов, также дважды щелкнуть левой кнопкой мыши по пустой кнопке. В результате появится окно настройки параметров кнопки. После задания параметров кнопки (см. пример на рис. 7.5) можно "нарисовать" пиктограмму кнопки и ресурс панели инструментов приобретет вид, показанный на рис. 7.3. Аналогично можно модифицировать параметры уже существующей кнопки.

Как для кнопки панели инструментов задать текст, выдаваемый в строку статуса, и текст всплывающей подсказки?

Достаточно в поле **Prompt** окна **Properties** настройки параметров кнопки задать текст, выдаваемый в строку статуса (он предшествует символу \n) и текст подсказки (он следует за символом \n).

Как можно изменить взаимное расположение кнопок панели инструментов, разделить их сепараторами или удалить кнопки?

С помощью редактора ресурсов вы можете легко изменить взаимное расположение кнопок панели инструментов. Для этого с помощью левой кнопки мыши достаточно "перетащить" соответствующие кнопки на требуемое место. Группы взаимосвязанных кнопок панели инструментов можно *разделить* сепараторами. Для этого кнопки, которые должны примыкать к сепаратору, нужно с помощью мыши сначала надвинуть друг на друга, а затем — раздвинуть. И, наконец, для *удаления* кнопки достаточно с помощью мыши переместить удаляемую кнопку за пределы панели инструментов.

В чем заключается программная поддержка панели инструментов?

См. конец *разд. 7.2*.

Как создать и настроить строку статуса?

См. *разд. 7.3*.

П1.8. Глава 8

На что требуется обратить первостепенное внимание в главе 8?

Новым в этой главе является использование редактора ресурсов для создания и настройки широко применяемых в приложениях диалоговых окон с управляющими элементами.

Что представляют собой диалоговые окна и управляющие элементы?

Важной частью Windows-приложений являются *диалоговые окна*, обеспечивающие интерактивное взаимодействие с приложением. Как правило, диалоговые окна включают *дочерние* окна, которые называют *элементами управ-*

ления. С их помощью можно задать, получить или отобразить значения соответствующих переменных приложения.

Укажите типы диалоговых окон и дайте им характеристику.

Диалоговые окна бывают двух типов — *модальные* и *немодальные*. Разница между ними заключается в способе управления потоками сообщений. Модальные диалоговые окна блокируют все остальные окна приложения так, что пользователь не может ничего сделать, пока не закроет модальное диалоговое окно. Иначе говоря, модальное окно отсекает поток сообщений, идущих от мыши и клавиатуры к родительским или *одноуровневым* окнам, делая их недоступными. Если пользователь все же пытается взаимодействовать с недоступным окном, то система предупреждает об этом звуковым сигналом. Однако модальные диалоговые окна обычно не препятствуют переключению с одного приложения на другое.

Укажите разновидности диалоговых окон и дайте им характеристику.

Диалоговые окна являются важнейшей частью пользовательского интерфейса Windows и Windows-приложений. Поэтому наряду с *пользовательскими диалоговыми окнами*, создаваемыми для нужд приложения, уже давно сложился *стандартный набор широко используемых диалоговых окон* (см. табл. 8.1). Пояснять, что собой представляют стандартные диалоговые окна и как ими пользоваться, видимо, нет необходимости. Скорее всего, вы часто сталкивались и пользовались ими.

Перечислите типы стандартных диалоговых окон Windows?

См. табл. 8.1.

Как получить самые точные и полные сведения о составе и иерархии классов библиотеки MFC? Каково место стандартных классов диалоговых окон в иерархии классов библиотеки MFC?

Самые точные и полные сведения о составе и иерархии классов библиотеки MFC можно получить из справочной системы IDE Microsoft Visual Studio 2005. Для этого достаточно выполнить команду **Help | Index** и в поле **Look for:** появившегося окна **MFC Reference — Microsoft Visual Studio 2005 Documentation — Microsoft Document Explorer** набрать **MFC, hierarhy** и посмотреть вкладки **Hierarhy Chart** и **Hierarhy Chart Categories**. Положение стандартных классов диалоговых окон в иерархии классов библиотеки MFC показано на рис. 8.1.

Что отличает диалоговые окна от других?

Прежде всего, то, что их можно проектировать визуально, с помощью редактора ресурсов. И это является очевидным преимуществом, т. к. нет необхо-

димости задавать положение элементов управления в коде, можно просто визуально поместить их на шаблоне ресурсов диалогового окна в редакторе ресурсов.

Укажите назначение диалоговых окон.

Диалоговое окно используется для того, чтобы *принять ввод/данные*, сделанные пользователем в той или иной форме.

Какие сообщения обрабатывает диалоговое окно?

Диалоговое окно обычно обрабатывает два сообщения (`WM_INITDIALOG` и `WM_COMMAND`).

Когда нужно обрабатывать сообщение *WM_INITDIALOG*?

Диалоговое окно получает сообщение `WM_INITDIALOG` после того, как оно было создано и все его элементы управления инициализированы, но до вывода его на экран. Поэтому если вы создадите свой перегруженный метод обработки сообщения `WM_INITDIALOG`, то сможете модифицировать элементы управления или данные перед тем, как диалоговое окно появится на экране.

Укажите назначение класса *CDialog* библиотеки MFC.

Библиотека классов MFC содержит класс `CDialog` — базовый класс диалогового окна. Этот класс абстрагирует шаблон ресурсов диалогового окна и включает в себя различные методы, с которыми вам необходимо ознакомиться для работы с диалоговыми окнами. Средства создания каркаса приложения MFC позволяют быстро отобразить элементы управления диалогового окна на методы вашего производного класса диалогового окна. Каждый подчиненный элемент управления диалогового окна посылает сообщения, для которых можно создать соответствующие методы обработки.

Как создать диалоговое окно?

Для создания диалогового окна следует создать объект типа `CDialog` или типа, производного от `CDialog`. Как только такой объект создан, появляются две возможности: вызвать метод `CDialog::DoModal`, чтобы создать модальное диалоговое окно, или же вызвать метод `CDialog::Create`, чтобы создать немодальное диалоговое окно. В обоих случаях на этом этапе создается диалоговое окно со своими элементами управления и оно связывается с созданным объектом MFC.

Перечислите стандартные и виртуальные методы класса `CDialog` библиотеки классов MFC и укажите их назначение.

См. табл. 8.2 и 8.3.

Какие элементы управления можно использовать в диалоговом окне?

См. рис. 8.2.

Назовите обязательные кнопки диалогового окна, укажите способы закрытия окна.

Диалоговое окно всегда снабжается двумя кнопками — **Cancel** (Отмена) и **OK**. Это окно *автоматически закрывается*, когда нажимается либо кнопка **Cancel** (Отмена), с помощью метода `CDialog::OnCancel`, либо **OK**, с помощью метода `CDialog::OnOK`, либо когда вы в своем коде вызываете метод `CDialog::EndDialog`. Также закрыть диалоговое окно можно с помощью кнопки **Закрыть**, расположенной в строке заголовка справа.

Укажите особенности работы с немодальными диалоговыми окнами.

Немодальные диалоговые окна чаще всего размещаются в динамической памяти с помощью оператора `new`. В этом случае следует перегрузить метод `OnCancel` и вызвать из него метод `DestroyWindow`. Вызов же метода базового класса CDialog::`OnCancel` приводит, в свою очередь, к вызову метода `CDialog::EndDialog`, который прячет диалоговое окно, вместо того, чтобы разрушить его, а это совсем не то, что нужно! Другой метод, за которым нужно тщательно следить в немодальных диалоговых окнах, — это `PostNcDestroy`. Так как немодальные диалоговые окна обычно создаются с помощью оператора `new`, необходимо вызывать метод `PostNcDestroy` для освобождения занятой области памяти. Впрочем, сказанное важно только для непрерывно работающих приложений с немодальными диалоговыми окнами.

Как следует добавлять в проект класс диалогового окна, его таблицу сообщений и обработчики сообщений управляющих элементов?

Файлы с определением класса диалога, производного от класса `CDialog` библиотеки MFC, следует включать в проект с помощью мастера, вызываемого командой **Project | Add Class...** после выбора проекта во вкладке **Solution Explorer**. Добавление в класс диалогового окна таблицы сообщений, прототипов и определений функций, обрабатывающих сообщения от управляющих элементов, выполняйте с помощью мастера обработки событий, вызываемого командой **Add Event Handler...** контекстного меню соответствующего управляющего элемента *(см. разд. 6.3)*.

Как создать и настроить шаблон ресурсов диалогового окна?

Шаблон ресурсов диалогового окна проще всего создать с помощью редактора ресурсов. Для этого достаточно на вкладке **Resource View** выбрать проект, выполнить команду **Project | Add Resource...** и в появившемся окне **Add**

Resource выбрать ресурс **Dialog**, а затем нажать кнопку **New**. В результате создается шаблон ресурсов диалогового окна с двумя кнопками — **OK** и **Cancel**. Размеры диалогового окна, местоположение и размеры указанных кнопок можно с помощью мыши легко изменить. В результате, на вкладке **Resource View** появится ресурс **Dialog** с некоторым идентификатором, заданным редактором ресурсов по умолчанию. Если для данного идентификатора двойным "щелчком" левой кнопки мыши загрузить ресурс диалогового окна в окно редактирования, а затем с помощью правой кнопки мыши вызвать контекстное меню и выполнить команду **Properies**, то можно задать требуемые свойства шаблона ресурсов диалогового окна.

Как в диалоговом окне разместить управляющие элементы? Укажите особенность радиокнопок. Как задать свойства радиокнопок?

Для размещения в диалоговом окне нужных *управляющих элементов* можно воспользоваться панелью управляющих элементов диалогового окна (**Toolbox**). В демонстрационном примере UsgModalDlg *(см. главу 8)* такими управляющими элементами являются три радиокнопки, с помощью которых задается стиль линии — сплошной, штриховой или пунктирный. Отличительным признаком радиокнопок является то, что активной является только одна кнопка из группы радиокнопок. Для их добавления в диалоговое окно достаточно последовательно, на панели **Toolbox**, с помощью мыши выбрать радиокнопку (Radio Button) и перетащить ее в нужное место диалогового окна. Размеры и названия радиокнопок также можно легко изменить желаемым образом. Для настройки свойств радиокнопки достаточно вызвать ее контекстное меню, выполнить команду **Properties** и в появившейся одноименной вкладке задать свойства радиокнопки. Свойства радиокнопок для демонстрационного примера UsgModalDlg показаны на рис. 8.8.

Где хранится шаблон ресурсов диалогового окна?

Шаблон ресурсов диалогового окна хранится, как составная часть, в файле проекта Resource.rc, а идентификаторы диалогового окна и его управляющих элементов — в файле resource.h. Напоминаем, что эти файлы непосредственно не следует редактировать — они поддерживаются редактором ресурсов.

В чем состоит программная поддержка модального диалогового окна в классе *FrameWnd* демонстрационного проекта UsgModalDlg?

См. *разд. 8.2.2*.

Какой класс библиотеки MFC поддерживает работу окон редактирования?

Это класс `CEdit`.

Что представляет собой окно редактирования?

Окно редактирования — это прямоугольное окно с большими возможностями. Окна редактирования выполняют множество полезных функций, значительно облегчающих обработку текста. Например, приложение Блокнот, поставляемое вместе с Windows, представляет собой простой тестовый редактор, большая часть функциональных возможностей которого определяется как раз возможностями окна редактирования.

Чем отличаются однострочные и многострочные окна редактирования?

Однострочное окно редактирования выводит весь текст в одной строке, независимо от наличия в нем символов возврата каретки. *Многострочное* окно выводит текст в несколько строк, разделенных символами возврата каретки.

Перечислите стили окна редактирования. Можно ли их комбинировать с другими стилями?

Стили окна редактирования перечислены в табл. 8.4. Их можно сочетать с общими стилями окон, доступными объектам класса `CWnd`.

Охарактеризуйте основные методы класса *CEdit*.

Методы класса `CEdit` перечислены в табл. 8.5.

Как поместить или извлечь текст из окна редактирования?

Для *извлечения* текста из окна редактирования можно использовать метод `CWnd::GetWindowText`. Чтобы *поместить текст в окно редактирования*, вызывайте метод `CWnd::SetWindowText( )`.

П1.9. Глава 9

Какие классы библиотеки MFC поддерживают работу с кнопками?

Работу с кнопками поддерживают классы `CButton` и `CBitMapButton`.

Перечислите основные виды кнопок.

Существует четыре основных *типа кнопок* — обыкновенные (нажимаемые) кнопки, кнопки **OK** и **Cancel** (используются в диалоговых окнах), радиокнопки и отмечаемые кнопки (флажки).

Что представляет собой обычная кнопка?

Кнопка — это прямоугольное окно обычно небольшого размера и имеющее название, в котором указано ее назначение.

Укажите способы применения кнопок.

Кнопки могут применяться несколькими способами. Для ввода команды пользователя используется *нажимаемая кнопка*. Если существует взаимоисключающий набор опций, то применяются *радиокнопки*. Если несколько опций не являются взаимоисключающими (пользователь одновременно может выбрать несколько возможностей), используется набор *отмечаемых кнопок*. Чтобы создать видимую рамку вокруг взаимосвязанной группы элементов управления (например, группы радиокнопок или группы отмечаемых кнопок), используется *блок группы*.

Какие элементарные стили можно использовать для кнопок?

Подобно другим окнам, кнопки могут сочетать в себе ряд *элементарных стилей окон* `CWnd` (см. листинг 2.3). Кроме общих стилей окон, для кнопок можно сочетать *элементарные стили кнопок*, перечисленные в листинге 5.1.

Укажите основные элементы таблицы сообщений от кнопок.

См. табл. 9.1.

В чем состоит отличительная особенность отмечаемых кнопок?

Они используются для задания значений нескольких опций, которые, в отличие от радиокнопок, не являются взаимоисключающими (пользователь может одновременно выбрать несколько возможностей).

Перечислите и охарактеризуйте разновидности отмечаемых кнопок.

В зависимости от значения свойства **Tri-state** из группы свойств **Behavior**, определяющих поведение кнопки, различают *стандартные отмечаемые кнопки* (два состояния, свойство **Tri-state** равно **False**) и *отмечаемые кнопки с тремя состояниями* (**Tri-state** равно **True**).

Как разместить отмечаемую кнопку в диалоговом окне?

Для *размещения в диалоговом окне отмечаемой кнопки* можно воспользоваться панелью управляющих элементов диалогового окна (**Toolbox**). В демонстрационном примере UsgCheckBox из *главы 9* для их включения в диалоговое окно достаточно последовательно, на панели **Toolbox**, с помощью мыши выбрать отмечаемую кнопку (Check Box) и перетащить ее в нужное место диалогового окна. Размеры отмечаемой кнопки можно изменить.

Как настроить свойства отмечаемой кнопки?

Для *настройки свойств отмечаемой кнопки* достаточно вызвать ее контекстное меню, выполнить команду **Properties** и в появившейся одноименной вкладке задать свойства отмечаемой кнопки.

Что собой представляет кнопка с растровым изображением? Какой класс библиотеки MFC предназначен для работы с рисованными кнопками?

Библиотека MFC содержит класс `CBitMapButton` для кнопок с растровыми изображениями, производный от класса `CButton` и наследующий все его возможности. *Кнопка с растровым изображением* — это стандартная нажимаемая кнопка, которая вместо поясняющего текста содержит растровый рисунок.

Как связать растровую кнопку с изображением?

Класс `CBitMapButton`, наследуя все методы класса `CButton`, определяет еще два метода инициализации — `LoadBitmaps` и `AutoLoad` — расширяющие возможности обычной нажимаемой кнопки и обеспечивающие удобные средства для загрузки растровых изображений, которые затем будут нарисованы в клиентской области растровой кнопки. При этом метод `AutoLoad` автоматически связывает объект `CBitMapButton` с требуемыми растровыми изображениями, пользуясь *связанными* идентификаторами кнопок и Bitmap-ресурсов.

Как из одного приложения запустить другое приложение?

С помощью стандартной функции `WinExec`.

Как включить в проект Bitmap-ресурс и задать его свойства?

Для *включения* в проект указанного ресурса достаточно на вкладке **Resource View** выбрать проект, выполнить команду **Project | Add Resource...**, в появившемся окне **Add Resource** выбрать ресурс **Bitmap** и нажать кнопку **New**. В результате создается шаблон Bitmap-изображения и появится ресурс **Bitmap** с некоторым идентификатором, заданным редактором ресурсов по умолчанию. Если для данного идентификатора двойным "щелчком" левой кнопки мыши загрузить ресурс в окно редактирования, а затем с помощью правой кнопки мыши вызвать контекстное меню и выполнить команду **Properies**, то можно задать свойства Bitmap-ресурса и нарисовать изображения.

Как создать изображение растровой кнопки в отжатом и нажатом состояниях?

Изображения кнопки в отжатом и нажатом состоянии очень похожи (практически совпадают) и отличаются только небольшим сдвигом изображения и цветом внешнего слоя. Такое решение в работающем приложении создает эффект нажатия кнопки.

Какие ограничения накладывает метод *AutoLoad* при связывании с изображениями?

Метод `AutoLoad` автоматически связывает объект `CBitMapButton` с требуемыми растровыми изображениями, пользуясь связанными идентификаторами кнопок и Bitmap-ресурсов (см. пример в проекте UsgBtnBmp *главы 9*).

Что представляет собой элемент спин (Spin) с дружественным окном?

Спин (вращатель) представляет собой элемент управления, состоящий из двух связанных кнопок с изображениями стрелок. Эти элементы управления обычно встречаются группами и предназначены для получения информации от пользователя. Нажатие на кнопки со стрелками уменьшает или увеличивает внутреннее значение вращателя, называемое текущим положением. Это значение выводится в *дружественное окно* (окно редактирования), дополнительно позволяющее пользователю вводить числовые значения. Спин и его дружественное окно выглядят и действуют как единый элемент управления. Спин автоматически посылает текущее положение прокрутки дружественному окну, соответственно изменяя его текст. Текущее положение ограничено минимальным и максимальным значениями, определяемыми приложением.

В каком классе библиотеки MFC определены функциональные возможности для элемента управления спин (вращатель)?

Все *функциональные* возможности вращателя определены в классе `CSpinButtonCtrl` библиотеки MFC. Как и для всех других управляющих элементов библиотеки MFC, класс `CSpinButtonCtrl` является производным от класса `CWnd` и наследует всю гамму его возможностей.

Перечислите элементарные стили спина и дайте им характеристику.

См. табл. 9.2.

Перечислите элементы таблицы сообщений для спина и дайте им характеристику.

См. табл. 9.3. При использовании спина с дружественным элементом редактирования достаточно использовать в таблице сообщений элемент `ON_EN_UPDATE`.

Перечислите и охарактеризуйте основные методы класса *CSpinButtonCtrl* библиотеки MFC.

См. табл. 9.4. Перечисленные методы описывают способы *получения* (get) и *установки* (set) свойств спина.

Что представляет собой полоса прокрутки?

Полоса прокрутки — это интерактивный визуальный элемент управления, способный реагировать на ввод пользователя несколькими способами. Она состоит из *бегунка*, перемещающегося по *полосе*, а также *кнопок со стрелками* на каждом конце полосы. Полоса прокрутки может быть расположена горизонтально или вертикально. При нажатии на кнопки со стрелками происходит уменьшение или увеличение внутреннего значения, называемого *позицией прокрутки*. Позицию прокрутки можно также изменить, если передвинуть бегунок или щелкнуть по полосе. Максимальное и минимальное значения позиции прокрутки устанавливаются программно.

В каком классе библиотеки MFC определены функциональные возможности полосы прокрутки?

Все *функциональные* возможности полосы прокрутки определены в классе `CScrollBar` библиотеки MFC. Как и для всех других управляющих элементов библиотеки MFC, класс `CScrollBar` является производным от класса `CWnd` и наследует все его возможности.

Перечислите элементарные стили полосы прокрутки и дайте им характеристику.

См. табл. 9.5.

Опишите прототипы методов обработки сообщений от полос прокрутки.

Прототипы методов обработки сообщений от горизонтальной и вертикальной полос прокрутки имеют вид:

```
// Обработчик сообщений от горизонтальной полосы прокрутки
afx_msg void OnHScroll( UINT nSBCode, UINT nPos,
                        CScrollBar *pScrollBar );
// Обработчик сообщений от вертикальной полосы прокрутки
afx_msg void OnVScroll( UINT nSBCode, UINT nPos,
                        CScrollBar *pScrollBar );
```

Первый параметр `nSBCode` — это один из возможных кодов уведомлений прокрутки (см. табл. 9.7), сообщающий приложению, какие действия совершает пользователь с полосой прокрутки. Второй параметр методов `OnHScroll` и `OnVScroll` — `nPos` — задает новую позицию бегунка после его перемещения. Третий параметр — `pScrollBar` — задает указатель на полосу прокрутки, которая посылает сообщение.

Перечислите и охарактеризуйте основные методы класса *CScrollBar* библиотеки MFC.

См. табл. 9.8.

Что представляет собой ползунок?

Ползунок — это также интерактивный визуальный элемент управления, состоящий из перемещаемой *ручки* и соответствующей *разметки*, отмечающей интервалы изменений. Ползунок может быть расположен горизонтально или вертикально. При перемещении ручки в одном или другом направлении происходит уменьшение или увеличение *значения ползунка*. Максимальное и минимальное значения ползунка устанавливаются программно.

В каком классе библиотеки MFC определены функциональные возможности ползунка?

Все *функциональные* возможности ползунка определены в классе `CSliderControl` библиотеки MFC. Как и для всех других управляющих элементов библиотеки MFC, класс `CSliderControl` является производным от класса `CWnd` и наследует всю гамму его возможностей.

Перечислите элементарные стили ползунка и дайте им характеристику.

См. табл. 9.9.

Что обозначает префикс *TBS_*, используемый в макросах элементарных стилей ползунка?

Все макросы стилей ползунка начинаются с префикса `TBS_`, указывающего на первоначальное название этого элемента управления (TBS, Track Bar Styles). Сейчас этот элемент управления принято называть ползунком.

Опишите прототипы методов обработки сообщений от ползунков.

Прототипы методов обработки сообщений от горизонтальной и вертикальной полос прокрутки имеют вид:

```
// Обработчик сообщений от горизонтальной полосы прокрутки
afx_msg void OnHScroll( UINT nSBCode, UINT nPos,
                        CSliderControl *pScrollBar );
// Обработчик сообщений от вертикальной полосы прокрутки
afx_msg void OnVScroll( UINT nSBCode, UINT nPos,
                        CSliderControl *pScrollBar );
```

Первый параметр — `nSBCode` — это один из возможных кодов уведомлений прокрутки (см. табл. 9.7), сообщающий приложению, какие действия совершает пользователь с ползунком. Второй параметр методов `OnHScroll` и

`OnVScroll` — `nPos` — задает новую позицию бегунка после его перемещения. Третий параметр — `pScrollBar` — задает указатель на полосу прокрутки, которая посылает сообщение. При перегрузке методов `OnHScroll` и `OnVScroll` необходимо привести этот указатель к типу `CSliderCtrl`.

Обязательно ли перегружать методы обработки сообщений от ползунков?

Методы обработки сообщений ползунков `OnHScroll` и `OnVScroll` можно, вообще говоря, не перегружать в классе, производном от класса `CDialog` библиотеки MFC. В этом случае будут использованы методы базового класса `CDialog` библиотеки MFC.

Перечислите и охарактеризуйте основные методы класса *CSliderCtrl* библиотеки MFC.

См. табл. 9.12.

П1.10. Глава 10

Что представляет собой список?

Список является удобным средством выбора, когда в приложении есть несколько связанных элементов данных, из которых нужно выбрать требуемые элементы. Список представляет собой прямоугольное окно, в котором обычно находится несколько строк.

Списки — это широко распространенные стандартные элементы управления, позволяющие выбрать один или несколько элементов, представленных в прямоугольном окне. Списки могут иметь собственные полосы прокрутки, которые дают возможность перемещаться вдоль списка данных. Обычно список предоставляет пользователю возможность выбора имени файла, каталога и т. п. Список имеет предопределенный интерфейс клавиатуры, который позволяет перемещаться по списку на один элемент с помощью клавиш-стрелок или на одну страницу с помощью клавиш <Page Up> и <Page Down>. Если для списка установлен соответствующий стиль, то список позволяет выбирать несколько элементов с помощью сочетаний клавиш <Ctrl> (или <Shift>) и мыши. Каждый элемент в списке имеет свой индекс, используемый для обращения к этому элементу, причем индексация начинается с нуля.

Перечислите имеющиеся виды списков.

Существуют две разновидности списков — список с возможностью выбора *одного* элемента (список единичного выбора) и список с возможностью выбора *нескольких* элементов (список множественного выбора).

Какой класс библиотеки MFC поддерживает работу со списками?

Библиотека MFC обеспечивает все функциональные возможности стандартного списка с помощью класса `CListBox`. Его положение в иерархии классов библиотеки MFC можно получить в справочной системе.

Укажите способы применения списков.

Списки могут быть созданы как подчиненные элементы управления любого окна непосредственно в программном коде, но, как правило, они определяются с помощью редактора ресурсов в шаблоне ресурсов диалогового окна.

Перечислите элементарные стили списка и дайте им характеристику.

См. табл. 10.1.

Укажите элементы таблицы сообщений от списков.

См. табл. 10.2.

Перечислите и охарактеризуйте основные методы класса *CListBox* библиотеки MFC.

См. табл. 10.3—10.6.

Что представляет собой комбинированный список?

Комбинированный список обеспечивает всю полноту функциональных возможностей списка в сочетании с возможностями внедренного элемента управления — статического текста или элемента редактирования. Отсюда и происходит название элемента управления — комбинированный список. Список может быть либо раскрывающимся в ответ на нажатие кнопки ▼ в правой части элемента, либо постоянно видимым. Если какой-либо элемент списка выбран, то он выводится в окне комбинированного списка.

Перечислите имеющиеся виды комбинированных списков и дайте им характеристику.

Существуют три разновидности комбинированных списков — простой комбинированный список, раскрывающийся комбинированный список и раскрывающийся список.

- *Простой комбинированный* список виден всегда, а окно редактирования является дружественным элементом управления.
- *Раскрывающийся комбинированный* список выводится на экран только тогда, когда его раскрывают, а окно редактирования является дружественным элементом.

- *Раскрывающийся* список выводится на экран тогда, когда его раскрывают, а дружественным элементом является статический текст.

Какой класс библиотеки MFC поддерживает работу с комбинированными списками?

Все *функциональные* возможности комбинированного списка определены в классе `CComboBox` библиотеки MFC, который является производным от класса `CWnd` и наследует всю гамму его возможностей.

Укажите способы применения комбинированных списков.

Комбинированный список может быть создан в программном коде как подчиненный элемент *любого* окна или же, что удобнее, использован в шаблоне ресурсов диалогового окна.

Перечислите элементарные стили комбинированного списка и дайте им характеристику.

См. табл. 10.7.

Укажите элементы таблицы сообщений от комбинированных списков.

См. табл. 10.8.

Перечислите и охарактеризуйте основные методы класса *CComboBox* библиотеки MFC.

См. табл. 10.9—10.11.

Что представляет собой индикатор прогресса?

Индикатор прогресса — это окно, обеспечивающее визуальную обратную связь во время какой-либо длительной операции, выполняемой приложением. Индикатор прогресса дает возможность наблюдать за ходом операции и по этой причине применяется только для вывода. Индикатор прогресса, как и ползунок, имеет *диапазон* и *текущую позицию*. Диапазон соответствует продолжительности операции, а текущая позиция показывает, насколько к данному моменту выполнена операция. На основе этих значений подсчитывается процент наполнения индикатора.

Какой класс библиотеки MFC поддерживает работу с индикатором прогресса?

Все *функциональные* возможности индикатора прогресса определены в классе `CProgressCtrl` библиотеки MFC, который также является производным от класса `CWnd` и наследует все его возможности. Индикатор прогресса может быть создан в программном коде как подчиненный элемент *любого* окна или же, что удобнее, использован в шаблоне ресурсов диалогового окна.

Перечислите элементарные стили индикатора прогресса и дайте им характеристику. Охарактеризуйте основные методы класса *CProgressCtrl* библиотеки MFC, используемые при работе с индикатором прогресса.

См. табл. 10.12 и 10.13.

Что представляет собой таймер? Как запустить, остановить таймер или обработать его сообщение?

Таймер является элементом управления, который периодически извещает приложение об истечении заданного интервала времени. Происходит это путем посылки сообщения `WM_TIMER`. Таймер запускается с помощью метода

```
UINT_PTR CWnd :: SetTimer( UINT_PTR nIDEvent, UINT nElapse,
      void(CALLBACK *lpfnTimer)(HWND, UINT, UINT_PTR, DWORD) );
```

Первый параметр `nIDEvent` определяет идентификатор таймера, который может быть любым беззнаковым целым числом (в Windows одновременно допускается не более 16 таймеров). Второй параметр `nElapse` задает интервал таймера в миллисекундах (в Windows реальное разрешение составляет 1/182 секунды). Последний параметр определяет функцию, которая будет получать сообщения таймера из приложения (`NULL` — такая функция отсутствует). Остановить поток сообщений таймера можно в любой момент вызовом метода

```
BOOL CWnd :: KillTimer( UINT_PTR nIDEvent );
```

Единственным параметром метода является идентификатор таймера, который нужно остановить. Сообщения каждого из таймеров можно обработать функцией-обработчиком

```
afx_msg void OnTimer( UINT nIDEvent );
```

Ее единственным параметром является идентификатор таймера.

П1.11. Глава 11

Что представляет собой динамически подключаемая библиотека?

Динамически подключаемые библиотеки весьма широко применяются в операционных системах Windows и это не случайно. DLL являются средством создания по-настоящему *модульного* программного обеспечения, обеспечивая модульность в процессе выполнения программы. Вместо "гигантских" исполняемых файлов, которые пришлось бы перестраивать и тестировать при любом, даже самом незначительном, изменении кода, гораздо эффективнее создавать компактные DLL-модули и тестировать их по отдельности.

Почему создавать DLL достаточно просто?

Объясняется это тем, что со стороны мастера создания приложений и библиотеки классов MFC поддержка DLL существенно расширена и улучшена.

Какие типы DLL поддерживаются библиотекой классов MFC? Дайте их характеристику.

Существует два типа динамически подключаемых библиотек, поддеживаемых библиотекой классов MFC — обычные DLL и DLL расширений.

Обычная (regular) DLL может быть связана с библиотекой MFC как статически, так и динамически. Она доступна любым приложениям, поддерживающим вызов DLL. Обычная DLL может экспортировать только функции языка C и не способна экспортировать классы, обычные или переопределяемые методы классов. Это объясняется тем, что каждый компилятор языка C++ использует свой метод расширения имен функций. Но при этом, внутри самой обычной DLL можно использовать классы языка C++ и классы библиотеки MFC. Ресурсы помещаются в обычную DLL точно так же, как и в само приложение. Поэтому они будут доступны любому приложению.

DLL расширений (extension DLL) могут быть динамически связаны только с библиотекой MFC и могут использоваться лишь приложениями на базе MFC. Каждая DLL расширений может свободно передавать и получать объекты классов, производных от классов библиотеки MFC.

Как в DLL расширений обеспечивается экспорт класса?

Для этого в объявление экспортируемого класса нужно добавить макрос, а обо всем остальном позаботится библиотека MFC:

```
// Объявление класса: обратите внимание на макрос AFX_EXT_CLASS
//   для экспорта класса
class AFX_EXT_CLASS MainThread : public CWinApp
{
        // ...
}
```

Для чего используется функция *DLLMain*? Когда она вызывается?

Код библиотеки времени выполнения языка C++ запускает DLL путем вызова функции `DllMain` каждый раз, когда процесс или поток подключается или отключается от DLL.

Что представляют собой процесс и поток (потоки) выполнения?

Процесс — это программа, загруженная в память и готовая к исполнению. Процесс имеет свое собственное адресное пространство (размером до 4 Гбайт)

и состоит из кода, данных и других системных ресурсов, доступных *потокам выполнения* этого процесса. Процесс должен иметь, по крайней мере, один поток, но их, при необходимости, может быть и больше. Процесс всегда начинается только с одного потока, остальные потоки создаются по мере необходимости.

Поток выполнения — это базовый элемент, которому операционная система выделяет время процессора. Каждый поток пользуется набором структур, где хранит свой контекст в то время, пока дожидается очередного кванта времени процессора. Каждый из потоков, принадлежащих одному процессу, совместно использует адресное пространство процесса и может получать доступ к его глобальным системным ресурсам. Поток может выполнять любую часть кода процесса.

Какой интерфейс имеет функция *DLLMain*?

Функция `DllMain` использует соглашение вызова `APIENTRY` и имеет три параметра:

```
BOOL WINAPI DllMain( HINSTANCE hinstDLL, DWORD fdwReason,
                     LPVOID lpvReserved );
```

Параметр `hinstDLL` — это дескриптор процесса или потока, параметр `fdwReason` показывает причину вызова функции `DllMain`, а параметр `lpvReserved` зарезервирован для использования Windows.

Какие значения может принимать параметр функции *DLLMain*, показывающий причину ее вызова?

См. табл. 11.1.

Как должен выглядеть каркас функции *DLLMain*? Всегда ли нужно писать ее текст в исходном коде?

Реализация каркаса необязательной функции `DllMain` может выглядеть следующим образом:

```
BOOL WINAPI DllMain(
    HINSTANCE        hinstDLL,
    DWORD            fdwReason,
    LPVOID           lpvReserved )
{
    switch( fdwReason )
    {
        case DLL_PROCESS_ATTACH:
            //
```

```
            case DLL_THREAD_ATTACH:
                //
            case DLL_PROCESS_DETACH:
                //
            case DLL_THREAD_DETACH:
                //
        }
        return TRUE;
    }
```

Если DLL не нуждается в выделении памяти или других ресурсов, можно полностью исключить функцию `DllMain` из своего исходного кода. Компилятор автоматически подставит функцию `DllMain`, используемую по умолчанию.

Как создать DLL расширений с помощью мастера?

См. *разд. 11.2.1*.

П1.12. Глава 12

Как запустить MFC Application Wizard?

В поле **Recent Projects** вкладки **Start Page** выполните команду **Create: Project...**. В появившемся окне **New Projects** задайте параметры проекта в соответствии с приведенным ранее рис. 12.1 главы 12 и нажмите кнопку **OK**. В результате появится окно **MFC Application Wizard — MDIApp**, позволяющее выполнить требуемую настройку каркаса приложения на базе MFC (см. рис. 12.2).

Как получить справку по текущему окну настройки MFC Application Wizard?

С помощью кнопки с пиктограммой в виде знака вопроса **?** или с помощью клавиши <F1> можно получить справочную информацию по любому из параметров текущего окна настройки приложения. Для этого достаточно с помощью левой кнопки мыши нажать кнопку **?** или клавишу <F1>. В результате появится окно **MFC Application Wizard — Microsoft Visual Studio 2005 Documentation — Microsoft Document Explorer**, позволяющее получить справку по любому параметру настройки, представленному в текущем окне настройки.

Укажите назначение кнопок, имеющихся в текущем окне настройки MFC Application Wizard.

При нажатии кнопки **Finish** последующие шаги настройки приложения не выполняются и используются параметры приложения, предлагаемые масте-

ром по умолчанию. При нажатии кнопки **Previous** происходит возврат к предыдущему окну настройки и вы можете скорректировать свои прежние решения. При нажатии кнопки **Cancel** построение приложения прекращается. И, наконец, при нажатии кнопки **Next** выполняется переход к следующему окну настройки.

Укажите назначение и возможности исходного окна настройки MFC Application Wizard.

Для задания параметров создаваемого каркаса имеется две возможности — путем нажатия кнопки **Finish** принять настройки, предлагаемые мастером по умолчанию, или выполнить выборочную настройку каркаса приложения на базе MFC. Во втором случае следует либо выбрать следующую вкладку **Application Type** (первую из оставшихся вкладок), либо нажать кнопку **Next**. После нажатия кнопки **Next** появится следующее окно настройки **Application Type**.

Укажите назначение и возможности окна настройки *Application Type* мастера MFC Application Wizard.

В диалоговом окне мастера **Application Type** (см. рис. 12.3) можно указать мастеру, должно ли приложение поддерживать работу с одним документом (SDI, Single Document Interface), с несколькими документами (MDI, Multiple Document Interface) или же это будет диалоговое окно. SDI-приложение, поддерживающее работу с одним документом, создать проще всего, поскольку в этом случае не нужно учитывать взаимодействия между разными документами, одновременно открытыми одним приложением. Вместе с тем, MDI-приложение более универсально. Поэтому согласитесь с вариантом, предлагаемым мастером. Обратите внимание на то, как меняется рисунок-пояснение в заголовке окна при изменении типа приложения. Убедитесь, что установлена предлагаемая по умолчанию опция поддержки архитектуры документ/представление **Document/View architecture support**. Чтобы использовать опцию **Use Multi-Byte Character Set** выключите выбираемую кнопку **Use Unicode libraries**. Для остальных опций согласитесь со значениями, предлагаемыми по умолчанию (английский язык для ресурсов, стандартный проект MFC, динамическое подключение библиотеки MFC). Щелкните по кнопке **Next** и на экране появится следующее окно настройки **Compound Document Support**.

Укажите назначение и возможности окна настройки *Compound Document Support* мастера MFC Application Wizard.

В окне настройки **Compound Document Support** (см. рис. 12.4) можно выбрать тип приложения OLE (контейнер и/или сервер). Для получения дополнительной информации воспользуйтесь справочной системой. Рассмотрение

указанных вопросов выходит за рамки учебного пособия. Поэтому оставьте активной опцию **None**, предлагаемую по умолчанию, и щелкните по кнопке **Next**, чтобы перейти к следующему окну настройки **Document Template Strings**.

Укажите назначение и возможности окна настройки *Document Template Strings* мастера MFC Application Wizard.

В этом окне (см. рис. 12.5) можно задать параметры типа документа, связанного с приложением. Это позволит операционной системе Windows связать тип документа с приложением. В дальнейшем, когда пользователь дважды щелкнет на имени файла, операционная система автоматически запустит приложение, связанное с ним. Подробнее о настроечных параметрах этого окна можно узнать, используя кнопку **?** или <F1>. В поле **File Extension** введите расширение mye и нажмите кнопку **Next**, чтобы перейти к следующему окну настройки **Database Support**.

Укажите назначение и возможности окна настройки *Database Support* мастера MFC Application Wizard.

Установки в окне настройки **Database Support** (см. рис. 12.6) делаются только в том случае, если в приложении будет осуществляться работа с базами данных. Оставьте опцию **None**, предлагаемую по умолчанию. Щелкните на кнопке **Next** и вы перейдете к окну настройки **User Interface Features**.

Укажите назначение и возможности окна настройки *User Interface Features* мастера MFC Application Wizard.

С помощью этого окна (см. рис. 12.7) можно добавлять в программу различные средства, определяющие интерфейс приложения и параметры главного и дочерних окон приложения. К их числу относятся, например, опция главного окна **Tick frame**, позволяющая изменять размеры главного окна с помощью рамки, опции создания кнопок минимизации и максимизации (главное окно и дочерние окна) и опции создания в главном окне системного меню, панели инструментов, строки статуса и др. Щелкните на кнопке **Next** и вы перейдете к окну настройки **Advanced Features**.

Укажите назначение и возможности окна настройки *Advanced Features* мастера MFC Application Wizard.

Окно **Advanced Features** (см. рис. 12.8) позволяет задать контекстно-зависимую помощь, поддержку печати, вид представления управляющих элементов и т. п. Включите контекстно-зависимую помощь, а для других параметров примите значения, предлагаемые мастером по умолчанию. Щелкните на кнопке **Next** и вы перейдете к последнему окну настройки **Generated Classes**.

Укажите назначение и возможности окна настройки *Generated Classes* мастера MFC Application Wizard.

В последнем окне мастера **Generated Classes** (см. рис. 12.9) отображается список классов, которые будут сгенерированы для нашего приложения (`CMDIAppView`, `CMDIAppApp`, `CMainFrame`, `CChildFrame` и `CMDIAppDoc`). Для класса `CMDIAppView`, реализующего функции области просмотра, в списке **Base class** можно выбрать базовый класс (`CEditView`, `CFormView`, `CHtmlEditView`, `CHtmlView`, `CListView`, `CRichEditView`, `CScrollView`, `CTreeView` или `CView`), от которого он будет порождаться. Класс `CEditView` реализует функции текстового редактора и будет использован в нашем приложении. После щелчка по кнопке **Finish** будут сгенерированы файлы приложения, в том числе файл ReadMe.txt с итоговым отчетом, подготовленным мастером.

Как русифицировать ресурсы каркаса приложения, полученного с помощью MFC Application Wizard?

Выполните команду **Project | Settings...**, в появившемся окне **MDIApp Property Pages** выберите вкладку **Resources | General**, в поле **Culture** установите язык **Русский (0x419)** и нажмите кнопку **OK**. Затем если окно **Properties Window** отсутствует, откройте его с помощью комбинации клавиш <Alt>+<Enter> или командой **View | Oher Window | Properties Window**. Во вкладке **Resource View** выберите, например, ресурс **Accelerator | IDR_MAINFRAME**, выполните команду **Properties** его контекстного меню и в окне **Properties** выберите язык **Русский**. Аналогичным образом русифицируйте все остальные ресурсы.

Как модифицировать и русифицировать строку заголовка главного окна каркаса приложения на базе MFC, полученного с помощью MFC Application Wizard?

Перейдите во вкладку **Resource View — MDIApp**, последовательно раскройте ресурсы **MDIApp**, **MDIApp.rc**, **String Table**, двойным щелчком мыши по ресурсу **String Table** загрузите его в окно редактирования, на вкладке **MDIApp.rc (String Table)** отредактируйте заголовок в соответствии с рис. 12.17 и нажмите клавишу <Enter>.

Как русифицировать диалоговое окно *About* (О программе) каркаса приложения, полученного с помощью MFC Application Wizard?

Во вкладке **Resource View — MDIApp** раскройте ресурс **Dialog**, двойным щелчком мыши по идентификатору **IDD_ABOUTBOX** загрузите ресурс в окно редактирования, во вкладке **MDIApp.rc (IDD_ABOUTBOX — Dialog)** настройте свойства **Caption** шаблона диалогового окна и статических текстов в соответствии с рис. 12.18.

Как русифицировать меню, команды меню и кнопки панели инструментов каркаса приложения, полученного с помощью MFC Application Wizard?

Русификацию меню рассмотрим на примере меню **File** главного окна приложения (см. рис. 12.11). Во вкладке **Resource View — MDIApp** раскройте ресурс **Menu**, двойным щелчком мыши по идентификатору **IDR_MDIAppTYPE** загрузите ресурс меню в окно редактирования, во вкладке **MDIApp.rc (IDR_MDIAppTYPE — Menu)** выберите меню **File** и настройте его свойство **Caption** в соответствии с рис. 12.20. Обратите внимание на то, как задается "горячая" клавиша в имени меню (на рис. 12.20 такой клавишей является <Ф>). Аналогичным образом русифицируются остальные меню каркаса MDI-приложения. При этом важно, чтобы "горячие" клавиши меню в строке меню были уникальными. Русификацию команд меню и кнопок панели инструментов рассмотрим на примере команды **New** меню **File** и кнопки **New** панели инструментов. В окне редактирования, во вкладке **MDIApp.rc (IDR_MDIAppTYPE — Menu)** выберите команду **New** меню **File** и настройте ее свойства в соответствии с рис. 12.20. Обратите внимание на то, как задается "горячая" клавиша в имени команды меню (на рис. 12.20 такой клавишей являктся <Н>). Обратите внимание на свойства **Caption**, **Prompt** и на то, что идентификаторы (свойство **ID**) у команды **New** меню **File** и кнопки **New** панели инструментов совпадают. Поэтому достаточно русифицировать только либо команду **New** меню **File**, либо кнопку **New** панели инструментов. Аналогичным образом русифицируются другие команды остальных меню и кнопки на панели инструментов. При этом также следует обеспечить уникальность "горячих" клавиш команд в рамках соответсттвующего меню ("горячие" клавиши команд разных меню могут совпадать).

Как русифицировать ресурс таблицы строк каркаса приложения, полученного с помощью MFC Application Wizard?

В окне редактирования выберите вкладку **MDIApp.rc (String Table)**, отредактируйте некоторые ее строки в соответствии с рис. 12.21.

П1.13. Глава 13

Как добавить новое меню с командами в каркас приложения, полученный с помощью MFC Application Wizard?

Для добавления нового меню с командами выберите вкладку **Resource View**, раскройте все ресурсы и двойным щелчком левой кнопки мыши по идентификатору меню загрузите ресурс меню в окно редактирования. Добавьте новое меню с командами и настройте их (см. рис. 13.1—13.3). Сохраните выполненные изменения.

Как добавить в программный код прототип, макрос и определение функции-обработчика команды меню?

Для этого достаточно выполнить команду **Add Event Handler...** контекстного меню команды, сконфигурировать программный код (см. рис. 13.4) и нажать кнопку **Add and Edit**. Далее в каркас блока функции-обработчика следует добавить необходимый программный код.

Как добавить кнопку на панель инструментов каркаса приложения, полученного с помощью MFC Application Wizard?

Для добавления в панель инструментов новой кнопки дважды щелкните левой кнопкой мыши по идентификатору панели инструментов и она будет загружена в окно редактирования. Для отделения с помощью сепаратора новой кнопки, расположенной справа от существующих, достаточно ее просто перетащить с помощью мыши немного вправо. Чтобы настроить параметры кнопки, следует щелкнуть по ней два раза левой кнопкой мыши, задать ее свойства в окне **Properties** и нарисовать пиктограмму кнопки (см. рис. 13.6).

Какие разновидности справочных систем поддерживаются в IDE корпорации Microsoft?

В ранних версиях IDE поддерживались две справочные системы — традиционная система WinHelp, основанная на RTF-файлах [7], и новая справочная система HTML Help, в основе которой лежат откомпилированные HTML-файлы. В последней версии IDE Microsoft Visual Studio 2005 поддерживается только более удобная и перспективная новая справочная система HTML Help.

Какими способами можно получить доступ к справочной системе приложения на базе MFC?

Доступ к справочной системе приложения на базе MFC можно получить несколькими способами (все они поддерживаются в каркасе приложения, полученного с помощью MFC Application Wizard). Чаще всего, как более удобный и оперативный, используется *режим интерактивной справки*. Режим "Что это такое?" включается при одновременном нажатии пользователем клавиш <Shift>+<F1> или кнопки на панели инструментов с пиктограммой, содержащей стрелку и знак вопроса. При этом указатель мыши принимает также вид стрелки со знаком вопроса и для получения справки о команде, кнопке или области окна пользователь щелкает мышью на соответствующем элементе интерфейса. Режим интерактивной справки выключается после вывода справки на экран нажатием клавиши <Esc> или после переключения на другое приложение. Другим способом получения справочной информации является использование меню **Help** (Справка), имеющегося в большинстве оконных приложений.

И, наконец, справочную информацию можно получить с помощью клавиши <F1>. При ее нажатии, когда фокус ввода находится в активном окне приложения, диалоговом окне или на команде меню или кнопке панели инструментов, вызывается раздел справочной системы, посвященный выбранному элементу интерфейса (например, команде меню).

Что такое гипердокумент (HTML-документ)?

Обычный документ, например обычная книга, имеет линейную структуру — один предмет изложения следует за другим в определенном порядке. В гипердокументе (HTML-документ или HTML-файл) можно, используя *гиперссылки*, перейти к любому его разделу, перемещаясь по документу в *произвольном* порядке.

Что представляет собой текстовое описание темы справки?

Для каждой темы (Topic), которую нужно включить в справку, следует подготовить текстовое описание. *Текстовое описание* оформляется либо в виде HTML-файла, либо в виде набора таких файлов, объединенных гиперссылками. В последнем случае один из файлов будет *основным* (в нем, например, может быть оглавление), а остальные — *вспомогательными* (на них должны указывать гиперссылки из основного файла).

Как запустить Microsoft Help Workshop?

В IDE Microsoft Visual Studio 2005 перейдите в окно **Solution Explorer**, выберите в папке **HTML Help Files** проекта файл проекта с расширением hhp, выполните команду **Open With...** его контекстного меню, в появившемся окне выберите **Microsoft® HTML Help Workshop (HTML Help)** и нажмите кнопку **OK**. В результате появится Microsoft Help Workshop.

Как выполнить русификацию проекта справки?

В Microsoft Help Workshop выполните команду **Change project options**, в появившемся окне **Options** задайте русский язык и нажмите кнопку **OK**.

Как создать текстовое описание темы справки (поясните на примере демонстрационного проекта MDIApp3)?

В среде проекта справки выполните команду **New**, в появившемся диалоговом окне **New** выберите **HTML File** и нажмите кнопку **OK**. В следующем диалоговом окне (см. рис. 13.12) задайте заголовок файла (для файла ChildWindow.htm — ChildWindow) и нажмите кнопку **OK**. Заголовок файла будет использоваться при некоторых вариантах получения справочной информации. Для сохранения файла под нужным именем в среде проекта

справки выполните команду **File | Save File As...**, в окне **Сохранить как** укажите имя файла ChildWindow.htm и нажмите кнопку **Сохранить**. В результате будет создан пустой файл. Таким же образом создайте пустые файлы Load.htm, Unload.htm с заголовками Load и Unload. Заполнение созданных файлов требуемой информацией удобнее проводить в IDE. Так для заполнения файла ChildWindow.htm выполните команду **Open File**, из папки **help** проекта выберите файл ChildWindow.htm и нажмите кнопку **Open**. В окне редактирования появится пустой файл (см. рис. 13.13). Для заполнения основного файла справки удобнее всего скопировать содержимое какого-либо аналогичного существующего файла с гиперссылками на другие файлы справки, например, menu_file.htm и откорректировать его надлежащим образом. Вид этого файла после такой корректировки показан на рис. 13.14. Для настройки гиперссылок, выделенных синим цветом и подчеркиванием, следует с помощью правой кнопки мыши вызвать контекстное меню и выполнить в нем команду **Properties**. В появившемся окне указать имя файла, соответствующего гиперссылке, и нажать кнопку **OK** (см. рис. 13.15). Для гиперссылки "Загрузить" таким файлом является Load.htm, а для гиперссылки "Выгрузить" — файл Unload.htm. Аналогичным образом заполните файлы Load.htm и Unload.htm. Эти файлы не содержат гиперссылок и их заполнять проще (см. рис. 13.16—13.17).

Как включить в файл справки графическое изображение?

Для *включения в файл справки графического изображения* достаточно файл справки поместить в окно редактирования, выбрать для его отображения вкладку **Source** и вставить в текст файла строки, заключенные в рамку:

```
<p>
    Акселератор отсутствует</p>
<p>
    Панель инструментов:
    <IMG alt="" src="file:///C:\Toolbar.bmp"></p>
<p>
    Кнопка "Загрузить" на панели инструментов вторая
    справа.</p>
```

В выделенных строках подстрока C:\Toolbar.bmp задает файл с графическим изображением, включая полный путь к нему.

Как добавить в раздел *FILES* проекта справки подготовленные файлы справки?

С этой целью в Microsoft Help Workshop выберите заголовок сегмента **[FILES]**, выполните команду **Add/Remove topic files**, в появившемся окне

Topic Files нажмите кнопку **Add**, в новом диалоговом окне, удерживая нажатой клавишу <Ctrl>, выберите созданные файлы справки и нажмите кнопки **Открыть** и **OK**. Кнопкой **Save Project, contens and index files** сохраните сделанные изменения.

Как обеспечить получение контекстной справки для добавленных управляющих элементов (поясните на примере демонстрационного проекта MDIApp3)?

В Microsoft Help Workshop, в разделе **ALIASES** поставьте в соответствие идентификаторы элементов управления HTML-файлам, в которых эти элементы описаны. Это позволит получать по этим элементам контекстную справку. Идентификаторы элементов определены в файле hlp\HTMLDefines.h (команда **Загрузить** имеет идентификатор `HID_LOADCHILDWINDOW` и **Выгрузить** — идентификатор `HID_UNLOADCHILDWINDOW`). Как правило, используются HTML-файлы соответствующих тем. Чтобы это выполнить, в среде **HTML Help Workshop** выберите вкладку **[ALIAS]**, активизируйте команду **HtmHelp API information**, в появившемся диалоговом окне выберите вкладку **Alias**, нажмите кнопку **Add** и настройте окно **Alias** в соответствии с рис. 13.18. Аналогичным образом установите соответствие `HID_UNLOADCHILDWINDOW` и Unload.htm. В завершение нажмите кнопку **OK**, а для сохранения указанных изменений, как и ранее, выполните команду **Save Project, contens and index files**.

Как добавить в оглавление созданную тему (поясните на примере демонстрационного проекта MDIApp3)?

С помощью **HTML Help Workshop** на вкладке **Contents** в контекстном меню для **Menus** выполните команду **Insert Topic...**, в появившемся окне **Table of Contens Entry** в поле **Entry title** введите "Дочернее окно", нажмите кнопку **Add**, далее с помощью кнопки **Browse** выберите файл ChildWindow.htm. Для завершения и сохранения последовательно нажмите кнопку **Открыть**, дважды нажмите кнопку **OK** и во вкладке **Project** выполните команду **Save Project, contens and index files**.

Как добавить в справочную систему ключевые слова для новой темы (поясните на примере демонстрационного проекта MDIApp3)?

Для обеспечения всех возможностей получения справки на вкладке **Index** добавьте *ключевые слова*. При этом указывайте ключевое слово и заголовок HTML-файла, в котором идет речь о соответствующем понятии. С этой целью, в среде **HTML Help Workshop**, выберите вкладку **Index**, нажмите кнопку, снабженную всплывающей подсказкой **Insert a Keyword**, а затем сконфигурируйте появившееся диалоговое окно **Index Entry** в соответствии с

рис. 13.19. Для этого в поле **Keyword** введите "Дочернее окно", нажмите кнопку **Add** и появится окно **Path or URL**. В его списке **HTML titles** выберите заголовок ChildWindow и дважды нажмите кнопку **OK**. Аналогичным образом установите еще две связи:

- Загрузить — Load — Load.htm;
- Выгрузить — Unload — Unload.htm.

Сохраните все сделанные изменения с помощью кнопки **Save Project, contens and index files** из вкладки **Project** и закройте среду **HTML Help Workshop**.

ПРИЛОЖЕНИЕ 2

Тесты и курсовое проектирование. Варианты заданий

Далее приводятся варианты тестов по основным темам, рассмотренным в учебном пособии. *Тесты* можно использовать при проведении практических занятий и при экзаменационном тестировании. Наряду с тестами в приложении содержатся варианты заданий для курсового проектирования.

П2.1. Низкоуровневое проектирование Windows-приложений (гл. 2). Варианты тестов

Совет

Для сокращения затрат времени при ответе на приводимые далее тестовые задания используйте копию имеющегося на компакт-диске проекта простого оконного приложения FrameWnd, использующего Windows API *(см. главу 2)*.

Вариант 1

Создайте простое оконное приложение, использующее Windows API. Запретите команду **Закрыть** системного меню. Придумайте способ (способы) завершения работы приложения.

Вариант 2

Создайте простое оконное приложение, использующее Windows API, с пиктограммой в виде вопросительного знака.

Вариант 3

Создайте простое оконное приложение, использующее Windows API, с курсором в виде "песочных часов".

Вариант 4

Создайте простое оконное приложение, использующее Windows API, с серым цветом фона окна.

Вариант 5

Создайте простое оконное приложение, использующее Windows API, с заголовком, содержащим ваши атрибуты (например — фамилия, должность).

Вариант 6

Создайте простое оконное приложение, использующее Windows API. Запретите изменение размеров окна с помощью его рамки.

Вариант 7

Создайте простое оконное приложение, использующее Windows API. Окно должно иметь горизонтальную полосу прокрутки.

Вариант 8

Создайте простое оконное приложение, использующее Windows API. Окно должно первоначально отображаться в виде пиктограммы.

Вариант 9

Создайте простое оконное приложение, использующее Windows API. Окно должно первоначально отображаться в максимально возможном размере.

Вариант 10

Создайте простое оконное приложение, использующее Windows API. Кнопка **Закрыть** (расположена в правой части строки заголовка окна) должна быть неактивной. Придумайте способ (способы) завершения работы приложения.

Вариант 11

Создайте простое оконное приложение, использующее Windows API. Кнопка **Свернуть** (расположена второй справа в строке заголовка окна) должна быть неактивной.

Вариант 12

Создайте простое оконное приложение, использующее Windows API, без системного меню. Придумайте способ (способы) завершения работы приложения.

Вариант 13

Создайте простое оконное приложение, использующее Windows API. В главном окне предусмотрите вывод цветной "картинки" с поясняющей надписью.

Картинка должна содержать несколько замкнутых контуров (не эллипсы и не прямоугольники), закрашенных различными цветами.

Вариант 14

Создайте простое оконное приложение, использующее Windows API, с заданными размерами и местоположением на экране.

П2.2. Высокоуровневое проектирование Windows-приложений на базе библиотеки классов MFC (гл. 4). Варианты тестов

Совет

Для сокращения затрат времени при ответе на приводимые далее тестовые задания используйте копии имеющихся на компакт-диске проектов простых оконных приложений FrameWnd и FrameWnd_PCH, использующих библиотеку классов MFC (проекты рассмотрены в *главе 4*).

Вариант 1

Создайте простое оконное приложение, использующее библиотеку классов MFC, с заголовком, содержащим ваши атрибуты (например — фамилия, должность).

Вариант 2

Создайте простое оконное приложение, использующее библиотеку классов MFC, окно которого должно иметь горизонтальную полосу прокрутки.

Вариант 3

Создайте простое оконное приложение, использующее библиотеку классов MFC, окно которого должно первоначально отображаться в виде пиктограммы.

Вариант 4

Создайте простое оконное приложение, использующее библиотеку классов MFC, окно которого должно первоначально отображаться в максимально возможном размере.

Вариант 5

Создайте простое оконное приложение, использующее библиотеку классов MFC. Кнопка **Закрыть** (расположена в правой части строки заголовка окна) должна быть неактивной. Придумайте способ (способы) завершения работы приложения.

Вариант 6

Создайте простое оконное приложение, использующее библиотеку классов MFC. Кнопка **Свернуть** (расположена второй справа в строке заголовка окна) должна быть неактивной.

Вариант 7

Создайте простое оконное приложение, использующее библиотеку классов MFC, без системного меню. Придумайте способ (способы) завершения работы приложения.

Вариант 8

Создайте простое оконное приложение, использующее библиотеку классов MFC. В главном окне предусмотрите вывод цветной "картинки" с поясняющей надписью. Картинка должна содержать несколько замкнутых контуров (не эллипсы и не прямоугольники), закрашенных различными цветами.

Вариант 9

Создайте простое оконное приложение, использующее библиотеку классов MFC, с заданными размерами и местоположением на экране.

Вариант 10

Создайте простое оконное приложение, использующее библиотеку классов MFC, окно которого не должно допускать изменение размеров с помощью мыши и рамки окна.

П2.3. Windows-приложения с дочерними окнами (гл. 5). Варианты тестов

Совет

Для сокращения затрат времени при ответе на приводимые далее тестовые задания используйте копии проектов оконных приложений WndBtn, WndBtn_MFC, WndWndBtn и WndWndBtn_MFC, имеющихся на компакт-диске (проекты рассмотрены в *главе 5*).

Вариант 1

Создать приложение с использованием API, в главном окне которого имеются две кнопки, при нажатии которых должны создаваться два разных дочерних окна, которые имеют различный текстовый вывод и различные фоны

клиентской области окна. В первом из дочерних окон системное меню должно отсутствовать, но должна быть кнопка для завершения приложения.

Вариант 2

Создать приложение на базе MFC, в главном окне которого имеются две кнопки, при нажатии которых должны создаваться два разных дочерних окна, которые имеют различный текстовый вывод. В первом из дочерних окон системное меню должно отсутствовать, но должна быть кнопка для завершения приложения. Второе дочернее окно не должно допускать изменения своих размеров с помощью рамки и мыши.

Вариант 3

Создать приложение с использованием API, в главном окне которого имеются три кнопки. При нажатии первых двух кнопок должны создаваться два разных дочерних окна, которые имеют различный текстовый вывод и различные фоны клиентской области окна. В первом из дочерних окон системное меню должно отсутствовать, а во втором — должна содержаться кнопка, активизация которой должна привести к завершению работы приложения. При нажатии третьей кнопки в главном окне приложение также должно завершить свою работу.

Вариант 4

Создать приложение на базе MFC, в главном окне которого имеются три кнопки. При нажатии первых двух кнопок должны создаваться два разных дочерних окна, которые имеют различный текстовый вывод. В первом из дочерних окон системное меню должно отсутствовать, а во втором — должна содержаться кнопка, активизация которой должна привести к завершению работы приложения. При нажатии третьей кнопки в главном окне приложение также должно завершить свою работу.

Вариант 5

Создать приложение с использованием API, в главном окне которого имеется кнопка, при нажатии которой должно создаваться дочернее окно с текстовым выводом и кнопкой. При нажатии кнопки в дочернем окне создается следующее дочернее окно. Все окна должны иметь различный текстовый вывод и различные фоны клиентской области окна.

Вариант 6

Создать приложение на базе MFC, в главном окне которого имеется кнопка, при нажатии которой должно создаваться дочернее окно с текстовым выводом и кнопкой. При нажатии кнопки в дочернем окне создается следующее дочернее окно. Все окна должны иметь различный текстовый вывод.

Вариант 7

В приложении WndBtn_MFC из *главы 5* заменить размещение кнопок в динамической памяти их статическим размещением.

Вариант 8

В приложении WndWndBtn_MFC из *главы 5* обеспечить оригинальную обработку сообщений, например текстовый вывод в главном и дочернем окнах должен быть различным.

П2.4. Ресурсы: меню, ускорители и таблица строк. "Горячие" клавиши (гл. 6). Варианты тестов

Совет

Для сокращения затрат времени при ответе на приводимые далее тестовые задания используйте копии проектов оконных приложений UsgMenu и UsgMenu_MFC, имеющихся на компакт-диске (рассмотрены в *главе 6*).

Вариант 1

На базе копии проекта UsgMenu создать Windows-приложение, в главном окне которого имеется строка меню с двумя меню — первое меню предназначено для завершения работы приложения (это должно происходить сразу после активизации меню), а второе меню — для создания дочернего окна с текстовым выводом (это должно происходить сразу после активизации меню). Предусмотреть работу с меню с использованием "горячих" клавиш и акселераторов.

Вариант 2

На базе копии проекта UsgMenu_MFC создать Windows-приложение, в главном окне которого имеется строка меню с двумя меню — первое меню предназначено для завершения работы приложения (это должно происходить сразу после активизации меню), а второе меню — для создания дочернего окна с текстовым выводом. Предусмотреть работу с меню с использованием "горячих" клавиш и акселераторов. Меню прижать к правой границе строки меню.

Вариант 3

На базе копии проекта UsgMenu создать Windows-приложение, в главном окне которого имеется строка меню с двумя меню — первое меню предназначено для завершения работы приложения, а второе — для создания дочернего окна с цветной картинкой. Указанные действия должны происходить сразу

же после активизации соответствующего меню. Предусмотреть работу с меню с использованием "горячих" клавиш и акселераторов. В строке меню одно меню расположить справа, а другое — слева.

Вариант 4

На базе копии проекта UsgMenu_MFC создать Windows-приложение, в главном окне которого имеется строка меню с одним меню **Стиль карандаша,** объединяющим все команды. Команду **Выход** отделить от остальных сепаратором. Предусмотреть работу с использованием "горячих" клавиш и акселераторов. Меню расположить справа.

Вариант 5

На базе копии проекта UsgMenu создать Windows-приложение, в главном окне которого имеется строка меню с одним меню **Стиль карандаша**, объединяющим все команды. Команду **Выход** отделить от остальных команд сепаратором. Предусмотреть работу с использованием "горячих" клавиш и акселераторов. Меню расположить справа.

Вариант 6

В копии проекта UsgMenu_MFC удалить меню **Файл** и все из программного кода, что относилось к удаленному меню. Добавить акселератор, действие которого было бы эквивалентно действию удаленной команды **Выход**.

Вариант 7

В копии проекта UsgMenu_MFC в меню **Стиль карандаша** вначале удалите команду **Точечный** и относящийся к ней программный код, а затем с помощью мастера обработки событий добавьте эту же команду.

Вариант 8

В копии проекта UsgMenu_MFC сделайте недоступным меню **Стиль карандаша**.

П2.5. Ресурсы: панели инструментов, строка статуса и всплывающие подсказки (гл. 7). Варианты тестов

Совет

Для сокращения затрат времени при ответе на приводимые далее тестовые задания используйте копию проекта оконного приложения UsgToolStatusBars, имеющегося на компакт-диске (рассмотрено в *главе 7*).

Вариант 1

На базе копии проекта UsgToolStatusBars создать Windows-приложение, в главном окне которого имеется панель инструментов с двумя кнопками-пиктограммами и строка статуса. Предусмотреть для кнопок панели инструментов всплывающие подсказки и выдачу в строку статуса информации об их назначении. Одна из кнопок панели инструментов предназначена для создания дочернего окна с цветной "картинкой", а оставшаяся кнопка — для завершения работы приложения.

Вариант 2

На базе копии проекта UsgToolStatusBars создать Windows-приложение, в главном окне которого имеются две панели инструментов, каждая с одной кнопкой-пиктограммой, и строка статуса. Предусмотреть для кнопок панелей инструментов всплывающие подсказки и выдачу в строку статуса информации об их назначении. Одна из кнопок панелей инструментов предназначена для создания дочернего окна с цветной "картинкой", а другая кнопка — для завершения работы приложения.

Вариант 3

На базе копии проекта UsgToolStatusBars создать Windows-приложение со строкой статуса в верхней части окна и панелью инструментов в нижней части окна.

Вариант 4

В копии демонстрационного проекта добавьте на панель инструментов еще три кнопки, действие которых было бы эквивалентно командам меню **Стиль карандаша**. Отделите добавленные кнопки от существовавшей ранее сепаратором. Обеспечьте для всех кнопок всплывающие подсказки и сообщения в строке статуса.

Вариант 5

В копии демонстрационного проекта создайте еще одну панель инструментов с кнопками, действие которых было бы эквивалентно командам меню **Стиль карандаша**. Дополнительную панель инструментов разместите внизу, под строкой статуса. Обеспечьте для всех кнопок всплывающие подсказки и сообщения в строке статуса.

Вариант 6

В копии демонстрационного проекта поместите панель инструментов вдоль левой границы окна и запретите ее перемещения.

П2.6. Ресурсы: окна диалога и управляющие элементы (гл. 8). Варианты тестов

Совет

Для сокращения затрат времени при ответе на приводимые далее тестовые задания используйте копии демонстрационных проектов UsgModalDlg и NoModalDlgUsgEdit, имеющихся на компакт-диске (рассмотрены в *главе 8*).

Вариант 1

Модифицируйте копию проекта UsgModalDlg, чтобы использовалось немодальное диалоговое окно. Сохраните остальную функциональность исходного проекта.

Вариант 2

Модифицируйте копию проекта NoModalDlgUsgEdit, чтобы использовалось модальное диалоговое окно. Сохраните остальную функциональность исходного проекта.

Вариант 3

Модифицируйте копию проекта NoModalDlgUsgEdit, чтобы для работы с окном редактирования использовалась переменная, являющаяся членом класса `Dialog`, с типом `double` вместо типа `CString`.

Вариант 4

Модифицируйте копию проекта UsgModalDlg, чтобы в окне диалога использовались кнопки, обрабатываемые так же, как и в демонстрационном проекте WndBtn_MFC из *главы 5*.

Вариант 5

Модифицируйте копию проекта NoModalDlgUsgEdit, чтобы в диалоговом окне использовались кнопки, обрабатываемые так же, как и в демонстрационном проекте WndBtn_MFC из *главы 5*.

П2.7. Управляющие элементы: кнопки и элементы прокрутки (гл. 9). Варианты тестов

Совет

Для сокращения затрат времени при ответе на приводимые далее тестовые задания используйте копии демонстрационных проектов UsgGroupBox, UsgCheckBox,

UsgBtnBmp, UsgSpinEdit, UsgSlider и UsgScroll, имеющихся на компакт-диске (рассмотрены в *главе 9*).

Вариант 1

Модифицируйте копию проекта UsgGroupBox так, чтобы в модальном диалоговом окне использовались три элемента группировки, по две радиокнопки в каждой группе.

Вариант 2

Модифицируйте копию проекта UsgCheckBox так, чтобы использовались два элемента группировки (в первом — две стандартных отмечаемых кнопки, а во втором — две отмечаемых кнопки с тремя состояниями).

Вариант 3

Модифицируйте копию проекта UsgBtnBmp так, чтобы в модальном диалоговом окне использовалась одна рисованная кнопка. С ее помощью запустите какое-либо приложение (например, Блокнот). Измените изображения, используемые кнопкой.

Вариант 4

Модифицируйте копию проекта UsgSpinEdit так, чтобы в модальном диалоговом окне использовался один спин с дружественным элементом редактирования и поясняющей надписью. Добейтесь, чтобы после закрытия диалогового окна по кнопке **OK** появлялось окно сообщения, отображающее текущее состояние вращателя.

Вариант 5

Модифицируйте копию проекта UsgSpinEdit так, чтобы в модальном диалоговом окне дополнительно использовался еще один спин с дружественным элементом редактирования и поясняющей надписью. Добейтесь правильного функционирования вращателя в диалоговом окне.

Вариант 6

Модифицируйте копию проекта UsgScroll так, чтобы в модальном диалоговом окне дополнительно использовалась еще одна полоса прокрутки. Добейтесь правильного функционирования полосы прокрутки в диалоговом окне.

Вариант 7

Модифицируйте копию проекта UsgScroll так, чтобы вместо горизонтальных полос прокрутки использовались вертикальные полосы прокрутки.

Вариант 8

Модифицируйте копию проекта UsgScroll так, чтобы в модальном диалоговом окне с помощью трех нажимаемых кнопок можно было скрывать или показывать каждую из полос прокрутки.

Вариант 9

Модифицируйте копию проекта UsgSlider так, чтобы дополнительно использовался еще один ползунок. Добейтесь правильного функционирования ползунка в окне диалога.

Вариант 10

Модифицируйте копию проекта UsgSlider так, чтобы в модальном диалоговом окне вместо горизонтальных ползунков использовались вертикальные ползунки.

П2.8. Управляющие элементы: список, комбинированный список, индикатор прогресса и таймер (гл. 10). Варианты тестов

Совет

Для сокращения затрат времени при ответе на приводимые далее тестовые задания используйте копии демонстрационных проектов UsgListBoxes, UsgComboBoxes, и UsgProgressTimer, имеющихся на компакт-диске (рассмотрены в *главе 10*).

Вариант 1

Модифицируйте копию проекта UsgListBoxes так, чтобы в модальном диалоговом окне использовались три списка. Организуйте обмен данными между новым списком и любым из ранее существовавших списков.

Вариант 2

Модифицируйте копию проекта UsgListBoxes так, чтобы в модальном диалоговом окне использовались три списка. Организуйте обмен данными между любыми парами списков.

Вариант 3

Модифицируйте копию проекта UsgComboBoxes так, чтобы в модальном диалоговом окне использовались четыре комбинированных списка. Организуйте получение данных от добавленного комбинированного списка в главное окно приложения.

Вариант 4

Модифицируйте копию проекта UsgComboBoxes так, чтобы в модальном диалоговом окне использовался только один раскрывающийся комбинированный список. Организуйте добавление в комбинированный список элемента при изменении текста в дружественном окне редактирования. Устраните возможность переполнения списка.

Вариант 5

Модифицируйте копию проекта UsgProgressTimer так, чтобы в модальном диалоговом окне использовались два индикатора прогресса, управляемые разными таймерами.

Вариант 6

Модифицируйте копию проекта UsgProgressTimer так, чтобы в модальном диалоговом окне использовался индикатор прогресса, расположенный вертикально. Любым способом увеличьте время заполнения индикатора прогресса. Обеспечьте его заполнение со сглаживанием.

П2.9. Динамически подключаемые библиотеки (гл. 11). Варианты тестов

Вариант 1

Для демонстрационного проекта WndBtn_MFC, рассмотренного в *главе 5*, создайте и протестируйте DLL расширений.

Вариант 2

Для демонстрационного проекта WndWndBtn_MFC, рассмотренного в *главе 5*, создайте и протестируйте DLL расширений.

Вариант 3

Для демонстрационного проекта UsgToolStatusBars, рассмотренного в *главе 7*, создайте и протестируйте DLL расширений.

Вариант 4

Для демонстрационного проекта UsgModalDlg, рассмотренного в *главе 8*, создайте и протестируйте DLL расширений.

Вариант 5

Для демонстрационного проекта NoModalDlgUsgEdit, рассмотренного в *главе 8*, создайте и протестируйте DLL расширений.

Вариант 6

Для демонстрационного проекта WndEditBtn, рассмотренного в *главе 8*, создайте и протестируйте DLL расширений.

Вариант 7

Для демонстрационного проекта UsgBtnBmp, рассмотренного в *главе 9*, создайте и протестируйте DLL расширений.

Вариант 8

Для демонстрационного проекта UsgCheckBox, рассмотренного в *главе 9*, создайте и протестируйте DLL расширений.

Вариант 9

Для демонстрационного проекта UsgGroupBox, рассмотренного в *главе 9*, создайте и протестируйте DLL расширений.

Вариант 10

Для демонстрационного проекта UsgScroll, рассмотренного в *главе 9*, создайте и протестируйте DLL расширений.

Вариант 11

Для демонстрационного проекта UsgSlider, рассмотренного в *главе 9*, создайте и протестируйте DLL расширений.

Вариант 12

Для демонстрационного проекта UsgSpinEdit, рассмотренного в *главе 9*, создайте и протестируйте DLL расширений.

Вариант 13

Для демонстрационного проекта UsgComboBoxes, рассмотренного в *главе 10*, создайте и протестируйте DLL расширений.

Вариант 14

Для демонстрационного проекта UsgListBoxes, рассмотренного в *главе 10*, создайте и протестируйте DLL расширений.

Вариант 15

Для демонстрационного проекта UsgProgressTimer, рассмотренного в *главе 10*, создайте и протестируйте DLL расширений.

П2.10. Создание каркаса MFC-приложения с помощью мастера и его русификация (гл. 12). Варианты тестов

Варианты 1—10

Используя MFC Application Wizard, создайте каркас приложения на базе MFC с параметрами, заданными преподавателем.

Варианты 11—20

Выполните русификацию заданных преподавателем элементов каркаса приложения на базе MFC, полученного в вариантах 1—10.

П2.11. Модификация каркаса приложения на базе MFC, полученного с помощью мастера (гл. 13). Варианты тестов

Варианты 1—10

Используя копии демонстрационных проектов MDIApp1 или SDIApp1, имеющихся на прилагаемом компакт-диске, добавьте в них управляющие элементы, заданные преподавателем, и обеспечьте их функционирование.

Варианты 2—20

В демонстрационные проекты, полученные в вариантах 1—10, добавьте русифицированные темы справки для новых управляющих элементов и продемонстрируйте их работу.

П2.12. Экзаменационное тестирование

Наряду с традиционной формой, *экзаменационное* тестирование можно проводить и в более удобной, на наш взгляд, форме — в форме *тестовых вопросов*.

На экзамене каждому студенту может быть предложена комплексная проверочная работа, содержащая пять вопросов по некоторым из перечисленных далее основных разделов курса.

- Низкоуровневое проектирование Windows-приложений.
- Проектирование оконных приложений на базе библиотеки классов MFC.
- Windows-приложения с дочерними окнами.
- Ресурсы: меню, ускорители и таблица строк. "Горячие" клавиши.

- Ресурсы: панели инструментов и всплывающие подсказки. Строка статуса.
- Ресурсы: диалоговые окна и управляющие элементы.
- Элементы управления: кнопки и элементы прокрутки.
- Элементы управления: список, комбинированный список, индикатор прогресса и таймер.
- Динамически подключаемые библиотеки.
- Создание каркаса приложения на базе MFC и его русификация.
- Модификация каркаса приложения на базе MFC, полученного с помощью мастера.

Комплексная проверочная работа рассчитана на 1 ч. 40 мин. Ответ на каждый тестовый вопрос, в зависимости от правильности и полноты, оценивается 0, 0.25, 0.5, 0.75 или 1 баллом. Таким образом, максимальная сумма баллов может достигнуть 5.

В соответствии с набранными баллами выставляются следующие экзаменационные оценки:

- *отлично* (4,25—5 баллов);
- *хорошо* (3,5—4 балла);
- *удовлетворительно* (2,5—3,25 балла);
- *неудовлетворительно* (менее 2,5 баллов).

Представим пример возможного состава комплексной экзаменационной работы:

1. Windows-приложения на основе Windows API (экзаменационные тесты к *главам 2*, *5* и *6*).
2. Приложения на базе MFC (экзаменационные тесты к *главам 4—7*).
3. Диалоговые окна и управляющие элементы (экзаменационные тесты к *главам 8—10*).
4. Динамически подключаемые библиотеки (экзаменационные тесты к *главе 11*).
5. Создание каркаса приложения на базе MFC с помощью MFC Application Wizard и его модификация (экзаменационные тесты к *главам 12—13*).

П2.13. Курсовое проектирование. Варианты заданий

На практических занятиях и в процессе самостоятельной работы студенты выполняют курсовой проект в IDE Microsoft Visual Studio 2005, целью кото-

рого является освоение технологии разработки и применения приложений на базе MFC, полученных с использованием MFC Application Wizard и других мастеров. Возможные варианты заданий для курсового проектирования приведены в табл. П2.1.

Таблица П2.1. *Возможные варианты заданий для курсового проектирования*

№ варианта (формулировки решаемых задач см. далее)	Архитектура: SDI или MDI	Добавляемые управляющие элементы (интерфейс с решаемой задачей)
1, 7, 13, 19, 25	SDI	Меню с командами
2, 8, 14, 20, 26	SDI	Отдельная панель инструментов с кнопками
3, 9, 15, 21, 27	SDI	Меню с командами и отдельная панель инструментов с кнопками
4, 10, 16, 22, 28	MDI	Меню с командами
5, 11, 17, 23, 29	MDI	Отдельная панель инструментов с кнопками
6, 12, 18, 24, 30	MDI	Меню с командами и отдельная панель инструментов с кнопками

Примечание

Используйте Microsoft Visual Studio 2005. Для всех вариантов применяйте следующие общие настройки MFC Application Wizard: выключите **Use Unicode libraries**, включите **Context-sensitive Help**, для дочернего окна (дочерних окон) в качестве базового выберите класс `CEditView`, остальные настройки — по умолчанию. Полученный с помощью MFC Application Wizard каркас приложения на базе MFC следует русифицировать (исключая русификацию тем встроенной справки), которая бы потребовала большого объема работы, интегрировать в него решение заданной задачи и дополнить встроенную справку русифицированными темами справки для добавленных интерфейсных элементов управления. Средства, сгенерированные мастером, следует сохранить.

Далее следуют формулировки задач, решение которых следует интегрировать в полученный каркас приложения на базе MFC (табл. П2.1). Программа, решающая заданную задачу, должна использовать технологию объектно-ориентированного программирования. В рамках этой программы следует спроектировать базовый и производный классы или их шаблоны и выполнить их тестирование. Для ввода-вывода следует применить средства языка C++.

П2.13.1. Обработка текстов. Варианты заданий [1]

Вариант 1

Задан исходный текст на русском языке. Длина текста — не более NL строк, длина строки — не более NS символов, длина слова — не более NW символов. Исходный текст должен заканчиваться точкой ('!', '?'). Составить программу, которая в заданном тексте убирает лишние пробелы между словами, оставляя их по одному. В файле результатов должен быть исходный и преобразованный тексты.

Вариант 2

Задан исходный текст на русском языке. Длина текста — не более NL строк, длина строки — не более NS символов, длина слова — не более NW символов. Исходный текст должен заканчиваться точкой ('!', '?'). Составить программу, которая в заданном тексте находит слово (слова) максимальной длины. В файле результатов должен быть исходный текст, значение максимальной длины слова, список найденных слов (через запятую) и их количество.

Вариант 3

Исходные данные представлены в двух файлах. Первый файл содержит словарь русских слов (прописными буквами), разделенных запятыми. После последнего слова должна стоять точка. Длина текста — не более NUMW слов, длина строки — не более NS символов, длина слова — не более NW символов. Второй файл содержит приставки (прописными буквами), допустимые правилами словообразования. Составить программу, которая в файл вывода выводит исходный текст, список существующих приставок и преобразованный текст, в котором найденные приставки отделены от остальной части слова символом '-'.

Вариант 4

Задан исходный текст на русском языке. Длина текста — не более `NL` строк, длина строки — не более `NS` символов, длина слова — не более `NW` символов. Исходный текст должен заканчиваться точкой ('!', '?'). Составить программу, которая определяет, сколько слов содержат 1 слог, 2 слога и т. д. В файле результатов должен быть исходный текст и список слов (разделенных запятыми) с 1 слогом, 2 слогами и т. д.

Вариант 5

Задан исходный текст на русском языке. Длина текста — не более `NL` строк, длина строки — не более NS символов, длина слова — не более NW символов. Исходный текст должен заканчиваться точкой ('!', '?'). После обработки исходного текста полученные слова хранить в однонаправленном линейном

не кольцевом списке. Для каждого слова хранить также число согласных букв в слове. В полученном линейном списке найти слова, в которых количество согласных превышает заданное значение. Заданное значение содержится в первой строке исходного файла. В файл результатов напечатать исходный текст (эхопечать), состояние сформированного линейного списка и найденные слова.

Вариант 6

Задан исходный текст на русском языке. Длина текста — не более NL строк, длина строки — не более NS символов, длина слова — не более NW символов. Исходный текст должен заканчиваться точкой ('!', '?'). После обработки исходного текста полученные слова хранить в однонаправленном линейном не кольцевом списке. Для каждого слова хранить также число прописных букв в нем. В полученном линейном списке найти слова, в которых количество прописных букв превышает заданное значение. Заданное значение содержится в первой строке исходного файла. В файл результатов напечатать исходный текст (эхопечать), состояние сформированного линейного списка и найденные слова.

Вариант 7

Задан исходный текст на русском языке. Длина текста — не более NL строк, длина строки — не более NS символов, длина слова — не более NW символов. Исходный текст должен заканчиваться точкой ('!', '?'). После обработки исходного текста полученные слова хранить в однонаправленном линейном не кольцевом списке. В полученном линейном списке найти слова, начинающиеся и заканчивающиеся заданной буквой. В качестве заданной буквы используйте последнюю русскую букву преобразованного исходного текста. В файл результатов напечатать исходный текст (эхопечать), состояние сформированного линейного списка и найденные слова.

Вариант 8

Задан исходный текст на русском языке. Длина текста — не более NL строк, длина строки — не более NS символов, длина слова — не более NW символов. Исходный текст должен заканчиваться точкой ('!', '?'). После обработки исходного текста полученные слова хранить в однонаправленном линейном не кольцевом списке. Полученный линейный список отсортировать. В файл результатов напечатать исходный текст (эхопечать), состояние сформированного линейного списка и отсортированного списка.

Вариант 9

Задан исходный текст на русском языке. Длина текста — не более NL строк, длина строки — не более NS символов, длина слова — не более NW симво-

лов. Исходный текст должен заканчиваться точкой ('!', '?'). После обработки исходного текста полученные слова хранить в однонаправленном линейном не кольцевом списке. В полученном линейном списке найти слова, начинающиеся с гласных букв. В файл результатов напечатать исходный текст (эхопечать), состояние сформированного линейного списка и найденные слова.

П2.13.2. Обработка массивов. Варианты заданий [1]

Вариант 10

Векторная арифметика. Элементы вектора могут быть любого типа с плавающей точкой. Реализовать перегрузку различных операций над векторами и некоторую обработку вектора. Для решения данной задачи использовать шаблон базовых классов (размещение вектора в динамической памяти и его инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение вектора значениями из файла, печать содержимого вектора в файл) и шаблон производных классов (перегрузить операции сложения, вычитания, умножения и деления векторов, оператор `[]`, поиск максимального числа, встречающегося в заданном векторе более одного раза).

Вариант 11

Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (определение нормы заданной матрицы, т. е. значения $\max_i(\sum_j |a[i][j]|)$).

Вариант 12

Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (по заданной квадратной матрице размером N×N построить вектор длиной (2×N – 1), элементы которого — максимумы элементов диагоналей, параллельных главной, включая главную диагональ).

Вариант 13

Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (характеристикой строки матрицы назовем сумму ее положительных элементов, имеющих четные значения индексов; переставляя строки заданной матрицы, расположить их в соответствии с ростом характеристик).

Вариант 14

Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (для заданной квадратной матрицы найти минимум среди сумм модулей элементов диагоналей, параллельных побочной диагонали).

Вариант 15

Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (говорят, что матрица имеет седловой элемент $a[i][j]$, если элемент $a[i][j]$ является минимальным в i-й строке и максимальным в j-м столбце; найти номера строки и столбца какого-либо седлового элемента и его значение).

Вариант 16

Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (найти значение максимального элемента матрицы среди всех элементов тех строк матрицы, которые упорядочены либо по возрастанию, либо по убыванию значений элементов).

Вариант 17

Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (характеристикой столбца матрицы назовем сумму его отрицательных элементов, имеющих нечетные значения индексов; переставляя столбцы заданной матрицы, расположить их в соответствии с убыванием характеристик).

Вариант 18

Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (элемент матрицы называется локальным минимумом, если его значение строго меньше значений всех имеющихся соседей; подсчитать количество локальных минимумов заданной матрицы).

Вариант 19

Векторная арифметика. Элементы вектора могут быть любого типа с плавающей точкой. Реализовать перегрузку различных операций над векторами и некоторую обработку вектора. Для решения данной задачи использовать шаблон базовых классов (размещение вектора в динамической памяти и его инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение вектора значениями из файла, печать содержимого вектора в файл) и шаблон производных классов (перегрузить операции сложения, вычитания, умножения и деления векторов, оператор `[]`, найти элемент вектора, имеющий максимальное значение; элементы, стоящие после максимального, заменить нулями и переставить в начало вектора).

Вариант 20

Векторная арифметика. Элементы вектора могут быть любого типа с плавающей точкой. Реализовать перегрузку различных операций над векторами и некоторую обработку вектора. Для решения данной задачи использовать шаблон базовых классов (размещение вектора в динамической памяти и его инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение вектора значениями из файла, печать содер-

жимого вектора в файл) и шаблон производных классов (перегрузить операции сложения, вычитания, умножения и деления векторов, оператор `[]`, найти максимальное значение элемента вектора среди отрицательных и минимальное значение — среди положительных элементов).

Вариант 21

Векторная арифметика. Элементы вектора могут быть любого типа с плавающей точкой. Реализовать перегрузку различных операций над векторами и некоторую обработку вектора. Для решения данной задачи использовать шаблон базовых классов (размещение вектора в динамической памяти и его инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение вектора значениями из файла, печать содержимого вектора в файл) и шаблон производных классов (перегрузить операции сложения, вычитания, умножения и деления векторов, оператор `[]`, упорядочение элементов вектора по знаку, сначала положительных, а затем — отрицательных, в таком же порядке, как в исходном векторе).

Вариант 22

Векторная арифметика. Элементы вектора могут быть любого типа с плавающей точкой. Реализовать перегрузку различных операций над векторами и некоторую обработку вектора. Для решения данной задачи использовать шаблон базовых классов (размещение вектора в динамической памяти и его инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение вектора значениями из файла, печать содержимого вектора в файл) и шаблон производных классов (перегрузить операции сложения, вычитания, умножения и деления векторов, оператор `[]`, поиск максимального элемента вектора, и если он не равен нулю, то деление на него всех элементов вектора, если же максимальный элемент вектора равен нулю, то вектор не изменяется).

Вариант 23

Векторная арифметика. Элементы вектора могут быть любого типа с плавающей точкой. Реализовать перегрузку различных операций над векторами и некоторую обработку вектора. Для решения данной задачи использовать шаблон базовых классов (размещение вектора в динамической памяти и его инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение вектора значениями из файла, печать содержимого вектора в файл) и шаблон производных классов (перегрузить операции сложения, вычитания, умножения и деления векторов, оператор `[]`, поиск элементов, встречающихся в векторе более одного раза, из найденных элементов сформировать новый вектор).

П2.13.3. Решение геометрических задач. Варианты заданий [1]

Совет

В следующих далее вариантах заданий координаты точек могут быть любого типа с плавающей точкой. Координаты точек следует хранить в динамической памяти. При вводе исходных данных из файла на магнитном диске нужно предусмотреть контроль достаточности количества исходных данных, кратности прочитанных данных двум (точки на плоскости) или трем (точки в пространстве), наличия повторяющихся координат точек и т. п. Для решения заданной задачи использовать шаблон базовых классов (считывание из файла и размещение координат точек в динамической памяти — конструктор, при необходимости — конструктор копирования, деструктор, печать координат точек в файл) и шаблон производных классов (содержательное решение задачи).

Вариант 24

Из заданного множества точек на плоскости выбрать две различные точки так, чтобы количество точек, лежащих по разные стороны прямой, проходящей через эти две точки, различались наименьшим образом.

Вариант 25

Определить радиус и центр окружности, на которой лежит наибольшее число точек заданного на плоскости множества точек.

Вариант 26

Определить радиус и центр такой окружности, проходящей хотя бы через три различные точки заданного множества точек на плоскости, что минимальна разность числа точек, лежащих внутри и вне окружности.

Вариант 27

Расстояние между двумя множествами точек — это расстояние между наиболее близко расположенными точками этих множеств. Найти расстояние между двумя заданными множествами точек на плоскости.

Вариант 28

Задано множество точек в трехмерном пространстве. Найти такую из них, что шар заданного радиуса с центром в этой точке заключает в себе максимальное число точек множества.

Вариант 29

Выбрать три разные точки заданного на плоскости множества точек, составляющие треугольник наибольшего периметра.

Вариант 30

Определить радиус и центр окружности минимального радиуса, проходящей хотя бы через три различные точки множества точек на плоскости.

Замечание

Приведем несколько определений и указаний, связанных с решением геометрических задач.

Множество точек на плоскости — это конечное, возможно пустое множество различных пар вещественных чисел, соответствующих координатам точек на плоскости. Аналогично определяется и множество точек трехмерного пространства.

1. Для построения прямой, проходящей через две различные точки P и Q с координатами (P_X, P_Y) и (Q_X, Q_Y), достаточно в уравнении прямой $A * x + B * y + C = 0$ взять $A = P_Y - Q_Y$, $B = Q_X - P_X$, $C = (P_X - Q_X) * P_Y + (Q_Y - P_Y) * P_X$.
2. Прямая, проходящая через точку P перпендикулярно отрезку PQ, имеет уравнение $B * x + A * y + D = 0$ с приведенными в п. 1 значениями A, B и $D = B * P_X - A * P_Y$.
3. Подстановка в левую часть уравнения прямой $A * x + B * y + C = 0$ координат двух точек, лежащих по одну сторону от этой прямой, дает значение одного знака.
4. Расстояние между точками P и Q с координатами (P_X, P_Y) и (Q_X, Q_Y)

$$d = \sqrt{(Q_X - P_X)^2 + (Q_Y - P_Y)^2}$$

5. Точка пересечения прямых $A_1 * x + B_1 * y + C_1 = 0$ и $A_2 * x + B_2 * y + C_2 = 0$ определяется координатами

$$y_0 = \frac{B_1 * C_2 - B_2 * C_1}{A_1 * B_2 - A_2 * B_1}; \quad x_0 = \frac{C_1 * A_2 - C_2 * A_1}{A_1 * B_2 - A_2 * B_1}$$

6. Уравнение окружности с радиусом R и центром окружности с координатами (x_0, y_0)

$$(x - x_0)^2 + (y - y_0)^2 = R^2$$

7. Алгоритм получения центра окружности, проходящей через три заданные точки — определение точки пересечения перпендикуляров к серединам (x_0, y_0) хорд.

8. Площадь треугольника со сторонами a,b,c

$$s=\sqrt{p*(p-a)*(p-b)*(p-c)},\quad p=(a+b+c)/2$$

Примечание

В качестве программы, подключаемой к каркасу полученного приложения на базе MFC, можно использовать последнюю программу, спроектированную обучаемым (студентом).

И в заключение работы над курсовым проектом следует оформить *программную документацию* в соответствии с Единой Системой Программной Документации (ЕСПД) в виде отчета. Перечислим разделы отчета (виды программных документов).

- *Техническое задание* — формулировка решаемой задачи, требования к программе, язык программирования и IDE.
- *Текст программы*. В этом разделе следует привести листинги с исходными текстами в самодокументированном виде (новые файлы и фрагменты модифицированных файлов).
- *Описание программы*. Здесь нужно привести описание интерфейса, используемого для взаимодействия с программой, интегрированной в каркас приложения на базе MFC.
- *Программа и методика испытаний*. Здесь следует привести набор контрольных примеров с его обоснованием (тестирование интерфейсных элементов — акселераторы, "горячие" клавиши, сообщения для строки статуса, всплывающие подсказки; тестирование решаемой задачи; тестирование модифицированной части справочной системы).

ПРИЛОЖЕНИЕ 3

Технология .NET. Создание и отладка оконных приложений в Microsoft Visual Studio 2005. Справочная система

Далее, в качестве введения, кратко рассматриваются особенности новой технологии .NET (читается как "дот нет"), предложенной фирмой Microsoft, основное внимание уделяется созданию и отладке Windows-приложений в IDE Microsoft Visual Studio 2005 и, в заключение, рассматривается использование встроенной справочной системы.

Microsoft .NET — это новая технология разработки программного обеспечения [9]. В ее основе лежит идея универсального программного кода, который может быть выполнен любым компьютером, вне зависимости от используемой этим компьютером ОС, с одним "но" — ОС должна поддерживать технологию .NET. Универсальность программного кода обеспечивается за счет предварительной (на этапе разработки) компиляции исходной программы в промежуточный, универсальный код, который во время загрузки (выполнения) транслируется в выполняемую программу.

Трансляцию исходной программы в промежуточный код, который называется CIL-кодом (Common Intermediate Language, универсальный промежуточный код), выполняет компилятор соответствующей среды разработки, а преобразование промежуточного кода в выполняемый код осуществляет JIT-компилятор (от Just In Time — одновременно, "на лету"), являющийся элементом виртуальной выполняющей системы (VES, Virtual Execution System).

Microsoft .NET Framework for Windows — это платформа (среда), обеспечивающая выполнение .NET-приложений в среде Windows. Ее основными элементами являются общеязыковая универсальная выполняющая система (CLR, Common Language Runtime,) и библиотека классов (.NET Framework

Classes Library). Универсальная выполняющая система обеспечивает компиляцию (именно компиляцию, а не интерпретацию) CIL-программы и выполнение полученного кода. При этом процессы компиляции и выполнения протекают одновременно с оптимизацией выполняемой программы.

Переход на платформу .NET позволяет, прежде всего, повысить производительность работы программиста. Это объясняется тем, что для программирования разных компьютеров, поддерживающих технологию .NET, он может использовать один и тот же, хорошо знакомый ему язык программирования, одни и те же объекты библиотеки классов .NET Framework Classes Library. Также он может использовать этот язык для создания приложений различного типа.

Использование технологии .NET облегчает интеграцию приложений, созданных разными программистами на разных языках программирования (исходная программа компилируется в CIL-код, вид которого не зависит от языка, на котором записана исходная программа). Технология .NET облегчает развертывание приложений, т. к. вся информация, необходимая для запуска приложения, находится в выполняемом файле, а не в реестре ОС.

По состоянию на 2006—2007 гг. компиляторы для создания .NET-приложений разрабатывались более чем для 30 различных языков [10]. Помимо четырех языков, поставляемых с Visual Studio .NET (C#, Visual Basic .NET, Managed C++, JScript .NET), ожидаются .NET-версии Delphi, Smalltalk, COBOL, Pascal, Perl и множества других известных языков.

Замечание

Более подробные сведения о технологии .NET можно найти в [10, глава 1 "Философия .NET"] и в [11, глава 1 "Общеязыковая среда исполнения"]. Хотя технология .NET и не использована в данной книге, мы сочли полезным включение в книгу кратких сведений об этой технологии.

Создание и отладка программного проекта консольного приложения в предыдущих версиях IDE Microsoft Visual Studio 6.0 и Microsoft Visual Studio 2003 рассмотрены соответственно в [2] и в [1]. Поэтому далее рассмотрим методику создания и отладки оконных приложений для последней версии IDE — Microsoft Visual Studio 2005.

Совет

Рекомендуем вам ознакомиться с отличиями последней версии IDE — Microsoft Visual Studio 2005 от предыдущей версии IDE — Microsoft Visual Studio 2003. Их описание можно найти во встроенной справочной системе (в поле **Look for:** окна **Index** задайте "Visual Studio 2005", в появившемся окне **Index Results** выберите "What's New in Visual Studio 2005 for Native Developers").

П3.1. Создание оконного приложения на основе Windows API

Рассмотрим наиболее рациональные приемы создания Windows-приложения на примере демонстрационного проекта FrameWnd из *главы 2*, использующего Windows API.

П3.1.1. Создание пустого проекта оконного приложения

Чтобы создать новое оконное приложение (программу), необходимо создать новый проект. Запустите Microsoft Visual Studio 2005. В поле **Resent Projects** вкладки **Start Pages** (если ее нет на рабочем столе, то выполните команду **View | Other Windows | Start Page**) выполните **Create:Project...** В результате на экране появится диалоговое окно **New Project**. Здесь вы должны выбрать тип создаваемого приложения — в поле **Project Types** (тип проекта) выберите **Win32**, а в поле **Templates** (шаблоны) — **Win32 Project**. Выберите месторасположение нового проекта. Информация о расположении новой рабочей области проекта (путь) вводится в поле **Location** (местоположение). Это можно сделать, набрав путь вручную, или воспользоваться расположенной справа кнопкой **Browse...** (Просмотр). Разумеется, что соответствующий подкаталог должен быть предварительно создан. Укажите в поле **Name** (Имя) имя проекта, например, FrameWnd (рис. П3.1). После выполнения указанных действий нажмите кнопку **OK**, в результате чего на экране появится диалоговое окно мастера создания приложения **Win32 Application Wizard — FrameWnd**.

Для создания пустого проекта в этом окне выберите вкладку **Application Settings**, выберите опцию **Empty Project** и нажмите кнопку **Finish** (рис. П3.2).

В окне **Solution Explorer — FrameWnd** (если его нет на рабочем столе, то выполните команду **View | Solution Explorer**) для проекта FrameWnd правой кнопкой мыши вызовите контекстное меню и выполните команду **Properties**. В поле **Character Set** вкладки **General** выберите **Use Multi-Byte Character Set** (рис. П3.3) и нажмите кнопку **OK**.

В результате указанных действий будет создан пустой проект простого низкоуровневого оконного приложения FrameWnd, настроенный на использование Windows API.

Чтобы наполнить созданный проект, необходимо добавить в него файлы, содержащие текст программы. Для проекта FrameWnd такими файлами являются файлы FrameWnd.cpp, StdAfx.cpp и StdAfx.h, рассмотренные ранее в *главе 2*.

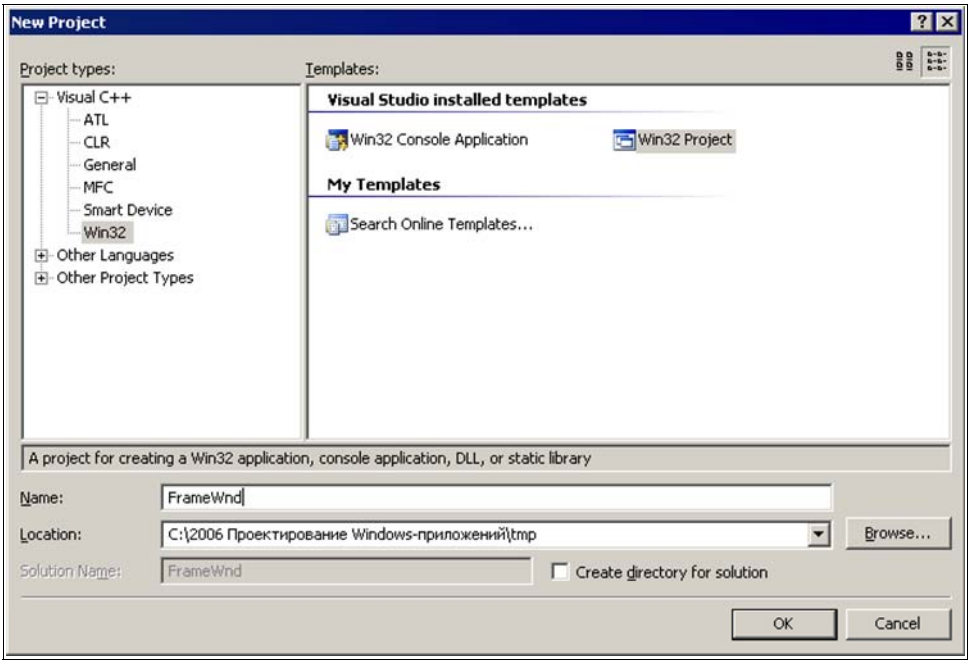


Рис. П3.1. Вид диалогового окна **New Project** при создании нового проекта оконного приложения

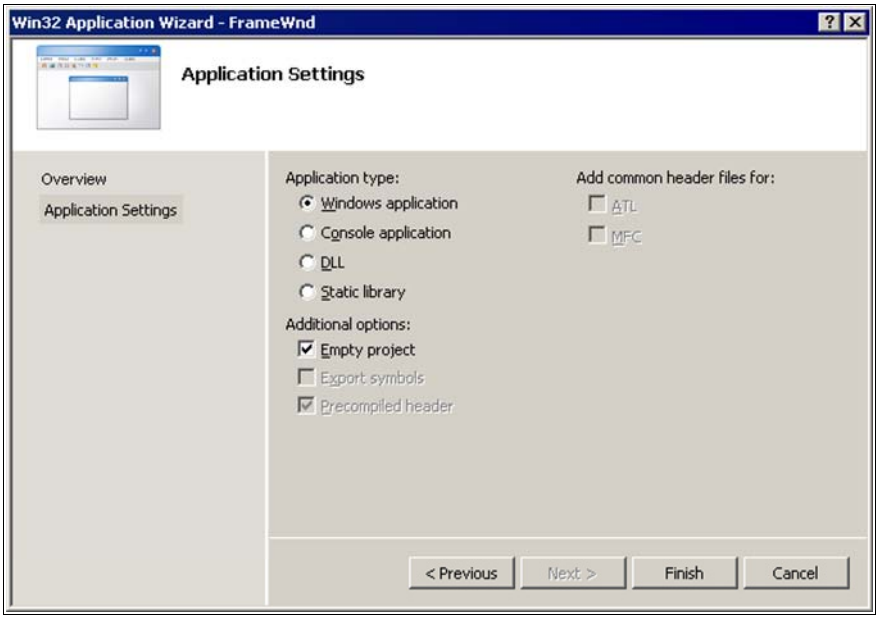


Рис. П3.2. Вид диалогового окна **Win32 Application Wizard — FrameWnd**

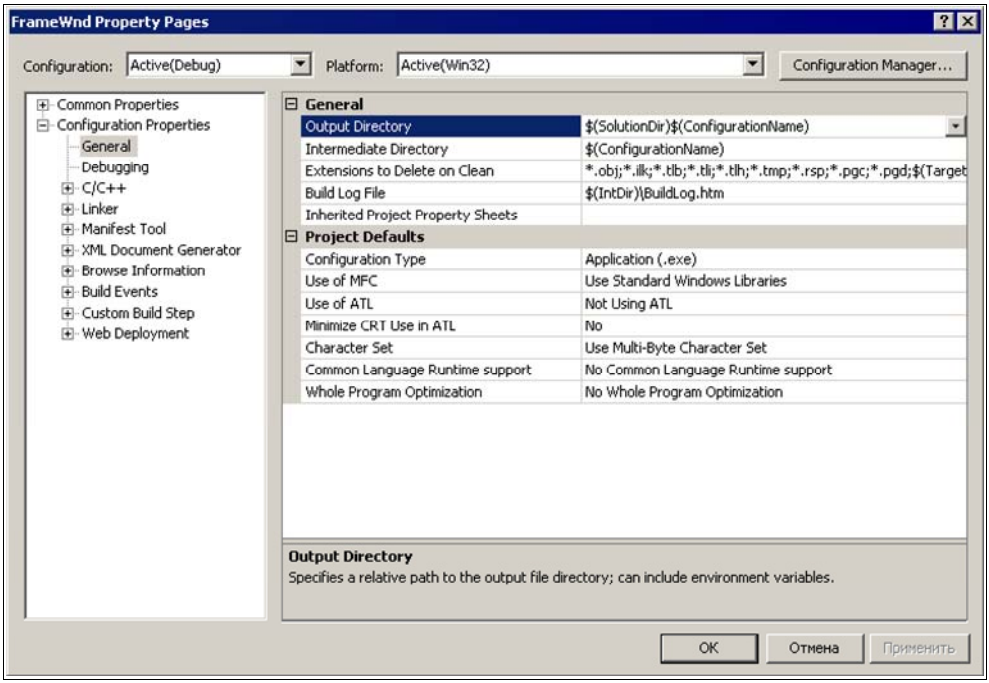


Рис. П3.3. Вид диалогового окна **FrameWnd Property Pages**

П3.1.2. Создание нового файла и включение его в проект

В окне **Solution Explorer — FrameWnd** (если его нет на рабочем столе, то выполните команду **View | Solution Explorer**) для проекта FrameWnd правой кнопкой мыши вызовите контекстное меню и выполните команду **Add | New Item...**. В появившемся окне **Add | New Item — FrameWnd** в поле **Categories** (категория) выберите **Code**, в поле **Templates** (Шаблоны) выберите **C++ File (.cpp)**, в поле **Name** (Имя) задайте имя файла FrameWnd (рис. П3.4) и нажмите кнопку **Add**.

В появившемся окне редактирования файла FrameWnd.cpp введите исходный текст файла в соответствии с листингами 2.1—2.5 и, с помощью команды **File | Save FrameWnd.cpp**, сохраните его. Аналогичным образом добавьте в проект пустые файлы StdAfx.cpp и StdAfx.h. В добавленные пустые файлы введите исходные тексты в соответствии с листингами 2.6—2.7 и также сохраните их.

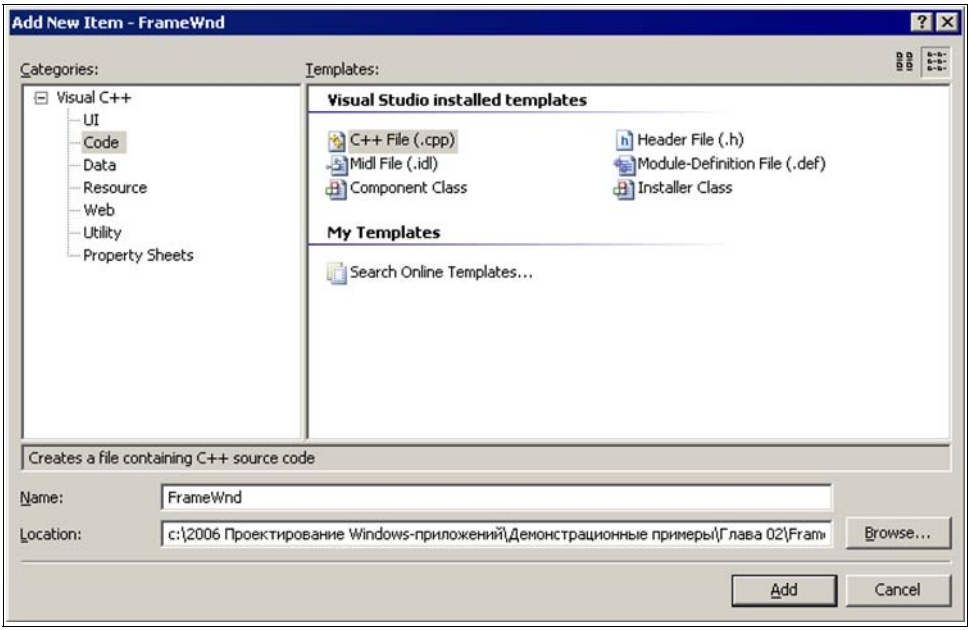


Рис. П3.4. Добавление в проект FrameWnd файла FrameWnd.cpp

П3.1.3. Добавление в проект существующего файла

Добавляемые в проект существующие файлы целесообразно скопировать в папку проекта. Для созданного проекта FrameWnd такие файлы (FrameWnd.cpp, StdAfx.cpp и StdAfx.h) можно взять из одноименного проекта на компакт-диске, использующего Windows API. Для добавления существующего файла выберите в окне **Solution Explorer — FrameWnd** проект FrameWnd и для него правой кнопкой мыши вызовите контекстное меню. В контекстном меню выполните команду **Add | Add Existing Item...**, в появившемся окне **Add Existing Item — FrameWnd** выберите файл для включения в проект (рис. П3.5) и нажмите кнопку **Add**.

Таким же образом можно включить в проект все необходимые существующие файлы (для проекта FrameWnd такими файлами являются StdAfx.cpp и StdAfx.h).

П3.1.4. Открытие существующего проекта

Чтобы открыть для работы существующий проект, запустите ИСР Microsoft Visual Studio 2005. В поле **Resent Projects** вкладки **Start Pages** (если ее нет на

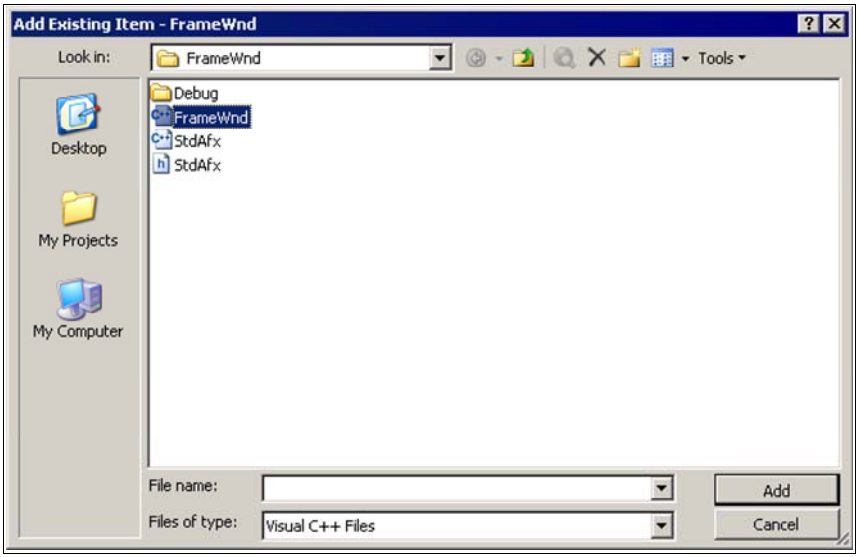


Рис. П3.5. Добавление в проект FrameWnd существующего файла FrameWnd.cpp

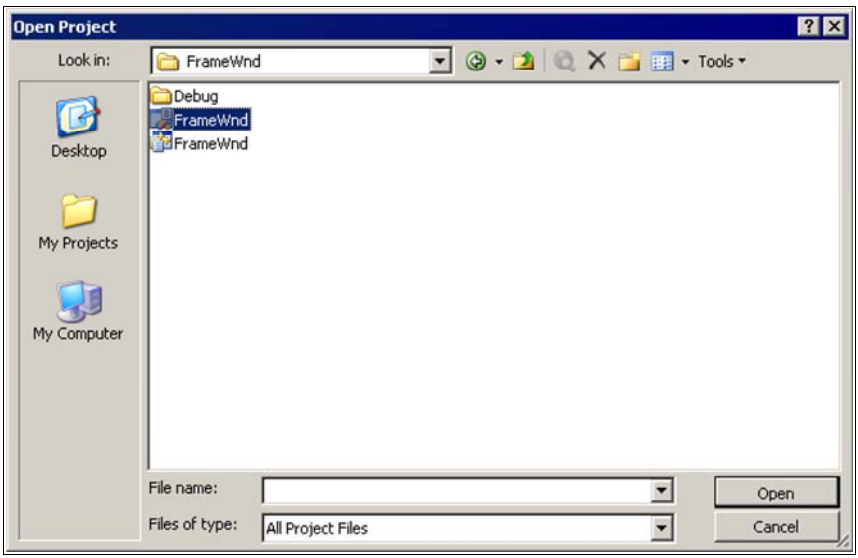


Рис. П3.6. Открытие для работы существующего проекта

рабочем столе, то выполните команду **View | Other Windows | Start Page**) выполните **Open:Project...**. В результате на экране появится диалоговое окно **Open Project** (рис. П3.6). Войдите в каталог проекта и "кликните" по файлу с расширением sln (от SoLutioN). В результате проект загрузится в IDE для последующей работы.

Замечание

Для работы в Microsoft Visual Studio 2005 можно открыть любой существующий проект более ранней версии: Microsoft Visual Studio C++ 6.0 или Microsoft Visual Studio 2003. При этом он будет конвертирован для использования в Microsoft Visual Studio 2005. Имейте в виду, что в версии Microsoft Visual Studio C++ 6.0 вместо файла с расширением sln следует выбрать файл с расширением dsw.

П3.1.5. Прекомпиляция стандартных заголовочных файлов

Целесообразность прекомпиляции и оформление стандартных заголовочных файлов, обеспечивающие возможность прекомпиляции, рассмотрены ранее в *разд. 2.7*. В проекте FrameWnd такими файлами являются StdAfx.cpp и StdAfx.h. Чтобы обеспечить в проекте прекомпиляцию стандартных заголовочных файлов, выполните следующее. В окне **Solution Explorer — FrameWnd** для проекта FrameWnd правой кнопкой мыши вызовите контекстное меню и выполните команду **Properties**. В появившемся окне **FrameWnd Property Pages** откройте вкладку **C/C++**, выберите категорию **Precompiled Header** и в поле **Create/Use Precompiled Headers** выберите **Use Precompiled Header (/Yu)** и нажмите кнопку **OK** (рис. П3.7).

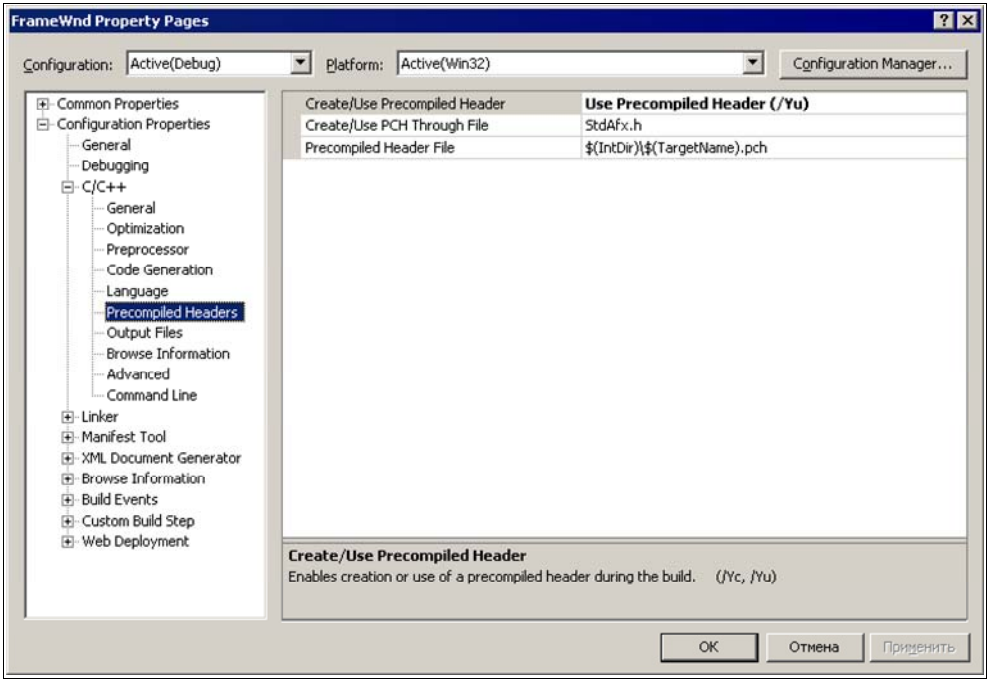


Рис. П3.7. Опции файлов проекта для прекомпиляции стандартных заголовочных файлов

Аналогично, в окне **Solution Explorer — FrameWnd** для файла StdAfx.cpp правой кнопкой мыши вызовите контекстное меню и выполните команду **Properties**. В появившемся окне **StdAfx.cpp Property Pages** откройте вкладку **C/C++**, выберите категорию **Precompiled Headers** и в поле **Create/Use Precompiled Header** выберите **Create Precompiled Header (/Yc)** и нажмите кнопку **OK** (рис. П3.8).

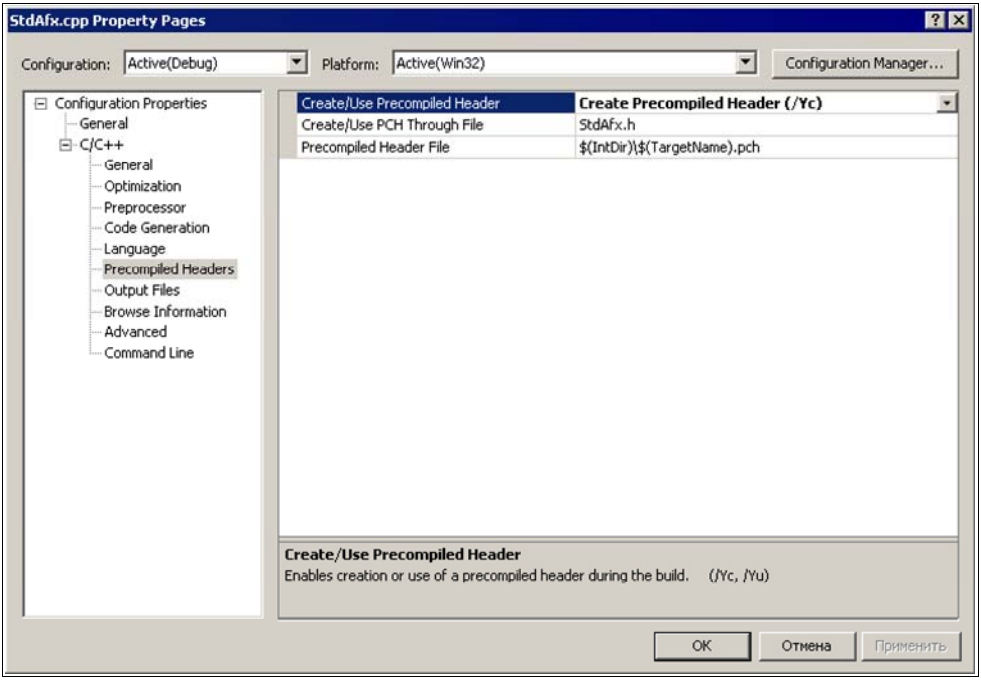


Рис. П3.8. Опции файла StdAfx.cpp для прекомпиляции стандартных заголовочных файлов

Примечание

После создания проекта или запуска существующего проекта можно выполнить последовательную компиляцию файлов с расширением cpp (в проекте FrameWnd такими файлами являются StdAfx.cpp и FrameWnd.cpp), командой **Build** построить проект и запустить его командой **Start Without Debugging**.

П3.2. Создание оконного приложения на основе библиотеки классов MFC

Рассмотрим, как и в предыдущем случае, наиболее рациональные приемы создания Windows-приложения на примере проекта FrameWnd из *главы 4*, использующего библиотеку классов MFC.

П3.2.1. Создание пустого проекта оконного приложения

Пустой проект оконного приложения создается аналогично указанному в *разд. П3.1.1* с единственным изменением, относящимся к последнему этапу.

В окне **Solution Explorer — FrameWnd** (если его нет на рабочем столе, то выполните команду **View | Solution Explorer**) для проекта FrameWnd правой кнопкой мыши вызовите контекстное меню и выполните команду **Properties**. В поле **Character Set** вкладки **General** выберите **Use Multi-Byte Character Set**, в поле **Use of MFC** выберите **Use MFC in a Shared DLL** (рис. П3.9) и нажмите кнопку **OK**.

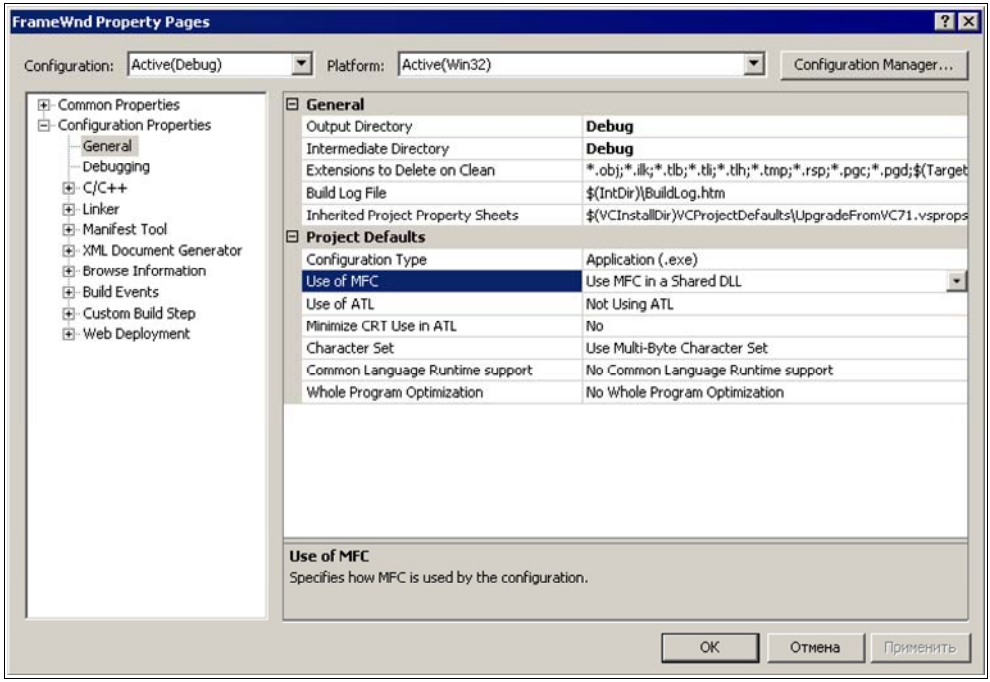


Рис. П3.9. Вид диалогового окна **FrameWnd Property Pages**

В результате указанных действий будет создан пустой проект простого оконного приложения FrameWnd, настроенный на использование библиотеки классов MFC.

Чтобы наполнить созданный проект, необходимо добавить в него файлы, содержащие текст программы. Это целесообразно сделать с максимальным использованием имеющихся мастеров оконных приложений, что и будет продемонстрировано далее.

П3.2.2. Добавление в проект оконного приложения необходимых классов и методов с использованием мастеров

В соответствии со сказанным ранее в *главе 4*, в проект нужно добавить два класса, производных соответственно от классов `CWinApp` и `CFrameWnd` библиотеки MFC.

Для добавления в проект класса `MainThread`, производного от класса `CWinApp` библиотеки классов MFC и предназначенного для инициализации и запуска оконного приложения, следует выполнить команду **Project | Add Class...**. В появившемся окне диалога нужно выбрать разновидность вставляемого класса (рис. П3.10), нажать кнопку **Add** и сконфигурировать класс (рис. П3.11).

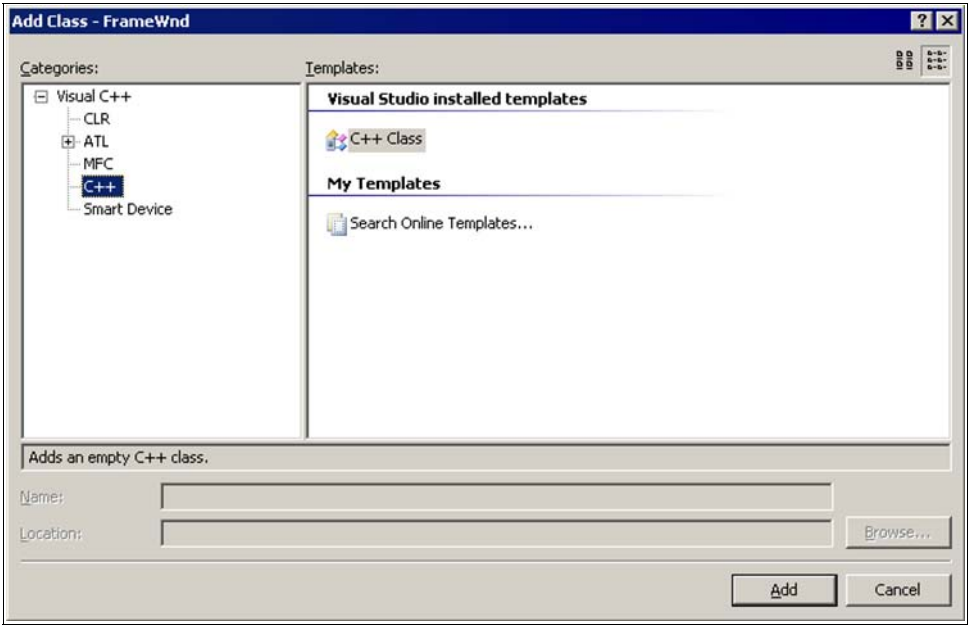


Рис. П3.10. Выбор класса для добавления в проект

После нажатия кнопки **Finish** появится окно сообщения **Microsoft Visual Studio** (рис. П3.12), в котором следует нажать кнопку **Да**, и в проект добавляются файлы MainThread.hpp и MainThread.cpp, содержащие объявление и реализацию класса `MainThread`.

Класс `MainThread` добавляется в проект для того, чтобы переопределить в нем виртуальный метод `InitInstance`, который будет автоматически вызван при создании объекта типа `MainThread` (в базовом классе `CWinApp` виртуальный

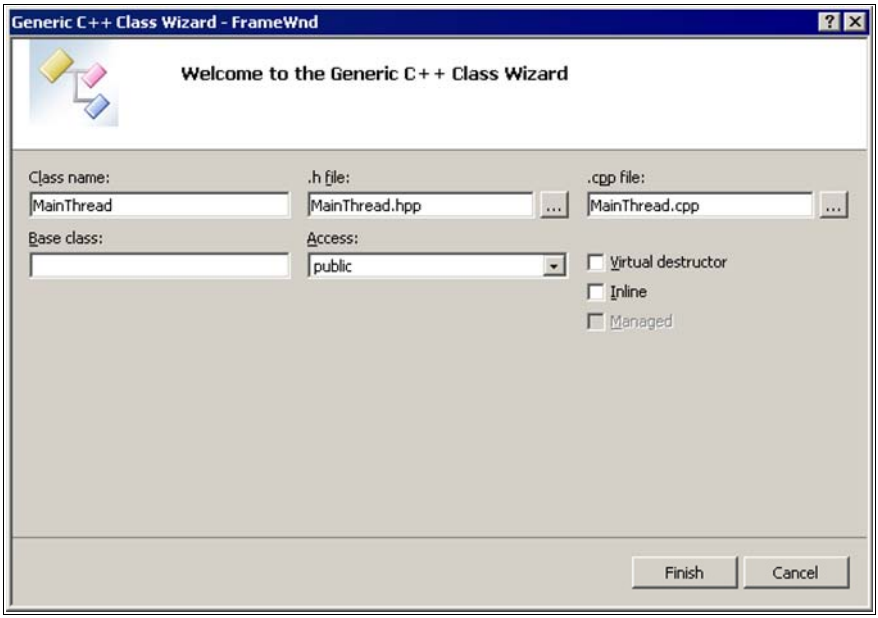


Рис. П3.11. Конфигурирование выбранного класса

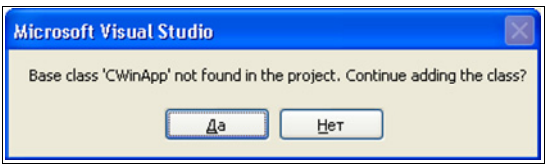


Рис. П3.12. Окно сообщения

метод `InitInstance` является просто "заглушкой" — в его блоке содержится единственный оператор возврата). Для добавления в класс `MainThread` виртуального метода `InitInstance` следует перейти в окно **Class View** и выполнить, для класса `MainThread`, команду **Add | Add Function...** контекстного меню. В появившемся окне **Add Member Function Wizard — FrameWnd** сконфигурируйте добавляемую функцию — метод класса (рис. П3.13) и нажмите кнопку **Finish**. В результате файлы MainThread.hpp и MainThread.cpp приобретут вид, представленный в листингах П3.1 и П3.2 соответственно.

Листинг П3.1. Файл MainThread.hpp

```
#pragma once

class MainThread :
    public CWinApp
```

```
{
public:
    MainThread(void);
public:
    ~MainThread(void);
public:
    virtual BOOL InitInstance(void);
};
```

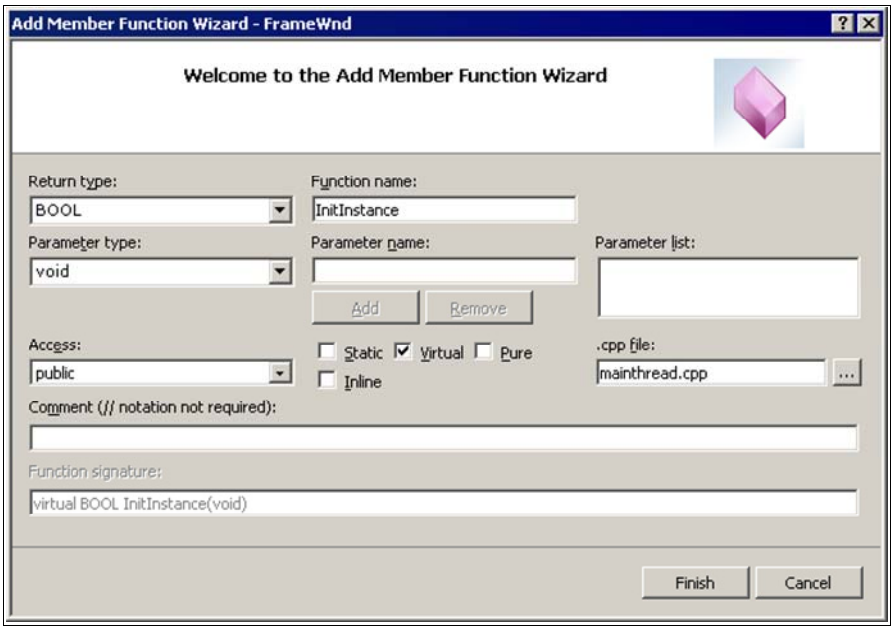


Рис. П3.13. Конфигурирование добавляемого метода класса

Листинг П3.2. Файл MainThread.cpp

```
#include "MainThread.hpp"

MainThread::MainThread(void)
{
}

MainThread::~MainThread(void)
{
}
```

```
BOOL MainThread::InitInstance(void)
{
    return 0;
}
```

Аналогичным образом в проект добавляется класс `FrameWnd`, производный от класса `CFrameWnd` библиотеки классов MFC, и в проект включаются еще два файла — FrameWnd.cpp и FrameWnd.hpp. Основным назначением класса `FrameWnd` является обработка сообщений главного окна. Единственным сообщением такого рода в нашем примере является сообщение `WM_PAINT`. Для добавления в класс `FrameWnd` функции-обработчика этого сообщения, как и ранее, следует перейти в окно **Class View** и выполнить, для класса `FrameWnd`, команду **Add | Add Function...** контекстного меню. В появившемся окне **Add Member Function Wizard — FrameWnd** сконфигурируйте добавляемую функцию-обработчик сообщения (рис. П3.14) и нажмите кнопку **Finish**. В результате файлы FrameWnd.hpp и FrameWnd.cpp приобретут вид, представленный в листингах П3.3 и П3.4 соответственно.

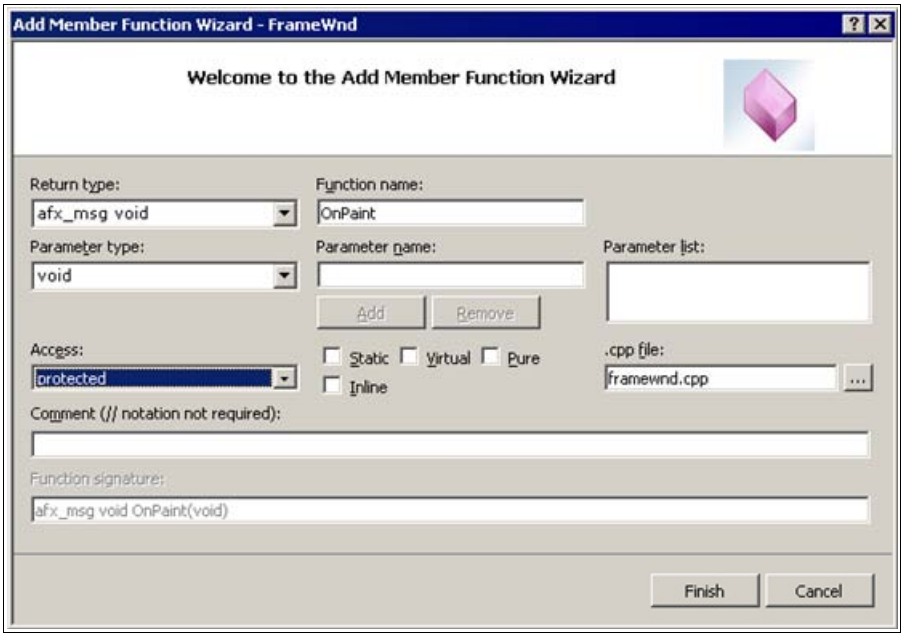


Рис. П3.14. Конфигурирование добавляемой функции, выполняющей обработку сообщения `WM_PAINT`

Листинг П3.3. Файл FrameWnd.hpp

```
#pragma once

class FrameWnd :
    public CFrameWnd
{
public:
    FrameWnd(void);
public:
    ~FrameWnd(void);
protected:
    afx_msg void OnPaint(void);
};
```

Листинг П3.4. Файл FrameWnd.cpp

```
#include "FrameWnd.hpp"

FrameWnd::FrameWnd(void)
{
}

FrameWnd::~FrameWnd(void)
{
}

afx_msg void FrameWnd::OnPaint(void)
{
    return afx_msg void();
}
```

П3.2.3. Добавление в проект объекта типа *MainThread* и модификация файлов, добавленных в каркас приложения

Добавление в проект файла Main.cpp, в котором создается объект типа `MainThread`, выполняется аналогично указанному в *разд. П3.1.2*. Полный исходный код этого файла приведен в листинге 4.5.

При создании объекта класса `CWinApp` или производного от него класса (в нашем случае класса `MainThread`) происходит обращение к стартовой функции `_WinMainCRTStartup` библиотеки времени выполнения. В числе прочих опера-

ций, выполняемых этой функцией, имеется и вызов функции `WinMain` библиотеки классов MFC. Эта функция, как и обычная функция `WinMain`, выполняет следующее:

- инициализацию первого экземпляра приложения (метод `InitApplication`);
- инициализацию текущего экземпляра приложения (виртуальный метод `InitInstance` класса `CWinApp` или производного от него класса);
- запуск цикла обработки сообщений (виртуальный метод `Run` класса `CWinApp`);
- корректное завершение работы приложения (виртуальный метод `ExitInstance` класса `CWinApp`).

Модификация, в соответствии с листингами 4.1—4.4, ранее добавленных в проект с помощью мастеров, файлов MainThread.hpp, MainThread.cpp, FrameWnd.hpp и FrameWnd.cpp, преследует две цели. Во-первых, наполнить "скелеты" этих файлов *конкретным содержанием* и, во-вторых, продемонстрировать *профессиональное оформление* исходных текстов.

Примечание

Особенностью оформления исходных текстов проекта оконного приложения является включение в них, а не в текст учебного пособия, важной информации в виде комментариев. Это делает возможным при работе с проектом получать ответы на возникающие вопросы более оперативно, не прибегая к чтению текста учебного пособия.

П3.2.4. Прекомпиляция стандартных заголовочных файлов

Прекомпиляция стандартных заголовочных файлов приложений использует файлы StdAfx.h, StdAfx.cpp и имеет следующие отличия:

- в прекомпилируемый файл StdAfx.h включаются другие стандартные заголовочные файлы, отличные от файла Windows.h, используемого в низкоуровневых Windows-приложениях (см. листинг 4.6);
- прекомпилируемый файл StdAfx.h включается во все файлы MFC-проекта с расширениями cpp, а не в файлы с расширениями h или hpp.

Файл StdAfx.cpp соответствует листингу 2.7. Опции, задающие прекомпиляцию, задавайте следующим образом. В окне **Solution Explorer** для проекта правой кнопкой мыши вызовите контекстное меню и выполните команду **Properties**. В появившемся окне **Property Pages** откройте вкладку **C/C++**, выберите категорию **Precompiled Headers** и в поле **Create/Use Precompiled Header** выберите **Use Precompiled Header (/Yu)** и нажмите кнопку **OK**. Ана-

логично в окне **Solution Explorer** для файла StdAfx.cpp правой кнопкой мыши вызовите контекстное меню и выполните команду **Properties**. В появившемся окне **StdAfx.cpp Property Pages** откройте вкладку **C/C++**, выберите категорию **Precompiled Headers** и в поле **Create/Use Precompiled Header** выберите **Create Precompiled Header (/Yc)** и нажмите кнопку **OK**.

Совет

Для ускорения отладки и компиляции используйте прекомпиляцию стандартных заголовочных файлов.

Замечание

Программный проект FrameWnd_PCH простого приложения на базе MFC с прекомпиляцией имеется на прилагаемом компакт-диске, а пример настройки проекта для прекомпиляции можно посмотреть в *разд. 4.5*.

П3.2.5. Об использовании файлов созданного проекта оконного приложения FrameWnd_PCH

Вид окна **Solution Explorer** для проекта FrameWnd_PCH с прекомпиляцией показан на рис. П3.15.

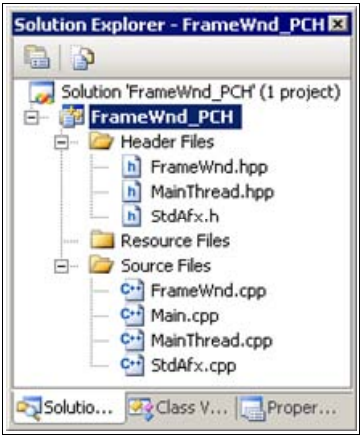


Рис. П3.15. Вид окна **Solution Explorer** для проекта FrameWnd_PCH с прекомпиляцией

Совет

Все файлы проекта используйте в проектах других Windows-приложений в качестве его "скелета" — файлы StdAfx.h, StdAfx.cpp, Main.cpp и MainThread.hpp *без изменений*, а файлы MainThread.cpp, FrameWnd.hpp и FrameWnd.h с *незначительной модификацией*.

П3.2.6. Использование других мастеров при создании приложений на базе MFC

Наряду с рассмотренными ранее *(разд. П3.2.2)* мастерами, при создании приложений можно использовать мастер добавления в класс переменной и мастер обработки событий для управляющих элементов.

Для *добавления в класс* проекта переменной или функции (метода) при модификации файлов пользуйтесь соответствующими мастерами (см. рис. 5.5 и 5.6).

Для добавления в соответствующий класс проекта таблицы сообщений, прототипов и определений функций, обрабатывающих команды меню или элементы других ресурсов, при модификации файлов с определением класса пользуйтесь *мастером обработки событий*. Для запуска мастера откройте вкладку **Resource View**, раскройте ресурс меню и загрузите в окно редактирования файл ресурса Resource.rc. Для соответствующей команды вызовите контекстное меню, выполните команду **Add Event Handler...**, добавьте в соответствующий класс таблицу сообщений и обработчик команды (см. рис. 6.12) и нажмите кнопку **Add and Edit**. См. подробнее пример использования мастера обработки событий в *главе 6*.

П3.3. Основы работы с редактором ресурсов

В начале работы с редактором ресурсов, при необходимости, следует задать *русификацию добавляемых ресурсов*. С этой целью, на вкладке **Solution Explorer** вызовите контекстное меню правой кнопкой мыши, выполните в нем команду **Properties**, выберите вкладку **Resourses** и в поле **Culture** задайте язык **Русский (0x419)**. Обычно такая настройка редактора ресурсов выбрана по умолчанию.

П3.3.1. Добавление в проект ресурсов и их настройка

Для добавления в проект необходимых ресурсов перейдите во вкладку **Solution Explorer**, вызовите для проекта правой кнопкой мыши контекстное меню и выполните в нем команду **Add | Resource...**. В появившемся окне **Add Resource** (см. рис. 6.1) последовательно выберите нужные ресурсы, каждый раз нажимая кнопку **New**.

Во вкладке **Resource View** последовательно выберите ресурсы и настройте их. Примеры настройки ресурсов **Accelerator** и **Menu** приведены ранее в *главе 6* (см. рис. 6.4—6.5), ресурса **Toolbar** — в *главе 7* (см. рис. 7.4) и ресурса **Dialog** — в *главе 8* (см. рис. 8.7—8.14).

П3.3.2. Создание и редактирование меню и команд

Для *редактирования меню* выберите вкладку **Resource View**, откройте все ресурсы и щелкните мышью по ресурсу меню, обозначенному соответствующим идентификатором. В результате этого ресурс меню загрузится в окно редактирования (см. демонстрационный пример UsgMenu из *главы 6*, рис. 6.6), посредством встроенного редактора ресурсов его можно модифицировать. С помощью левой кнопки мыши можно выбрать любое меню — оно будет отмечено рамкой. Для выбранного меню появится выпадающее подменю, если оно есть. Выбранное меню с помощью левой кнопки мыши можно также переместить в строке меню на новое место. С помощью правой кнопки мыши для любого меню или команды меню можно вызвать всплывающее контекстное меню и воспользоваться им. Наконец, дважды щелкнув левой кнопкой мыши по меню, можно посмотреть параметры меню или изменить их. В *главе 6* приведен пример такого рода. Так, если это проделать для меню **Файл**, то появится окно **Properties**, показанное на рис. 6.7. Оно демонстрирует *типовую настройку* меню, расположенных в строке меню.

Для меню **Файл** в свойстве **Caption** (см. рис. 6.7) символ &, предшествующий букве Ф, делает эту букву в названии меню **Файл** активной (она будет подчеркнута). Отметим еще несколько важных свойств меню. Свойство **Enabled** должно иметь значение **True**, иначе меню будет выключено (недоступно). Свойство **Popup** обычно имеет значение **True** — это означает, что при активизации меню появится всплывающее окно, содержащее команды меню. Если это свойство имеет противоположное значение, то меню настраивается и используется подобно команде (об этом будет сказано далее). Свойство **Help** обычно имеет значение **False**. Это означает, что меню будут располагаться так, как это показано на рис. 6.7. В противном случае соответствующее меню и все меню справа от него в строке меню будут прижаты к правой границе.

Чтобы посмотреть или изменить настройку какой-либо команды меню, например команды **Сплошной** меню **Стиль карандаша** из демонстрационного примера UsgMenu *главы 6*, достаточно выбрать это меню и в появившемся окне дважды щелкнуть левой кнопкой мыши по команде **Сплошной** (см. рис. 6.8).

Свойства **Caption**, **Popup** и **Help** обсуждались ранее. Свойство **Enabled** не имеет значения для команды меню. Свойство **ID** задает идентификатор команды, который будет использован при программной обработке действия команды. Свойство **Separator** обычно имеет значение **False**. В противном случае соответствующая команда будет отображаться в виде разделительной линии. Свойства остальных команд меню аналогичны. Для программной поддержки в исходном коде используются идентификаторы команд.

Замечание

Обратите внимание на два момента. Во-первых, в конце названия команд меню в указанном ранее примере для справки включена информация об акселераторе (об этом пойдет речь далее). Во-вторых, все доступные стандартные идентификаторы и созданные к этому времени нестандартные идентификаторы можно посмотреть и выбрать для использования в качестве значения свойства ID (см. рис. 6.8) с помощью кнопки, расположенной в его правой части. Кнопка появляется автоматически при активизации поля значения не только свойства ID, но и многих других свойств. Это очень удобно.

Для *создания нового меню* достаточно дважды щелкнуть по пустому меню, расположенному рядом с последним заданным меню. В появившемся окне **Properties** можно задать параметры меню. Аналогично, для *создания новой команды меню* достаточно дважды щелкнуть по пустому полю команды меню, расположенному внизу. В появившемся окне настройки можно задать параметры команды меню.

Совет

Меню в строке меню или *команды меню* можно *расположить в любом нужном порядке* путем их "перетаскивания" с помощью левой кнопки мыши. Проверьте эту возможность в процессе экспериментов с копией проекта.

П3.3.3. Создание и редактирование акселераторов

Рассмотрим *акселераторы* на примере демонстрационного проекта UsgMenu из *главы 6*. С этой целью, на вкладке **Resource View — UsgMenu** щелкните мышью по ресурсу акселераторов, обозначенному идентификатором `IDR_MAINFRAME`. В результате этого ресурс акселераторов загрузится в окно редактирования (см. рис. 6.9) и с помощью встроенного редактора ресурсов его можно модифицировать. Левой кнопкой мыши можно выбрать любой из акселераторов — он будет подсвечен. Чтобы посмотреть и изменить настройки любого акселератора, например акселератора с идентификатором `ID_PENSTYLE_SOLID`, достаточно выбрать и дважды щелкнуть по нему левой кнопкой мыши. На рис. 6.10 показана типичная настройка для акселератора. Обратите внимание на то, какой идентификатор использует данный акселератор — это `ID_PENSTYLE_SOLID`, который уже использовался для команды **Стиль карандаша | Сплошной Alt+D**. Это означает, что при нажатии клавиатурной комбинации <Alt>+<D> будет использована та же обработка, что и при выборе команды **Стиль карандаша | Сплошной Alt+D**. Обратите также внимание на то, что все доступные стандартные и созданные нестандартные идентификаторы можно посмотреть и выбрать, для использования, в окне **ID** с помощью кнопки, расположенной в его правой — это удобно. Аналогичную настройку имеют остальные акселераторы.

Совет

Задать свойства акселератора можно и непосредственно в окне редактирования — в полях строки соответствующего акселератора.

Примечание

Акселераторы могут быть *типа* **VIRTKEY** или **ASCII**. В первом случае можно использовать любые комбинации модификаторов — пустой, <Alt>, <Ctrl>, <Shift>, <Ctrl>+<Alt>, <Ctrl>+<Shift>, <Alt>+<Shift> и <Ctrl>+<Alt>+<Shift>. Во втором случае модификатор может отсутствовать или использоваться единственный модификатор <Alt>.

Для создания нового акселератора достаточно, в окне редактирования, дважды щелкнуть по пустому нижнему полю. В появившемся окне **Properties** можно задать параметры акселератора, аналогично тому, как это было показано на рис. 6.10.

П3.3.4. Создание и настройка панелей инструментов и кнопок

Рассмотрим *панели инструментов* с кнопками на примере демонстрационного проекта UsgToolStatusBars из *главы 7*.

Для *включения* панели инструментов в проект проще всего выбрать вкладку **Resource View**, выбрать проект UsgToolStatusBars, выполнить команду меню **Project | Add Resource...** и в появившемся окне (см. рис. 6.1) выбрать тип ресурса **Toolbar** и нажать кнопку **New**. В результате этого в проект включается ресурс панели инструментов.

Для настройки панели инструментов надо выбрать идентификатор панели инструментов, открыть вкладку **Properties** и задать параметры панели инструментов в соответствии с рис. 7.4.

Для *включения кнопки в созданную панель инструментов* достаточно дважды "щелкнуть" левой кнопкой мыши по ресурсу панели `IDR_MAINFRAME` и в окне редактирования появится ресурс панели инструментов. Далее следует дважды "щелкнуть" левой кнопкой мыши по пустой кнопке. В результате появится окно настройки параметров кнопки (см. рис. 7.5). После *задания параметров* кнопки можно "нарисовать" пиктограмму кнопки, ресурс панели инструментов приобретет вид, показанный на рис. 7.3.

На рис. 7.5 следует обратить внимание на два момента. Во-первых, идентификатор кнопки совпадает с идентификатором команды **Файл | Выход**. Поэтому выполнение этой команды и нажатие кнопки на панели инструментов приведут к одинаковому результату — работа приложения будет завершена. Во-вторых, в поле **Prompt** задается текст, выводимый в строку статуса (он

предшествует символу \n) и текст всплывающей подсказки (он следует за символом \n).

Примечание

С помощью редактора ресурсов можно легко изменить взаимное расположение кнопок панели инструментов. Для этого следует левой кнопкой мыши "перетащить" соответствующие кнопки на нужное место. Группы взаимосвязанных кнопок панели инструментов можно разделить сепараторами (разделяющими линиями). Для достижения указанной цели необходимо кнопки, которые нужно разделить сепаратором, с помощью левой кнопки мыши сначала надвинуть друг на друга, а потом раздвинуть. И наконец, существующую кнопку можно удалить с панели инструментов путем ее перемещения левой кнопкой мыши за пределы панели инструментов.

П3.3.5. Создание и настройка шаблона ресурсов диалогового окна

Рассмотрим создание и редактирование шаблона ресурсов диалогового окна на примере демонстрационного проекта UsgModalDlg из *главы 8*.

Для создания шаблона ресурсов достаточно на вкладке **Resource View** выбрать проект UsgModalDlg, выполнить команду **Project | Add Resource...**, в появившемся окне **Add Resource** выбрать ресурс **Dialog** и нажать кнопку **New**. В результате создается шаблон ресурсов диалогового окна с двумя кнопками — **OK** и **Cancel**. Размеры окна, местоположение и размеры кнопок можно легко изменить с помощью мыши, сделав их, например, такими, как показано на рис. 8.6.

В результате на вкладке **Resource View** появится ресурс **Dialog** с некоторым идентификатором, заданным редактором ресурсов по умолчанию. Если для данного идентификатора двойным "щелчком" левой кнопки мыши загрузить ресурс в окно редактирования, а затем с помощью правой кнопки мыши вызвать контекстное меню и выполнить команду **Properies**, то можно задать требуемые свойства ресурса. Для демонстрационного проекта указанные свойства показаны на рис. 8.7.

В шаблонах могут быть использованы различные элементы управления (см. рис. 8.2). Вот некоторые из них:

- рисунок (Picture Control);
- статический текст (Static Text);
- элемент редактирования (Edit Control);
- блок группы (Group Box), нажимаемая кнопка (Button);
- отмечаемая кнопка (Check Box);

- радиокнопка (Radio Button);
- комбинированный список (Combo Box);
- список (List Box);
- горизонтальная (Horizontal Scroll Bar) и вертикальная (Vertical Scroll Bar) полосы прокрутки;
- спин (или "вращатель") (Spin Control);
- индикатор прогресса (Progress Control);
- бегунок (Slider Control);
- аниматор (Animation Control).

Для размещения нужных управляющих элементов в диалоговом окне можно воспользоваться панелью управляющих элементов (**Toolbox**) (см. рис. 8.2). В демонстрационном примере такими управляющими элементами являются три радиокнопки, с помощью которых задается стиль линии — сплошной, штриховой или пунктирный. Отличительным признаком радиокнопок является то, что *активной является только одна кнопка* из группы. Для включения их в диалоговое окно достаточно последовательно на панели **Toolbox**, с помощью мыши, выбрать радиокнопку (Radio Button) и перетащить ее в нужное место диалогового окна. Размеры и названия радиокнопок также можно легко изменить желаемым образом. Для демонстрационного примера их размеры и названия имеют вид, показанный на рис. 8.4.

Для настройки свойств радиокнопки достаточно вызвать ее контекстное меню, выполнить команду **Properties** и задать свойства радиокнопки. Свойства радиокнопок для демонстрационного примера показаны на рис. 8.8.

Полученный шаблон хранится как составная часть в файле проекта Resource.rc, а идентификаторы окна и его управляющих элементов — в файле resource.h. Напоминаем, что эти файлы не стоит редактировать непосредственно, вручную — пользуйтесь редактором ресурсов.

П3.4. Некоторые особенности IDE

Рассмотрим особенности работы в IDE с вкладками (окнами) **Object Browser**, **Resource View** и **Properties**.

Вкладка **Object Browser** (ее можно сделать видимой командой **View | Object Browser**) является удобным средством работы с исходным текстом программного проекта, позволяющим наглядно представить структуру проекта в целом, классов проекта и их состав.

Вид вкладки, для программного проекта WndBtn_MFC из *главы 5*, после выбора глобального объекта `MainThread` показан на рис. 5.7 (обратите внимание

на настройку вкладки). Из рисунка видно, что проект использует два пользовательских класса: `MainThread` (производный от `CWinApp`), `FrameWnd` (производный от класса `CFrameWnd`) и один глобальный объект `MainThread` типа `MainThread`. Класс `MainThread` содержит единственный метод `InitInstance` со спецификатором доступа `public` (см. рис. 5.8). Класс `FrameWnd` содержит переменные `m_pBtnAbout` и `m_pBtnExit` — члены класса со спецификатором доступа `private`, методы `CreateChildControls`, `ButtonFrame`, `~ButtonFrame` со спецификатором доступа `public`, а также методы `OnBtnAboutClick`, `OnBtnExitClick` и `OnPaint` со спецификатором доступа `protected` (см. рис. 5.9). Обратите внимание, что в правом окне вкладки **Object Browser** перечисляются сначала методы, а потом переменные класса. И те и другие следуют в определенном порядке — сначала общедоступные, потом защищенные и, в конце, закрытые. Посмотрите и запомните, какими пиктограммами снабжаются члены класса в списке. Если во вкладке **Object Browser** "щелкнуть" мышью по названию класса, то в активном окне редактирования появится файл с объявлением соответствующего класса, причем текущей строкой будет строка заголовка класса. Аналогично, используя эту вкладку, можно в активное окно редактирования поместить файл с текущей строкой, соответствующей началу определения члена класса, который был выбран во вкладке **Object Browser**. Вы заметили, как это удобно? Советуем всегда пользоваться этой вкладкой.

Примечание

Вкладки **Resource View** и **Properties** можно сделать видимыми соответственно командами **View | Resource View** и **View | Other Window | Properties Window**.

Примечание

При необходимости, стандартное расположение окон IDE можно задать командой **Window | Reset Window Layout**.

В правой части строки заголовка окна **Properties** имеется три кнопки — средняя имеет вид "поплавка" (**Auto Hide**). При вертикальном расположении "поплавка" окно при всех манипуляциях сохраняется видимым (изменение расположения кнопки происходит при нажатии на нее левой кнопкой мыши). При горизонтальном расположении "поплавка" окно, когда к нему нет обращения, автоматически свертывается, а развертывается только на время обращения к нему.

Примечание

Аналогичные кнопки имеются во многих других окнах IDE. Это очень удобный инструмент — пользуйтесь им.

П3.5. Отладка приложения

Известно высказывание о том, что после обнаружения последней ошибки в программе остается еще хотя бы одна, стало аксиомой. Все ошибки можно разделить на *три группы* [1].

- *синтаксические* ошибки, которые выявляются на этапе компиляции. В зависимости от языка программирования, компилятор лучше или хуже выявляет такие ошибки. В ряде случаев синтаксическая ошибка в программе влечет за собой неадекватную реакцию компилятора. Например, нарушение парности скобок часто приводит к тому, что компилятор обнаруживает ошибку через десятки строк кода, обычно в конце файла. В последнем случае можно рекомендовать одновременный набор открывающей и закрывающей скобок (например, `{}`, `()`, `<>` или `[]`) с последующим вводом текста между ними;
- *логические* (часто их также называют *алгоритмическими*) ошибки. Их сложнее обнаружить и исправить. Часть из них выявляется на этапе отладки, часть на этапе сопровождения, а некоторые приводят к тяжелым последствиям;
- *информационные* ошибки, к появлению которых может привести, например, отсутствие обработки ошибок ввода-вывода, попытки деления на ноль, переполнение разрядной сетки компьютера и т. п. Для исключения и/или обработки информационных ошибок в ряде случаев приходится большую часть кода функций (методов) отводить для всевозможных проверок.

Для выявления и устранения ошибок приложения используют *средства отладки*, которые можно разделить на средства отладки, предоставляемые интерфейсом пользователя (интегрированной средой разработки), и программные средства отладки.

П3.5.1. Средства отладки IDE

Средства отладки Microsoft Visual Studio 2005 довольно многочисленны и разнообразны. Далее мы ограничимся рассмотрением только тех средств отладки, которые представляются нам наиболее удобными.

П3.5.1.1. Компиляция. Устранение синтаксических ошибок

На этапе отладки мы должны устранить все синтаксические и большую часть логических ошибок в программном проекте, допущенных на предыдущих этапах создания приложения. Для устранения синтаксических ошибок необ-

ходимо скомпилировать каждый созданный файл. Это можно сделать несколькими способами.

- *Скомпилировать каждый файл* проекта с расширением cpp. Для этой цели в окне редактирования следует выбрать соответствующий файл и воспользоваться командой **Build | Compile** или комбинацией клавиш <Ctrl>+<F7> или, что удобнее, кнопкой **Compile** на панели инструментов **Build**. При этом следует иметь в виду, что ссылки между файлами не проверяются.
- *Скомпилировать и скомпоновать все файлы проекта* ("собрать" или "построить" исполняемый файл) с помощью команды **Build | Build имя_проекта** или кнопки **Build имя_проекта** на панели инструментов **Build**, или команды **Build | Rebuild имя_проекта**. Единственным отличием этих команд является то, что команда **Rebuild имя_проекта** не проверяет даты создания файлов проекта и компилирует все файлы, а не только те, которые были модифицированы после компиляции. В результате создается исполняемый файл с расширением exe или файл с расширением dll.
- Сразу *запустить приложение*. Для этого можно использовать команду **Debug | Start Without Debugging**, запустить приложение комбинацией клавиш <Ctrl>+<F5> или, что удобнее, использовать кнопку **Start Without Debugging** на панели инструментов **Debug**. Если при этом будет предложено построить проект, то нужно ответить утвердительно.

Если программный проект содержит синтаксические ошибки, то при выполнении любой из представленных команд информация об ошибках автоматически отображается в окне **Output**, по умолчанию расположенном в нижней части IDE. Если окно **Output** было закрыто, то его можно вывести снова на экран с помощью команды **View | Output**, комбинацией клавиш <Ctrl>+ +<Alt>+<O> или <Alt>+<2>. Каждое сообщение об ошибке (error) или предупреждение (warning) начинается с имени файла, в котором они обнаружены, за которым следует номер строки, содержащей ошибку/предупреждение, и информация об ошибке (код и описание). Если дважды щелкнуть левой кнопкой мыши на строке с сообщением об ошибке или предупреждении, то ошибочная строка будет отмечена стрелкой в левой части в соответствующем окне редактирования. Имейте в виду, что это относится только к ошибкам компиляции. Чтобы получить более подробную информацию об ошибке или предупреждении, можно в окне **Output** выбрать соответствующую строку и нажать клавишу <F1>. Добейтесь того, чтобы в окончательном варианте проекта не было ошибок и предупреждений (хотя с предупреждениями исполняемый файл создается и может быть запущен).

После устранения синтаксических ошибок следует запустить программу любым из указанных ранее способов. При этом можно получить сообщение об ошибке (ax). Это тот самый случай, когда программный проект не содержит

синтаксических ошибок, а приложение не работает. Вызвано это так называемыми логическими (алгоритмическими) ошибками — т. е. сам алгоритм работы программы содержит ошибки, для обнаружения которых можно использовать разные методы (например, закомментировать фрагменты программы). Однако лучше всего воспользоваться *встроенным отладчиком*.

П3.5.1.2. Отладка приложения. Устранение логических (алгоритмических) ошибок

Встроенный *отладчик* предоставляет следующие возможности:

- запуск программы до заданного места;
- пошаговое выполнение программы;
- просмотр значений переменных в любом месте программы.

Для более удобной работы с отладчиком рекомендуем разместить в верхней части окна IDE панель инструментов для отладки и настроить ее. Для этой цели достаточно щелкнуть правой кнопкой мыши по свободной области окна рядом с любой из панелей и в контекстном меню выбрать панель **Debug**. Конечно же, это следует сделать только в том случае, если панель **Debug** была скрыта. Для настройки этой панели в нее следует добавить кнопки **Start Without Debugging** (запуск без отладки) и **Run To Cursor** (запуск до курсора). С этой целью достаточно выполнить команду **Tools | Customize**, в появившемся окне **Customize** выбрать вкладку **Commands** и в ней выбрать категорию **Debug** (рис. П3.16). Для добавления кнопки **Start Without Debugging** достаточно в списке **Commands** выбрать эту команду и с помощью левой кнопки мыши перетащить ее в нужное место панели инструментов **Debug**. Аналогичным образом на панель **Debug** добавляется кнопка **Run To Cursor**. После этого окно **Customize** можно закрыть.

Совет

Для удобства аналогичным образом включите панель инструментов **Build** и добавьте в нее кнопку **Compile**.

В результате окно IDE с панелями инструментов **Debug** и **Build** будет иметь вид, показанный на рис. П3.17.

Чтобы *запустить программу до заданного места*, предварительно необходимо установить так называемые *точки останова* (breakpoints), которые можно рассматривать как стоп-сигналы для отладчика. Конечно же, это возможно только в Debug-конфигурации проекта (Release-конфигурация отладку не поддерживает). Обычно они устанавливаются в местах, которые вызывают сомнение в правильности выполнения. При этом предполагается, что все операторы, предшествующие первой точке останова, выполняются правиль-

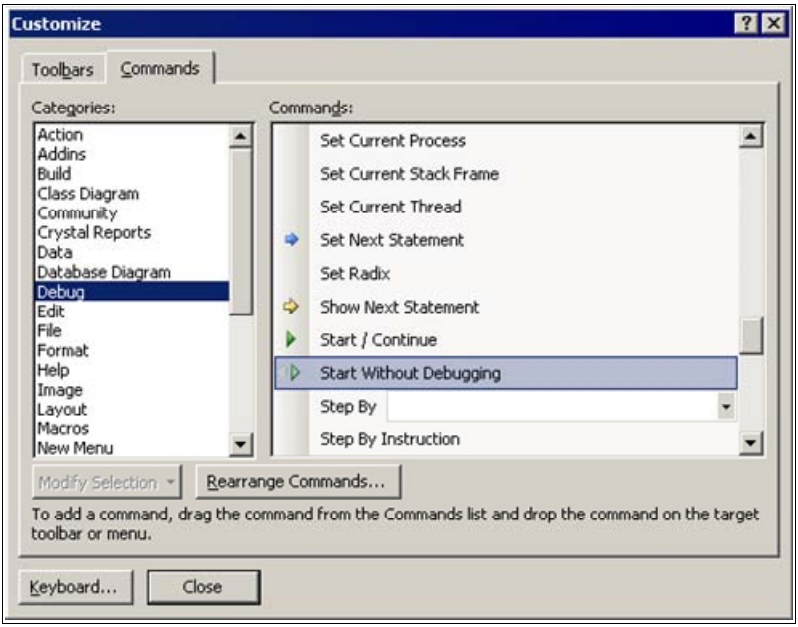


Рис. П3.16. Выбор кнопки **Start Without Debugging** для добавления ее на панель **Debug**

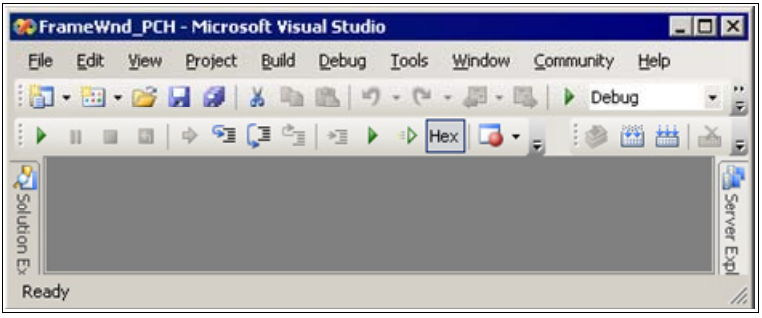


Рис. П3.17. Вид IDE с модифицированными панелями **Debug** и **Build**

но. Самый простой способ установки точки останова заключается в следующем. Курсор устанавливается на строку, на которой нужно остановить работу программы, и нажимается клавиша <F9>. Повторное нажатие клавиши <F9> удаляет точку останова. Строка останова в окне редактирования отмечена темно-красным кружком в крайней левой позиции. Если после задания точки (точек) останова запустить программу с помощью кнопки **Start Debugging** на панели инструментов **Standard**, команды **Debug | Start Debugging**, или нажав клавишу <F5>, то все операторы программы, предшествующие точке останова, будут выполняться в обычном режиме и только перед строкой останова

выполнение программы приостановится. При этом вид IDE существенно изменится. Во-первых, изменится состав команд некоторых меню, в частности меню **Debug**. Во-вторых, строка, которая будет выполняться следующей, будет отмечена желтой (по умолчанию) стрелкой. И наконец, в нижней части экрана автоматически появится новое окно с вкладками **Autos**, **Locals**, **Watch1** и другими вкладками, которые позволяют просматривать и менять значения переменных. Аналогичным образом программу можно последовательно запустить до следующих установленных точек останова. Завершить отладку приложения можно командой **Debug | Stop Debugging**, клавиатурной комбинацией <Shift>+<F5> или кнопкой **Stop Debugging** на панели инструментов **Debug**.

Для *пошагового* выполнения приложения в отладчике имеются следующие команды:

- **Debug | Step Over** (Отладка | Обойти вызов функции), клавиша <F10> или кнопка **Step Over** на панели инструментов **Debug** — выполняет операторы и/или вызовы функций в текущей строке и останавливается на следующей строке;
- **Debug | Step Into** (Отладка | Войти в функцию), клавиша <F11> или кнопка **Step Into** на панели инструментов **Debug** — выполняет операторы текущей строки и, если в ней имеется вызов функции, переходит к первому оператору вызванной функции. Если же текущая строка не содержит вызовов функции, то данная команда работает аналогично предыдущей команде;
- **Debug | Step Out** (Отладка | Выйти из функции), комбинация клавиш <Shift>+<F11> или кнопка **Step Out** на панели инструментов **Debug** — завершает выполнение текущей функции и переходит к оператору, непосредственно следующему за ее вызовом;
- кнопка **Run to Cursor** (Выполнить до курсора) на панели инструментов **Debug** выполняет программу до строки, где в текущий момент находится курсор.

Как уже говорилось ранее, в очередном месте останова, с помощью вкладок **Autos** (Авто), **Locals** (Локальные) и **Watch1** (Просмотр), можно просматривать и менять значения переменных.

Как правило, в окне **Autos** отображаются значения переменных, используемых в текущем и предыдущем операторах программы. Если значение переменной было изменено с момента прошлого его отображения в данном окне, оно отображается красным цветом. Если около имени объекта стоит знак плюс, то этот объект имеет тип класса. Чтобы посмотреть значения его членов, достаточно щелкнуть левой кнопкой мыши по этому крестику.

Два других окна — **Locals** и **Watch1** — имеют практически такую же структуру и различаются тем, что в окне **Locals** автоматически отображаются только локальные переменные текущего блока (после каждого шага выполнения программы значения этих переменных обновляются), а окно **Watch1** заполняет пользователь.

Переменные, которые нужно контролировать или изменять по желанию программиста, можно задать в окне **Watch1**. Для этого в свободной строке столбца **Name** для контроля значения переменной достаточно набрать идентификатор переменной и нажать клавишу <Enter> или выделить идентификатор переменной в окне редактирования и перетащить его, с помощью мыши, в свободную строку столбца **Name** и также нажать клавишу <Enter>. Для изменения значения переменной в процессе отладки следует выбрать строку с именем этой переменной, с помощью клавиши <Tab> перейти в столбец **Value**, набрать там новое значение и нажать клавишу <Enter>. При дальнейшей отладке, вместо прежнего значения, будет использовано модифицированное таким образом значение переменной. В этом окне могут просматриваться значения всех переменных и выражений, находящихся в текущей области видимости. Просматриваемое значение может быть, при необходимости, явно отформатировано. Синтаксис форматирования переменной или выражения предусматривает установку запятой после идентификатора переменной или выражения, а затем символа форматирования (табл. П3.1) [8].

Таблица П3.1. *Наиболее распространенные символы форматирования окна* ***Watch1***

Символ форматирования	Описание
d, i	Десятичное целое число со знаком
u	Десятичное целое число без знака
o	Восьмеричное целое число без знака
X, x	Шестнадцатеричное целое число без знака
l, h	Префикс long или short для типов d, i, u, o, x, X
f, e, g	Вещественное число в соответствующем формате (l, L — возможные префиксы)
c	Символ
s	Строка

Совет

Для просмотра значения переменной в *точке останова*, наряду с использованием окон **Autos**, **Locals** и **Watch1**, можно поставить курсор на имя интересующей нас переменной в окне редактирования. Если переменной было присвоено значение, то появится всплывающее окно со значением этой переменной. Это удобно и мы рекомендуем вам пользоваться этой возможностью.

И в этом случае завершить отладку приложения можно командой **Debug | Stop Debugging**, клавиатурной комбинацией <Shift>+<F5> или кнопкой **Stop Debugging** на панели инструментов **Debug**.

П3.5.2. Программные средства отладки [8]

При программировании приложения в него можно включить некоторые операторы, необходимые только при его отладке, т. е. только в отладочной версии приложения. Включение подобных операторов замедляет выполнение программы и чревато выдачей пользователю сообщений, понятных только программисту. Поэтому при создании окончательной версии (release) приложения эти операторы нужно удалить, это делается автоматически.

В Microsoft Visual Studio 2005 предусмотрено два режима трансляции приложения: **Debug** (Отладочный) и **Release** (Распространяемый), созданы специальные функции и макросы (например, макросы `ASSERT_VALID` и `TRACE`), работающие только в отладочном режиме и игнорируемые в окончательном приложении.

Большинству отладочных версий программ присущи следующие отличительные особенности:

- запрещена оптимизация;
- с помощью директивы препроцессора `#define` определяется макрос `_DEBUG`;
- подключаются отладочные версии библиотек поддержки и/или библиотек MFC;
- в проект включается символьная отладочная информация.

Макрос `_DEBUG` используется в директивах условной компиляции и в макросах отладки для автоматического исключения отладочной информации в окончательной версии приложения.

П3.5.2.1. Макрос *ASSERT_VALID*

Сначала приведем пример использования данного макроса (см. листинг 4.2):

```
pFrame = new FrameWnd;
ASSERT_VALID( pFrame );              // Обработка ошибки размещения
```

Макрос ASSERT_VALID (подтверждение правильности) используется для проверки некоторых логических условий, которые должны выполняться в данной точке программы. Всякий раз, когда логическое выражение (например, указатель pFrame), переданное макросу, будет принимать значение 0 (FALSE), выполнение приложения будет останавливаться и на экран будет выводиться окно сообщения. В данном окне указывается имя исполняемого файла (не всегда), имя исходного файла и номер строки, в которой произошла ошибка.

В окончательной версии приложения, в которой макрос _DEBUG не определен, макросы ASSERT_VALID не будут выполнять никаких действий. Это позволяет оставить их в тексте программы, не опасаясь, что они замедлят выполнение окончательной версии приложения или будут выдавать непонятные сообщения.

П3.5.2.2. Макрос *TRACE*

Макрос TRACE служит для вывода диагностических сообщений. Обычно, при создании приложения, программист знает те места, в которых могут возникнуть ошибки. Для вывода диагностических сообщений в подобных местах следует использовать макрос TRACE. Вообще-то под макросом TRACE понимается целое семейство макросов, включающее в себя макросы TRACE, TRACE0, TRACE1, TRACE2 и TRACE3. Числовой постфикс в имени макроса указывает на количество параметрических аргументов в данном макросе. Под параметрическим аргументом понимается идентификатор переменной, значение которой должно быть преобразовано в текстовую строку в соответствии с указанным форматом. Синтаксис макроса TRACE аналогичен синтаксису функции printf.

Макрос TRACE позволяет выводить сообщения с любым числом параметрических аргументов. Макросы TRACE0, TRACE1, TRACE2 и TRACE3 созданы исключительно с целью экономии места в сегменте памяти. Приведем пример использования макроса TRACE0 (см. листинг 5.2):

```
// Создание окна приложения
BOOL                    rc = pFrame->Create(
    // Используется предопределенное OS WINDOWS имя оконного
    //   класса
    NULL,
    // Заголовок фрейма окна
    "Демонстрация MFC-кнопок",
    // Стиль окна
    WS_SIZEBOX | WS_POPUPWINDOW | WS_DLGFRAME,
    // Местоположение и размеры окна: 0 — левая сторона,
    //   0 — верх, 530 — правая сторона, 290 — низ окна
    CRect( 0, 0, 530, 290 ) );
```

```
if( !rc )
{
    TRACE0( "\n Ошибка 1. Фрейм окна не был создан \n" );
    exit( 1 );
}
```

Содержимое строки, указанной в круглых скобках макроса TRACE0, выводится в окно **Debug**. В него же выводится код завершения приложения. Чтобы это происходило, нужно запускать приложение в режиме отладки (нажав клавишу <F5>).

В окончательной версии приложения, в которой макрос _DEBUG не определен, макросы TRACE не будут выполнять никаких действий. Это также позволяет оставить их в тексте программы, не опасаясь, что они замедлят выполнение окончательной версии приложения или будут выдавать непонятные сообщения.

П3.6. Тестирование приложения

Как и любой другой продукт, программа перед использованием должна быть тщательно проверена. Этот этап является едва ли не самым сложным во всем процессе создания программы — необходимо учесть все варианты ее поведения. Поэтому его нужно начинать не после завершения отладки, а одновременно с разработкой алгоритма. Одним из путей проверки или тестирования программы является ее выполнение по одному разу с каждой из возможных комбинаций входных данных — т. е. тестирование с использованием заранее подготовленного набора контрольных примеров.

Требования к контрольным примерам:

- набор контрольных примеров должен быть достаточным, чтобы показать выполнение всех требований технического задания и обеспечить полную проверку программного проекта — протестировать все ветви, имеющиеся в программе;
- контрольные примеры должны быть простыми в том смысле, чтобы анализ ожидаемых результатов был несложным (примеры должны быть небольшой размерности со значениями исходных данных, удобными для анализа).

Еще раз отметим, что создание достаточного и простого набора контрольных примеров является нетривиальной задачей.

Структура контрольного примера может быть, например, следующей:

- цель примера;
- исходные данные (для примеров с нормальным завершением) или как моделировать ошибку (для примеров с аварийным завершением);

- анализ ожидаемых результатов (для примеров с нормальным завершением — анализ ожидаемых результатов в точках останова).

Правила выбора точек останова:

- точки останова следует выбирать после выполнения каждой функции программного проекта (в них следует проверить результаты работы функции);
- если декомпозиция задачи выполнена не очень удачно и функция получилась большой (более страницы текста), то следует в ее теле выбрать промежуточные точки останова, разбив тело функции на алгоритмически законченные части;
- если функция была отлажена ранее, то после нее точку останова выбирать не следует;
- если функция результаты своей работы выводит в файл на диске, на экран или на принтер, то после такой функции точки останова можно, в принципе, не задавать.

Методика тестирования приложения для контрольных примеров с нормальным завершением.

- Программа запускается до первой точки останова так, как это указывалось ранее. Если полученные результаты совпадают с результатами анализа, приведенного в контрольном примере, то аналогично программа запускается до следующей точки останова и т. д.
- Если в очередной точке останова машинные результаты отличаются от ожидаемых, то текущий сеанс отладки завершается с помощью команды **Debug | Stop Debugging**. Программа повторно запускается до последней точки останова с хорошими результатами и с этого места выполняется по шагам с проверкой полученных результатов (выполняется построчная трассировка ошибочного участка). По результатам пошаговой проверки обнаруживается ошибка и текущий сеанс отладки также прекращается. Затем в исходный текст программы вносятся необходимые исправления и трассировка ошибочного участка повторяется. Этот процесс заканчивается после исправления ошибок, о чем будет свидетельствовать получение в очередной точке останова ожидаемых результатов.

П3.7. Использование встроенной справочной системы

Совет

После установки Microsoft Visual Studio 2005 в качестве следующего шага обязательно устанавливайте справочную систему MSDN (Microsoft Developer

Network) Library Help. При ее установке обязательно используйте опцию **Full** (Полная установка).

Доступ к справочной системе обеспечивается командами меню **Help** (рис. П3.18). Окна справки, появляющиеся в результате активизации команд меню **Help**, могут отображаться в окне IDE Microsoft Visual Studio 2005 или в окнах отдельного приложения справочной системы.

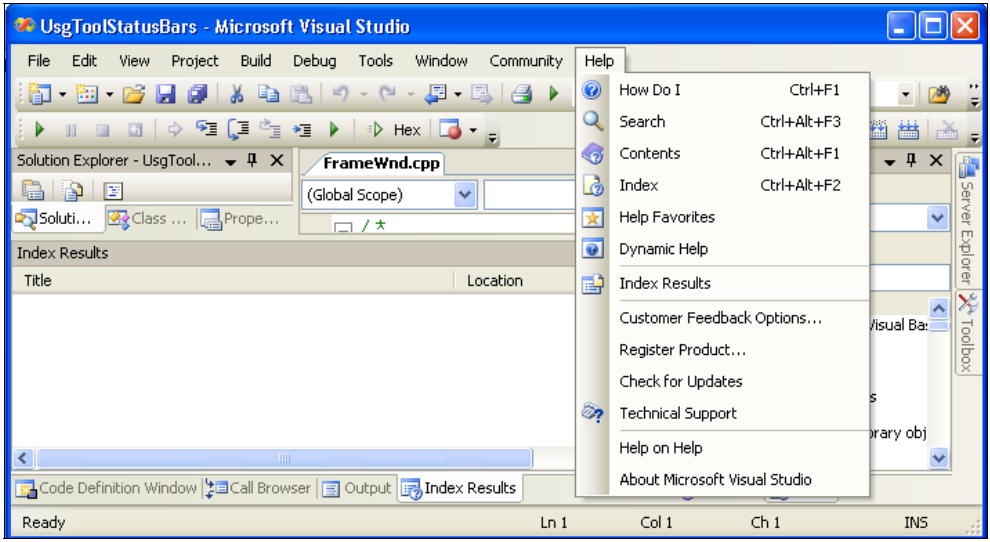


Рис. П3.18. Встроенная справочная система — команды меню **Help**

Для отображения окна справки в окне IDE следует выбрать команду **Tools | Options...**, в появившемся окне **Options** последовательно открыть **Environment** и **Help**, выбрать вкладку **General** и в поле **Show Help using:** выбрать **Intergated Help Viewer**. Такой способ отображения окна справки удобен при большом размере экрана. Для отображения окна справки в окне отдельного приложения справочной системы в поле **Show Help using:** следует выбрать **External Help Viewer**. Такой способ отображения окна справки целесообразно использовать при относительно небольшом размере экрана. При дальнейшем изложении мы будем ориентироваться на первый способ отображения окна справки.

П3.7.1. Команда *Help | Search*

При активизации данной команды на экран в область окна редактирования IDE выводится окно **Search** (Поиск). Данное окно позволяет выполнять поиск заданного текста в интерактивной справке IDE. Пример результата поиска в этом окне приведен на рис. П3.19.

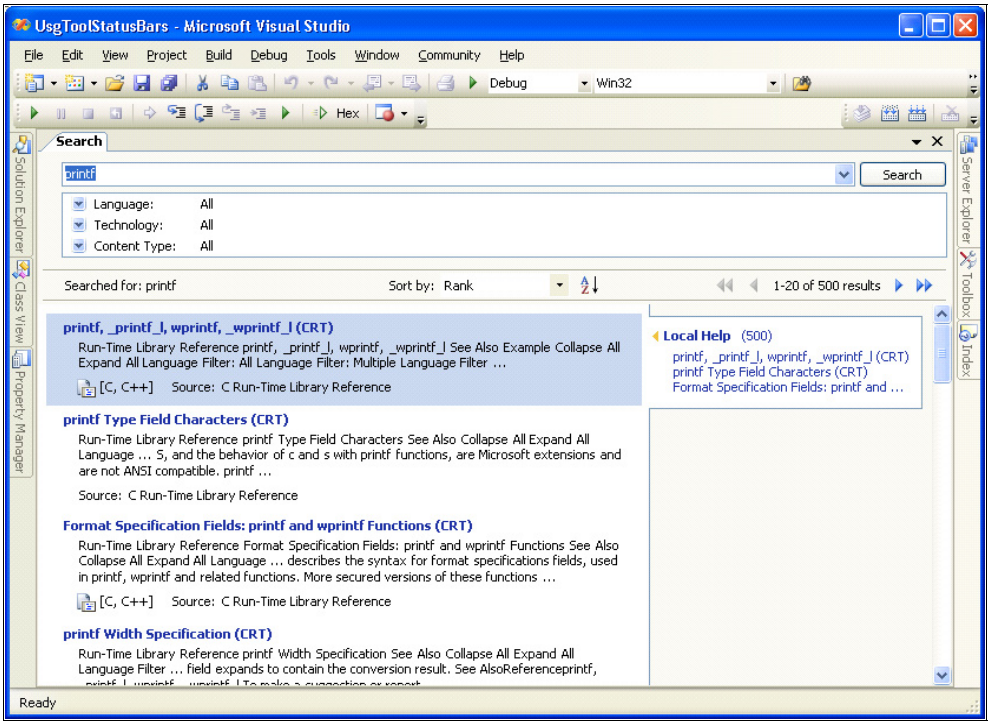


Рис. П3.19. Окно **Search**

П3.7.2. Команда *Help | Contents*

При выборе данной команды на экран IDE выводится окно **Contents** (Содержание), показанное на рис. П3.20. Это окно позволяет получить доступ к интерактивной справке IDE Microsoft Visual Studio 2005.

Совет

Обратите внимание на свойство **Dockable** (Присоединяемое) окна справки, показанного на рис. П3.20, и на список значений в поле **Filtered by:** (Фильтр по), используемых для фильтрации справочной информации (рис. П3.21).

Совет

Если окно **Contents** расположено не так, как показано на рис. П3.20, то следует предварительно выполнить удобную и весьма полезную команду **Window | Reset Windows Layout**, о которой уже говорилось ранее, и далее еще раз выполнить команду **Help | Contents**.

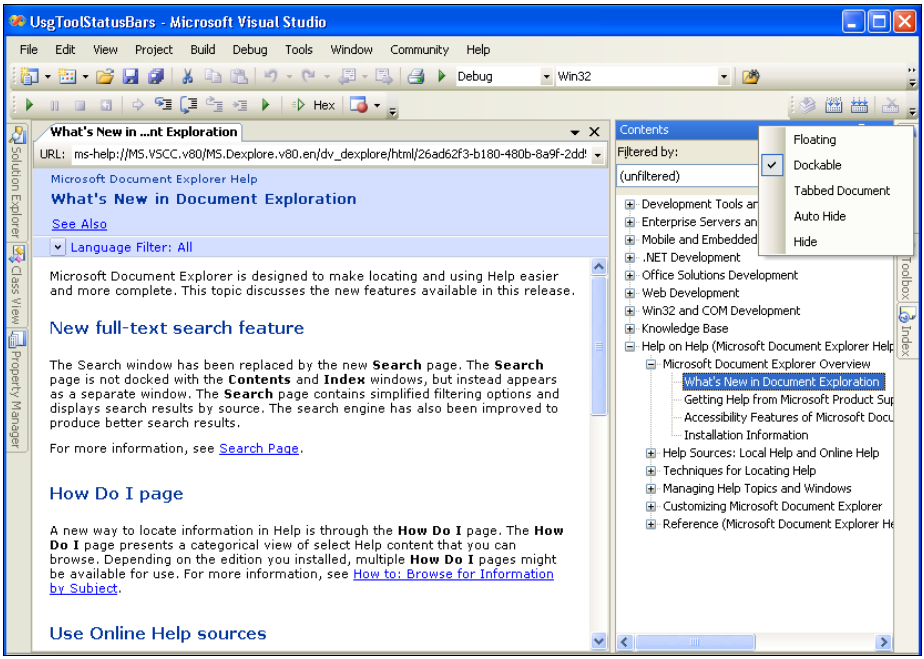


Рис. П3.20. Окно **Contents**

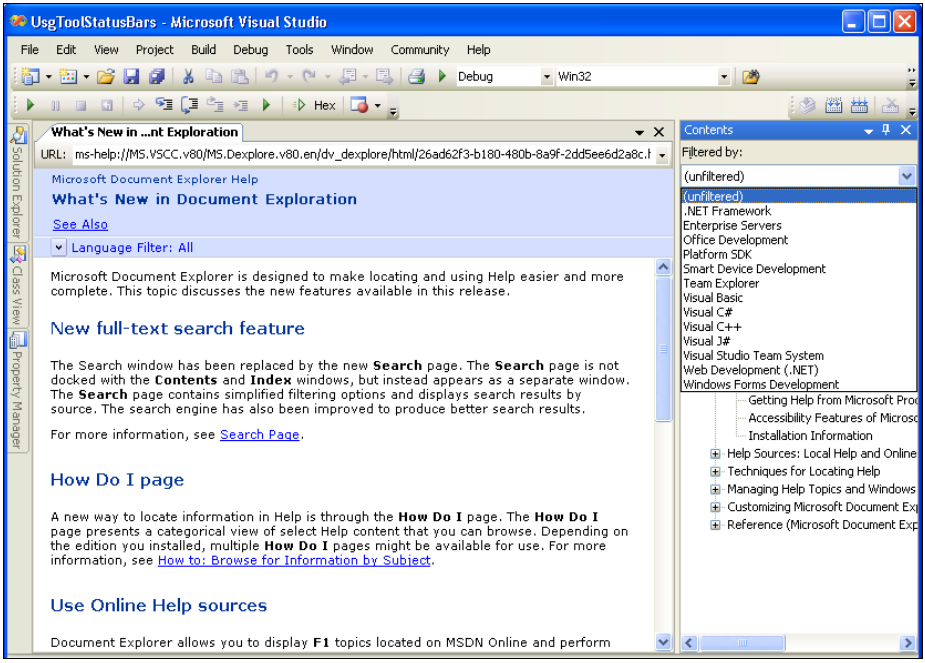


Рис. П3.21. Фильтрация справочной информации

П3.7.3. Команда *Help* | *Index*

При активизации команды **Help** | **Index** на экран IDE выводится окно справки **Index** (Предметный указатель), показанное на рис. П3.22. Данное окно позволяет произвести поиск по ключу в интерактивной справке IDE Microsoft Visual Studio 2005.

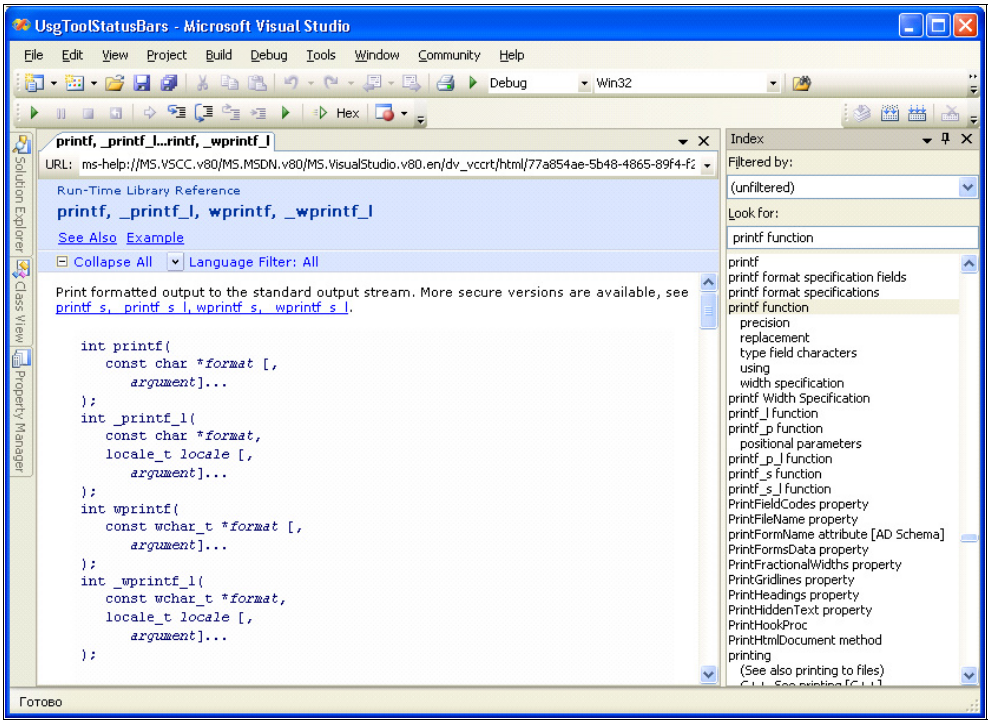


Рис. П3.22. Окно **Index**

П3.7.4. Команда *Help* | *Dynamic Help*

Эта команда меню используется для получения краткой справки по элементу IDE, с которым в данный момент производятся некоторые действия. Информация выводится в окно справки **Dynamic Help** (Оперативная справка), показанное на рис. П3.23.

П3.7.5. Команда *Help* | *Index Results*

Если результаты поиска по ключу, заданному в окне **Index** (Предметный указатель), не позволили однозначно определить требуемую тему справки, то

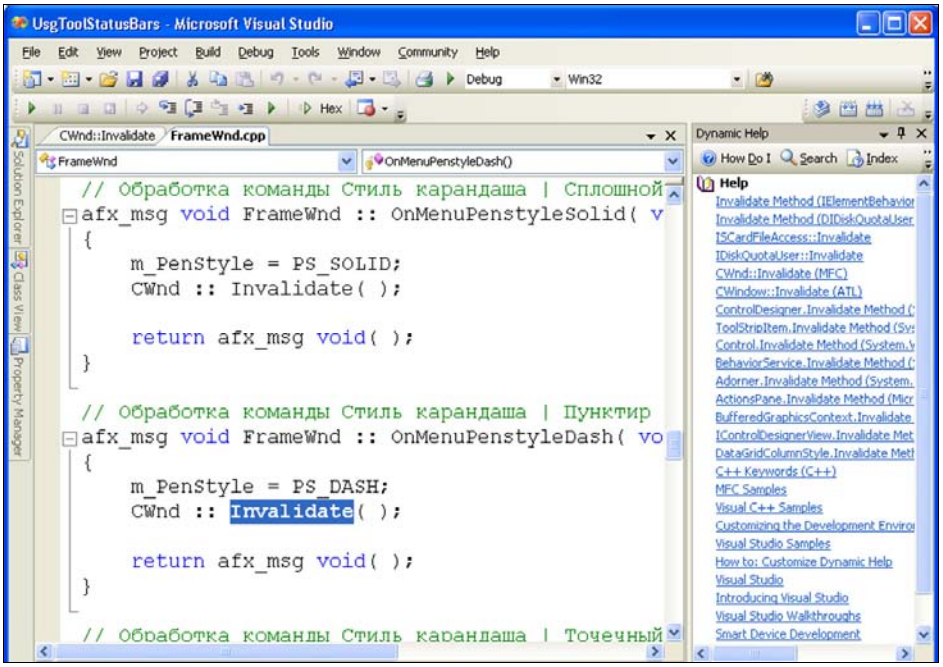


Рис. П3.23. Окно **Dynamic Help**

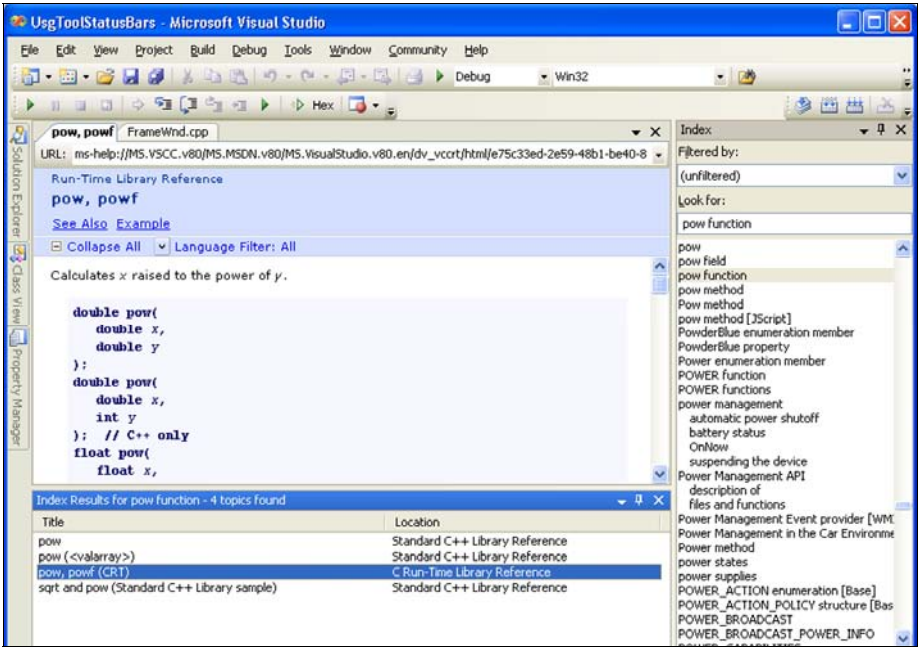


Рис. П3.24. Окно **Index Results**

будет выведено окно **Index Results** (Найденные разделы). Это окно расположено в нижней части рис. П3.24 и содержит темы, из которых следует сделать выбор.

П3.7.6. Остальные команды меню *Help*

Остальные команды меню **Help** (см. приведенный ранее рис. П3.18) — **Register Product...** (Регистрация продукта), **Technical Support** (Техническая поддержка), **Help of Help** (Справка о справке), **About Microsoft Visual Studio** (Об IDE) и др. являются менее востребованными и по этой причине не рассматриваются. Отметим лишь, что информацию о них можно получить с помощью команды **Help of Help** (Справка о справке).

Совет

Если компьютер, на котором вы работаете, не имеет выхода в Интернет, то следует отключить получение online-справки. Для этого следует выполнить команду **Tools | Options...**, последовательно раскрыть **Environment** и **Help**, выбрать вкладку **Online** и настроить ее в соответствии с рис. П3.25.

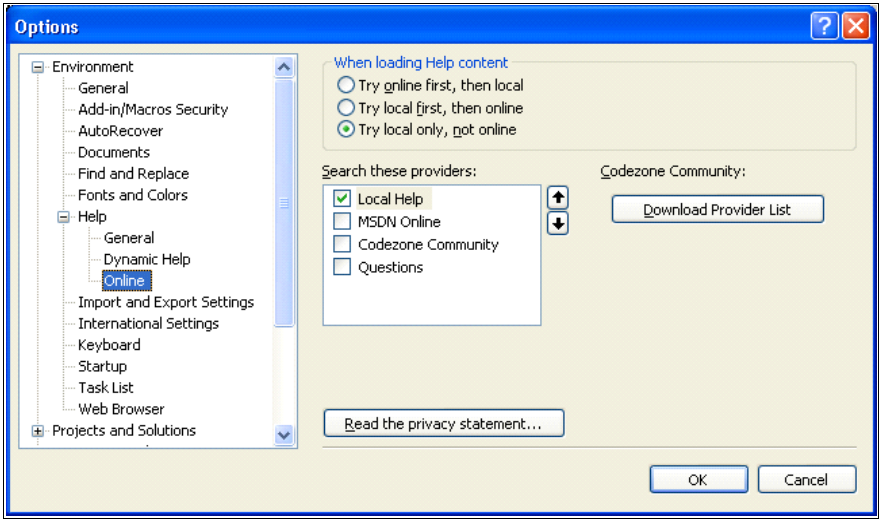


Рис. П3.25. Настройка компьютера, не имеющего выхода в Интернет

Примечание

Обращаем ваше внимание на еще один, весьма удобный, способ получения справочной информации. Вы можете поместить курсор или на ключевое слово, или на идентификатор типа, или на идентификатор функции (метода), или на другой элемент исходного кода и нажать клавишу <F1>. В результате в окне

редактирования IDE появится в виде вкладки справка о выбранном элементе. Пример такого окна справки для ключевого слова `if` в качестве выбранного элемента представлен на рис. П3.26. Пользуйтесь этой удобной возможностью.

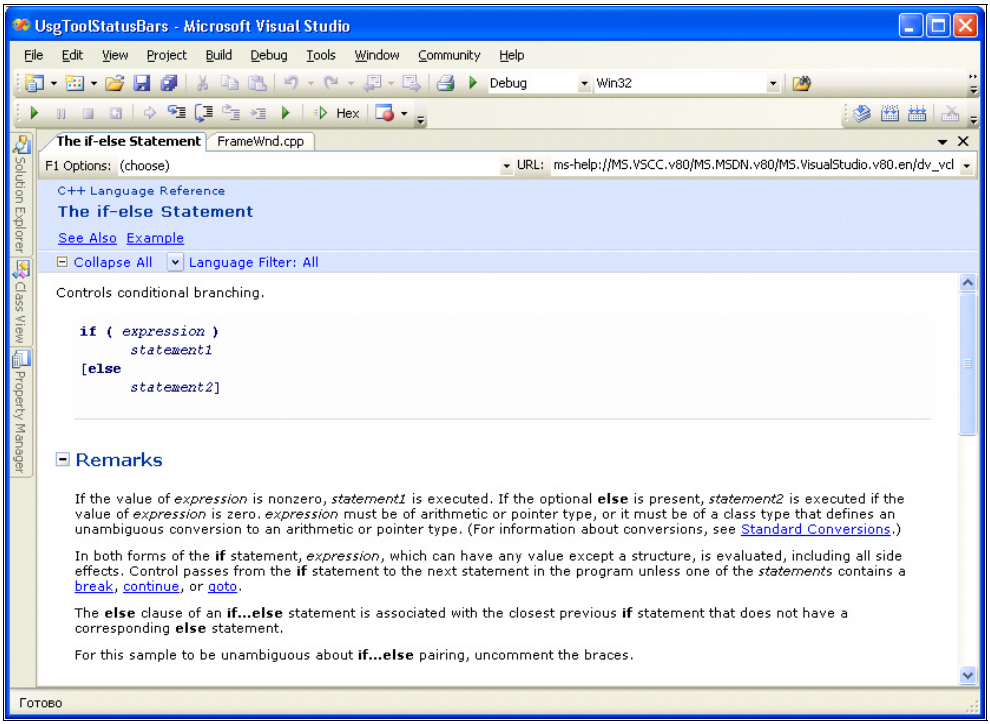


Рис. П3.26. Получение справки с помощью клавиши <F1>

ПРИЛОЖЕНИЕ 4

Описание прилагаемого компакт-диска

Папки и файлы компакт-диска, а также их назначение приведены в табл. П4.1.

Таблица П4.1. *Расположение папок и файлов компакт-диска и их назначение*

Папка или файл на компакт-диске	Описание	Глава
Демонстрационные примеры\UsgBitmap	Низкоуровневое приложение на базе Windows API с использованием вывода изображений	Полезный пример, не рассмотренный в книге
Демонстрационные примеры\UsgBitmap_MFC	Приложение на базе MFC с использованием вывода изображений	Полезный пример, не рассмотренный в книге
Демонстрационные примеры\Глава 02\FrameWnd	Низкоуровневое приложение на базе Windows API с выводом текста и графики в главное окно	2
Демонстрационные примеры\Глава 04\FrameWnd	Приложение на базе MFC без прекомпиляции с выводом текста и графики в главное окно	4
Демонстрационные примеры\Глава 04\FrameWnd_PCH	Приложение на базе MFC с прекомпиляцией с выводом текста и графики в главное окно	4
Демонстрационные примеры\Глава 05\WndBtn	Низкоуровневое приложение на базе Windows API с кнопками	5
Демонстрационные примеры\Глава 05\WndBtn_MFC	Приложение на базе MFC с кнопками	5
Демонстрационные примеры\Глава 05\WndWndBtn	Низкоуровневое приложение на базе Windows API с кнопками и дочерними окнами	5

Таблица П4.1 *(продолжение)*

Папка или файл на компакт-диске	Описание	Глава
Демонстрационные примеры\Глава 05\ WndWndBtn_MFC	Приложение на базе MFC с кнопками и дочерними окнами	5
Демонстрационные примеры\Глава 06\UsgMenu	Низкоуровневое приложение на базе Windows API с использованием меню	6
Демонстрационные примеры\Глава 06\UsgMenu_MFC	Приложение на базе MFC с использованием меню	6
Демонстрационные примеры\Глава 07\UsgToolStatusBars	Приложение на базе MFC с использованием панели инструментов и строки статуса	7
Демонстрационные примеры\Глава 08\NoModalDlgUsgEdit	Приложение на базе MFC с использованием немодального диалогового окна с элементом редактирования	8
Демонстрационные примеры\Глава 08\UsgModalDlg	Приложение на базе MFC с использованием модального диалогового окна с радиокнопками	8
Демонстрационные примеры\Глава 08\WndEditBtn	Приложение на базе MFC с использованием элемента редактирования вне диалогового окна	8
Демонстрационные примеры\Глава 09\UsgBtnBmp	Приложение на базе MFC с использованием рисованных кнопок	9
Демонстрационные примеры\Глава 09\UsgCheckBox	Приложение на базе MFC с использованием отмечаемых кнопок	9
Демонстрационные примеры\Глава 09\UsgGroupBox	Приложение на базе MFC с использованием элемента группирования	9
Демонстрационные примеры\Глава 09\UsgScroll	Приложение на базе MFC с использованием полос прокрутки	9
Демонстрационные примеры\Глава 09\UsgSlider	Приложение на базе MFC с использованием ползунка	9
Демонстрационные примеры\Глава 09\UsgSpinEdit	Приложение на базе MFC с использованием спина	9
Демонстрационные примеры\Глава 10\ UsgComboBoxes	Приложение на базе MFC с использованием комбинированных списков	10
Демонстрационные примеры\Глава 10\UsgListBoxes	Приложение на базе MFC с использованием списков	10
Демонстрационные примеры\Глава 10\ UsgProgressTimer	Приложение на базе MFC с использованием индикатора прогресса и таймера	10

Таблица П4.1 *(продолжение)*

Папка или файл на компакт-диске	**Описание**	**Глава**
Демонстрационные примеры\Глава 11	Содержит четыре приложения на базе MFC с использованием DLL расширений. FrameWndExDLL — создание DLL для приложения без ресурсов. TestFrameWndExDLL — использование DLL для приложения без ресурсов. UsgMenuExDLLResource — создание DLL для приложения с ресурсами. TestUsgMenuExDLLResource — использование DLL для приложения с ресурсами	11
Демонстрационные примеры\Глава 12	Содержит четыре приложения на базе MFC, полученных с помощью мастера и русифицированных. SDIApp — каркас приложения с архитектурой SDI. SDIApp1 — русифицированный вариант приложения SDIApp. MDIApp — каркас приложения с архитектурой MDI. MDIApp1 — русифицированный вариант приложения MDIApp.	12
Демонстрационные примеры\Глава 13	Содержит четыре приложения на базе MFC, демонстрирующих добавление в каркас приложения новых меню, панелей инструментов и русифицированных тем справки. SDIApp2 — приложение SDIApp1 с добавленными меню и панелью инструментов. SDIApp3 — приложение SDIApp2 с русифицированными темами справки для добавленных меню и панели инструментов. MDIApp2 — приложение MDIApp1 с добавленными меню и панелью инструментов. MDIApp3 — приложение MDIApp2 с русифицированными темами справки для добавленных меню и панели инструментов	13

Таблица П4.1 *(окончание)*

Папка или файл на компакт-диске	Описание	Глава
Тесты и курсовое проектирование. Варианты заданий.doc	Файл содержит полный текст приложения 2, имеющегося в книге в формате текстового редактора Microsoft® Word — более 120 вариантов тестов по основным разделам учебного пособия, 30 вариантов заданий на курсовое проектирование и материал, посвященный экзаменационному тестированию с использованием тестовых вопросов. Его наличие очень удобно для преподавателей — можно быстро и без ошибок копировать и печатать фрагменты файла для использования на практических занятиях и при экзаменационном тестировании	2, 4—13
\Copy	Папка, в которую, для надежности, помещена резервная копия информации, имеющейся на компакт-диске	

Исходный код всех демонстрационных примеров, имеющихся в учебном пособии, приведен на компакт-диске, в виде программных проектов, выполненных в IDE Microsoft Visual Studio 2005. Это удобно для экспериментов с ними. Для выполнения с демонстрационным примером желаемых вами действий не требуется создавать программный проект и вводить исходные файлы программы — достаточно лишь скопировать папку соответствующего проекта и перейти к экспериментам. Это существенно сэкономит время и устранит возможные ошибки кодирования. Программные проекты демонстрационных примеров включают также и исполняемые файлы (exe), т. е. вам не надо обязательно компилировать и компоновать заинтересовавшие вас примеры. Все приводимые программные проекты "самодостаточны". Это означает, что ни одному из них не требуются файлы других проектов.

Для приложений на базе MFC исполняемые файлы (exe) получены с использованием опции **Use MFC in a Static Library**, что позволяет запускать приложение на любом компьютере с операционной системой Microsoft Windows XP/2000, даже если на нем не установлена IDE Microsoft Visual Studio 2005. Если же на компьютере установлена IDE, то целесообразно заново получить исполняемый файл с использованием вместо указанной ранее опции **Use MFC in a Shared DLL** (устанавливается в поле **Use of MFC** вкладки **Configuration Properties | General** окна **Properties Pages**, появляющегося

при выполнении команды **Properties** контекстного меню проекта). Это позволит значительно (в несколько раз) уменьшить размер исполняемого файла.

Программные проекты демонстрационных примеров сгруппированы в папке "Демонстрационные примеры" в соответствии с главами книги (*главы 2*, *4—13*), что позволяет легко привязать их к тексту книги. В эту папку дополнительно помещены еще два интересных и полезных демонстрационных примера (проекты UsgBitmap и UsgBitmap_MFC), которые не рассматривались в книге.

Предметный указатель

W

А

Б

В

Г

Д

З

И

К

М

Н

О

П

Р

C

T

Ф

Ц

Ш

Э