



БИБЛИОТЕКА ПРОГРАММИСТА

Юрий Щупак

Win32 API

Разработка приложений
для Windows



Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2008

Щупак Юрий Абрамович

Win32 API. Разработка приложений для Windows

Серия «Библиотека программиста»

Заведующий редакцией

Руководитель проекта

Ведущий редактор

Корректор

Верстка

A. Сандрыкин

П. Манинен

Ю. Сергиенко

Е. Каюрова

С. Романов

ББК 32.973.2-018.2

УДК 004.451

Щупак Ю. А.

Щ95 Win32 API. Разработка приложений для Windows. — СПб.: Питер, 2008. — 592 с.: ил.

ISBN 978-5-388-00301-0

В этой книге изложены основные концепции и приемы программирования для Windows на языке C/C++ с применением Win32 API.

Книга ориентирована на широкий круг читателей: от начинающих программистов, студентов вузов, аспирантов и преподавателей до профессионалов в области программирования, владеющих языком C++, но не имеющих опыта разработки приложений для Windows.

Практика показывает, что если программисты начинают разработку проектов сразу с применением библиотек классов, подобных MFC или Windows Forms, не имея при этом опыта работы с Win32 API, то они сталкиваются с серьезными проблемами, как только дело доходит до создания реальных приложений. Напротив, опыт программирования с Win32 API позволяет осваивать более высокие технологии гораздо гармоничней и продуктивней.

Владение базовыми знаниями Win32 API является надежной основой для вашего профессионального роста в сфере программирования.

© ООО «Питер Пресс», 2008

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-5-388-00301-0

ООО «Питер Пресс», 198206, Санкт-Петербург, Петергофское шоссе, д. 73, лит. А29.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Подписано в печать 28.05.08. Формат 70x100/16. Усл. п. л. 47,73. Тираж 2500. Заказ

Отпечатано по технологии СтР в ОАО «Печатный двор» им. А. М. Горького.

197110, Санкт-Петербург, Чкаловский пр., 15.

Краткое содержание

Предисловие	13
Глава 1. «Hello, World!», или Первые шаги к пониманию основных концепций Windows	16
Глава 2. GDI — графический интерфейс устройства. Рисование линий, фигур, текста	58
Глава 3. GDI. Палитры, раstry, метафайлы	159
Глава 4. Средства ввода	213
Глава 5. Ресурсы Windows-приложения	242
Глава 6. Меню и быстрые клавиши	272
Глава 7. Диалоговые окна	304
Глава 8. Элементы управления общего пользования	379
Глава 9. Многозадачность	441
Глава 10. Таймеры и время	485
Глава 11. Библиотеки динамической компоновки DLL	515
Глава 12. Специальные приложения	539
Приложение 1. Интегрированная среда Visual C++ 6.0	563
Приложение 2. Интегрированная среда Visual Studio.NET ..	577
Приложение 3. Работа с утилитой Spy++	584
Литература	587
Алфавитный указатель	588

Содержание

Предисловие	13
Кому адресована эта книга	14
Как работать с книгой	14
От издательства	15
Глава 1. «Hello, World!», или Первые шаги к пониманию основных концепций Windows	16
Базовые концепции	16
Графический интерфейс пользователя	16
Многозадачность	17
Управление памятью	18
Независимость от аппаратных средств	19
Вызовы функций и DLL	19
Объектно-ориентированное программирование	20
Типы данных Win32	20
Архитектура, управляемая событиями	21
Оконная процедура	22
Оконные классы	23
Цикл обработки сообщений	25
Наипростейшая программа для Windows	25
Программа «Hello, world!» — первый вариант	30
Файл исходного текста программы	31
Регистрация класса окна	32
Создание окна	35
Использование функции CreateWindowEx	38
Отображение окна на экране	39
Обработка сообщений	40
Оконная процедура	42

Программа «Hello, world!» — второй вариант	46
Функции поддержки окон	49
Часто используемые сообщения	50
Модификация характеристик окна	51
Особенности программирования для Windows	52
Синхронные и асинхронные сообщения	53
Посылка сообщений из приложения	54
Использование глобальных или статических переменных	55
Получение дескриптора экземпляра приложения	55
Предотвращение зависания приложения в случае медленной обработки отдельных событий	56
Использование утилиты Spy++	57
Глава 2. GDI — графический интерфейс устройства.	
Рисование линий, фигур, текста	58
Контекст устройства	59
Типы контекстов устройства	59
Регионы Windows. Отсечение	60
Контекст дисплея	60
Использование сообщения WM_PAINT	63
Контекст принтера	65
Контекст в памяти (совместимый контекст)	65
Метафайловый контекст	66
Информационный контекст	66
Системы координат и преобразования	66
Экранные, оконные и клиентские координаты	66
Типы координатных систем	67
Физическая система координат	68
Система координат устройства	68
Страницчная система координат и режимы отображения	69
Мировая система координат	72
Получение информации о возможностях устройства	74
Управление цветом. Вывод пикселя	76
Цветовое пространство HLS	76
Цветовое пространство RGB	76
Определение цвета при работе с палитрой	77
Вывод пикселов	78

Атрибуты контекста устройства, влияющие на рисование	78
Режим рисования. Бинарные растровые операции	79
Режим смешивания фона и цвета графического элемента	81
Общие операции с графическими объектами	81
Линии и кривые	82
Рисование отрезков	83
Дуги	84
Кривые Безье	86
Перья	87
Стандартные перья	88
Простые перья	89
Расширенные перья	93
Кисти	96
Стандартные кисти	97
Пользовательские кисти	97
Замкнутые фигуры	101
Прямоугольники	101
Эллипсы, сегменты, секторы и закругленные прямоугольники	105
Многоугольники	107
Регионы и отсечение	109
Создание регионов	109
Операции с объектами регионов	111
Прорисовка регионов	111
Отсечение	112
Отображение текста	113
Наборы символов и кодировки	114
Unicode	117
Шрифты	118
Вывод текста	129
Полосы прокрутки и вывод текста	141
Примеры программ	146
Просмотрщик текстовых файлов	146
Вывод временной диаграммы напряжения переменного электрического тока	153
Глава 3. GDI. Палитры, растры, метафайлы	159
Палитры	159
Основные принципы управления палитрами	159

Системная палитра	161
Логическая палитра	166
Растры	174
Аппаратно-независимые растры	175
Аппаратно-зависимые растры	193
DIB-секции	200
Тернарные растровые операции	202
Метафайлы	206
Создание метафайла	207
Воспроизведение метафайла	210
Глава 4. Средства ввода	213
Клавиатура	213
Фокус ввода	214
Клавиши и символы	215
Аппаратные сообщения	215
Символьные сообщения	217
Работа с кареткой	219
Примитивный текстовый редактор	220
Мышь	225
Терминология, связанная с мышью	226
Сообщения мыши	226
Обработка двойного щелчка	227
Обработка сообщений от колеса мыши	227
Рисуем мышью	228
Эластичные прямоугольники	234
Улучшенное приложение для просмотра текстовых файлов	237
Глава 5. Ресурсы Windows-приложения	242
Редакторы ресурсов	243
Пиктограммы	243
Создание пиктограммы с помощью графического редактора	246
Импорт существующей пиктограммы	250
Просмотр и редактирование ресурсов приложения	251
Использование ресурса в приложении	252
Курсоры	256
Растровые образы	260
Ресурсы, определяемые программистом	263

Доступ к данным в ресурсе	263
Воспроизведение звуковых файлов	264
Таблицы строк	268
Глава 6. Меню и быстрые клавиши	272
Организация и виды меню	272
Типы пунктов меню	273
Системное меню	274
Клавиатурный интерфейс меню	273
Статус пунктов меню	275
Отметка пунктов меню	275
Пункт меню, применяемый по умолчанию	276
Определение меню в виде ресурса	276
Шаблон меню	276
Вызов редактора меню	277
Атрибуты пункта меню	278
Уровни меню	279
Процедура определения пункта для меню i-го уровня	279
Процедура определения меню нулевого уровня	280
Добавление меню к окну приложения	280
Внесение изменений в меню	281
Функция CheckMenuItem	282
Функция CheckMenuItem	282
Функция EnableMenuItem	283
Функция ModifyMenu	283
Функции для получения дескриптора меню	284
Сообщения меню	284
Приложение MenuDemo1	286
Работа с контекстным меню	293
Определение шаблона контекстного меню	293
Загрузка меню	293
Вызов меню	293
Приложение MenuDemo2	294
Быстрые клавиши	299
Модификация определения ресурса меню	299
Таблица быстрых клавиш	300
Загрузка таблицы быстрых клавиш	301

Модификация цикла обработки сообщений	301
Приложение MenuDemo3	302
Глава 7. Диалоговые окна	304
Типы диалоговых окон	304
Элементы управления в диалоговом окне	305
Создание и обработка диалогового окна	307
Шаблон диалогового окна	307
Шаблонная система единиц.....	308
Модальный диалог	308
Вызов и использование редактора диалоговых окон	309
Добавление элемента управления Рисунок	311
Добавление элементов управления Надпись	312
Выравнивание элементов управления на форме диалога	314
Определение диалоговой процедуры и вызов диалога	316
Изменение атрибутов элемента управления	320
Использование других элементов управления	323
Кнопки	323
Кнопка Owner draw	325
Флажки	331
Переключатели	333
Групповая рамка	333
Пример использования групповой рамки, флажков и переключателей ..	334
Клавиатурный интерфейс и порядок обхода элементов управления	340
Окно редактирования	341
Список	344
Комбинированный список	356
Немодальный диалог	366
Различия между модальными и немодальными окнами диалога	366
Пример использования немодального окна диалога	367
Окно сообщений	372
Диалоговые окна общего пользования	374
Глава 8. Элементы управления общего пользования	379
Основы применения	380
Инициализация библиотеки	380
Создание элементов управления общего пользования	382
Стили элементов управления общего пользования	383

Обмен сообщениями	384
Элементы управления главного окна	385
Панель инструментов	385
Окно подсказки	406
Замена класса KWnd на класс KWndEx	408
Строка состояния	413
Другие элементы управления	423
Индикатор процесса	423
Регулятор	428
Счетчик и поле с прокруткой	435
Создание счетчика	435
Глава 9. Многозадачность	441
Объекты ядра	441
Процессы и потоки	443
Планирование потоков	444
Классы приоритетов процесса и приоритеты потоков	445
Управление процессами	447
Использование функции CreateProcess	447
Завершение процесса	448
Запуск обособленных дочерних процессов	449
Управление потоками	451
Функция CreateThread	451
Функция Sleep	452
Пример многопоточного приложения	452
Взаимодействие потоков через глобальную переменную	456
Синхронизация	459
Атомарный доступ и семейство Interlocked-функций	459
Критические секции	460
Wait-функции	461
События	464
Семафоры	466
Мьютексы	467
Обмен данными между процессами	469
Виртуальная память. Адресное пространство процесса	470
Файлы данных, проецируемые в память	472
Использование проекции файла для реализации разделяемой памяти ..	472

Модель «клиент-сервер»	473
Обмен данными с помощью сообщения WM_COPYDATA	474
Приложение ServerApp	475
Приложение ClientApp	479
Не забывайте освобождать ресурсы	483
Когда многопоточность реально полезна?	484
Глава 10. Таймеры и время	486
Время Windows	486
Системное время	487
Измерение малых временных интервалов	488
Использование счетчика монитора производительности	488
Использование команды RDTSC	489
Программирование задержек в исполнении кода	497
Использование функции Sleep	498
Использование метода uDelay класса KTimer	501
Класс QTimer	502
Стандартный таймер	504
Первый способ использования стандартных таймеров	504
Второй способ использования стандартных таймеров	508
Мультимедийный таймер	509
Функции timeSetEvent и timeKillEvent	509
Тестирование мультимедийного таймера	511
Глава 11. Библиотеки динамической компоновки DLL	515
DLL и адресное пространство процесса	514
Создание собственной DLL	517
Вызов функций из DLL	519
Неявная загрузка DLL	519
Явная загрузка DLL	521
Отложенная загрузка DLL	523
Загрузка ресурсов из DLL	525
Функция входа/выхода	527
Локальная память потока (TLS)	529
Динамическая TLS	531
Статическая TLS	538

Глава 12. Специальные приложения	539
Анимация	539
Приложение со стандартным таймером	539
Двойная буферизация	543
Рисование в реальном времени	547
Требования к приемнику информации от метеорадиолокатора	547
Разработка модели программного имитатора	548
Приложение 1. Интегрированная среда Visual C++ 6.0	563
Запуск IDE. Типы приложений	563
Создание нового проекта	565
Добавление к проекту файлов с исходным кодом	566
Многофайловые проекты	568
Компиляция, компоновка и выполнение проекта	568
Конфигурация проекта	569
Как закончить работу над проектом	569
Как открыть проект, над которым вы ранее работали	569
Встроенная справочная система	570
Работа с отладчиком	570
Некоторые полезные инструменты	574
Приложение 2. Интегрированная среда Visual Studio.NET	577
Создание нового проекта	578
Добавление к проекту нового файла	580
Компиляция, сборка и выполнение	582
Работа с редакторами ресурсов	583
Приложение 3. Работа с утилитой Spy++	584
Список литературы	587
Алфавитный указатель	588

Предисловие

API – это аббревиатура названия *Application Programming Interface* (интерфейс прикладного программирования). API представляет собой совокупность функций и инструментов, позволяющих программисту создавать приложения (программы), работающие в некоторой среде.

Win32 API – это набор функций для создания программ, работающих под управлением Microsoft Windows 98, Windows NT или Windows 2000. Все функции этого набора являются 32-битными, что отражено в названии интерфейса.

При написании прикладных программ для Windows программистам приходится пользоваться огромным объемом документации, который изложен в справочной системе MSDN, содержащей очень мало примеров и представленной только на английском языке. Часто приходится тратить много времени, чтобы понять, как работает та или иная функция, и какова область ее применения. Особенно серьезные проблемы возникают у разработчиков, имеющих поверхностные представления о принципах функционирования операционной системы и о механизмах ее взаимодействия с приложениями.

В данной книге изложены основные концепции и приемы программирования для Windows на языке C/C++ с применением Win32 API. Существенное внимание в ней уделяется вопросам взаимодействия операционной системы и приложений. Даны основы рисования и копирования изображений, а также использования Windows-ресурсов. Рассмотрено создание графического интерфейса пользователя, включая меню, панели инструментов, диалоговые окна и элементы управления. Также излагаются основы построения многопоточных приложений и библиотек динамической компоновки DLL. Рассмотрены специальные вопросы рисования: анимация, технология рисования в реальном времени. Показано использование классов C/C++ для применения концепций объектно-ориентированного программирования при создании Windows-приложений.

Владение базовыми знаниями интерфейса прикладного программирования существенно облегчает изучение технологий программирования на более высоком уровне, например, с использованием библиотеки классов MFC (Microsoft Foundation Classes) или библиотеки классов Windows Forms.

Дело в том, что реализация как MFC, так и Windows Forms, основана на вызове функций Windows API. С точки зрения архитектуры эти интерфейсы представляют собой надстройку над Win32 API и предназначены для облегчения программирования Windows-приложений. Однако практика показывает, что для программистов, начинающих писать программы сразу на уровне MFC или Windows Forms, возникают серьезные проблемы, как только дело доходит до создания реальных приложе-

ний. И, наоборот, если вы сначала приобретаете опыт программирования с Win32 API, освоение более высоких технологий проходит гораздо гармоничней и продуктивней.

Есть и другой аспект, немаловажный для профессионального программирования. Очевидно, что MFC и Windows Forms не только повышают производительность труда программиста, но и обладают меньшей гибкостью по сравнению с интерфейсом более низкого уровня. Поэтому когда к эффективности реализации приложения предъявляются повышенные требования (например, по затратам памяти и быстродействию), более целесообразным является кодирование на уровне Win32 API.

На содержание книги повлияли:

- информация, почерпнутая автором из книг таких мэтров Windows-программирования как Чарльз Петцольд (Charles Petzold) [1], Джейфри Рихтер (Jeffrey Richter) [5] и Фень Юань (Feng Juan) [6];
- опыт практического программирования автора в ООО «Контур-НИИРС»;
- опыт проведения автором семинаров по программированию для Windows в Санкт-Петербургском государственном университете информационных технологий, механики и оптики.

Данная книга является вторым изданием, исправленным и дополненным, книги «Win32 API. Эффективная разработка приложений», опубликованной издательством «Питер» в 2006 г. В новой редакции книги добавлена глава, посвященная разработке DLL.

Кому адресована эта книга

Книга ориентирована на широкий круг читателей. Она будет полезна начинающим программистам, студентам вузов, аспирантам и преподавателям, а также более опытным коллегам, владеющим языком C++, но не имеющим опыта разработки приложений для Windows.

Предполагается, что читатель знаком с языком C/C++ и умеет если не писать, то хотя бы читать и понимать программы, написанные на этом языке. Если это не так, то придется на время отложить эту книжку в сторону, пока вы не ликвидируете этот досадный пробел в вашем образовании. Но не уходите надолго — книга будет вас ждать...:)

Как работать с книгой

Эта книга должна дать читателю базовые знания по программированию на C/C++ с использованием Win32 API. Конечно, для их усвоения недостаточно просто прочитать изложенный здесь материал. Необходимо активное и глубокое погружение в среду Win32 API, то есть выполнение большинства программных примеров на вашем компьютере.

Все приведенные в книге примеры протестированы на компьютере с операционной системой Microsoft Windows 2000 Professional в среде Microsoft Visual Studio 6.0. Вы можете использовать любую из систем Windows 98, Windows NT/2000 или

Windows XP. Желательно компилировать приведенные примеры программ в Visual Studio 6.0, выполняя указания по построению проектов, приведенные в Приложении 1.

Можно также компилировать проекты и в среде Microsoft Visual Studio.NET, но при этом надо точно выполнять требования к типу проекта, которые приведены в Приложении 2.

Указанные выше рекомендации по работе с книгой относятся к начинающим программистам. Более опытные программисты могут пользоваться книгой как справочником, обращаясь к тем разделам, которые содержат интересующую их информацию.

Программные примеры, рассмотренные в книге, содержат несколько классов и утилит общего назначения, которые вы можете применять и в своих приложениях. Имена этих классов начинаются с буквы английского алфавита «K», чтобы не было путаницы с классами библиотеки MFC.

С целью сокращения объема книги во многих листингах программ повторяющиеся фрагменты кода опущены, но делается ссылка на предшествующий листинг, содержащий этот фрагмент. Если у вас возникнут какие-то вопросы в связи с этими ссылками, вы всегда сможете найти полные тексты программ в комплекте файлов к данной книге, размещенном на сайте издательства «Питер».

В рамках одной книги невозможно осветить все аспекты программирования для Windows. Однако автор надеется, что после освоения базовых знаний вам будет гораздо легче разобраться с другими разделами программирования для Windows.

Автор будет признателен всем читателям за замечания, вопросы и пожелания, относящиеся к содержанию книги, которые можно присыпать по адресу shupak@mail.ru.

От издательства

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

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы можете найти по адресу <http://www.piter.com/download>.

Подробную информацию о наших книгах вы найдете на веб-сайте издательства: <http://www.piter.com>.

1

«Hello, World!», или Первые шаги к пониманию основных концепций Windows

Читатель, скорее всего, уже имеет опыт *пользователя* той или иной версии операционной системы Windows. Трудно предположить, что есть пользователь, который ни разу не работал с текстовым редактором Microsoft Word или с другим приложением пакета Microsoft Office. Переход в категорию *программиста* требует, однако, более близкого знакомства с внутренним устройством Windows. Поэтому, как бы ни хотелось нам поскорее откомпилировать код нашей первой программы, все же сначала придется поговорить об общих принципах функционирования операционных систем семейства Windows.

Базовые концепции

Операционная система Windows по сравнению с операционными системами типа MS-DOS обладает серьезными преимуществами и для пользователей, и для программистов. Среди этих преимуществ обычно выделяют:

- графический интерфейс пользователя;
- многозадачность;
- управление памятью;
- независимость от аппаратных средств.

Графический интерфейс пользователя

*Graphical User Interface*¹ (GUI) дает возможность пользователям работать с приложениями максимально удобным способом. Каждое приложение представлено на экране дисплея своим окном, которое выглядит как прямоугольная рабочая область с набором стандартных элементов управления. Окно идентифицируется своим заголовком, имеет кнопки минимизации и максимизации размеров, а также кнопку завершения приложения. Под заголовком обычно находится строка меню для выбора различных команд или режимов работы. Всплывающие окна

¹ Графический интерфейс пользователя.

диалога, вызываемые командами меню, также содержат привычные для пользователя элементы управления, такие как текстовые поля ввода информации, открывающиеся списки и кнопки различных типов. Большинство программ для Windows поддерживают работу с клавиатурой, и с мышью.

Стандартизация графического интерфейса имеет очень большое значение для пользователя, потому что одинаковый интерфейс экономит его время и упрощает изучение новых приложений. С точки зрения программиста, стандартный вид интерфейса обеспечивается использованием подпрограмм, встроенных непосредственно в Windows, что также приводит к существенной экономии времени при написании новых программ.

Многозадачность

Многозадачные операционные системы позволяют пользователю одновременно работать с несколькими приложениями или несколькими копиями одного приложения. Например, пользователь может открыть Visual Studio 6.0 и набирать в текстовом редакторе код одной из программ, приведенных в этой книге, одновременно слушая приятную музыку, воспроизводимую приложением «Проигрыватель Windows Media». В это же время программа explorer.exe может заниматься поиском необходимого файла по всему дисковому пространству компьютера.

В первых версиях Windows была реализована так называемая *кооперативная* (или *невытесняющая*) многозадачность, когда приложения должны были сами отдавать управление операционной системе, чтобы дать возможность работать другим программам. Такая стратегия имела существенный недостаток, ведь любое некорректно работающее приложение могло «подвесить» операционную систему. Но уже в Windows 95 была введена *вытесняющая* многозадачность. Операционная система автоматически переключается с одной задачи на другую, не ожидая, пока выполняемая программа освободит управление процессором.

Многозадачность осуществляется в Windows при помощи процессов и потоков. Любое приложение Windows после запуска реализуется как *процесс (process)*. Грубо говоря, процесс можно представить как совокупность программного кода и выделенных для его исполнения системных ресурсов. При инициализации процесса система всегда создает первичный (основной) *поток (thread)*, который исполняет код программы, манипулируя данными в адресном пространстве процесса.

Из основного потока при необходимости могут быть запущены один или несколько вторичных потоков, которые выполняются одновременно с основным потоком. На самом деле истинный параллелизм возможен только при исполнении программы на многопроцессорной компьютерной системе, когда есть возможность распределить потоки между разными процессорами. В случае обычного однопроцессорного компьютера операционная система выделяет по очереди некоторый квант времени каждому потоку.

Но каковы соотношения между потоками и окнами? Дело в том, что окно всегда принадлежит некоторому потоку. Поток может быть владельцем одного или нескольких окон, а может быть и вовсе безоконным. Например, если вторичный поток создан для приема данных из СОМ-порта, то он вполне может обойтись без своего окна.

Наконец, сами окна, принадлежащие потоку, находятся в некоторых иерархических взаимоотношениях. Одно окно является *окном верхнего уровня* (*top-level window*)¹, другие окна называются *дочерними* (*child windows*). Дочерние окна подчиняются своим *родительским окнам* (*parent windows*). Рисунок 1.1 иллюстрирует эти взаимоотношения.

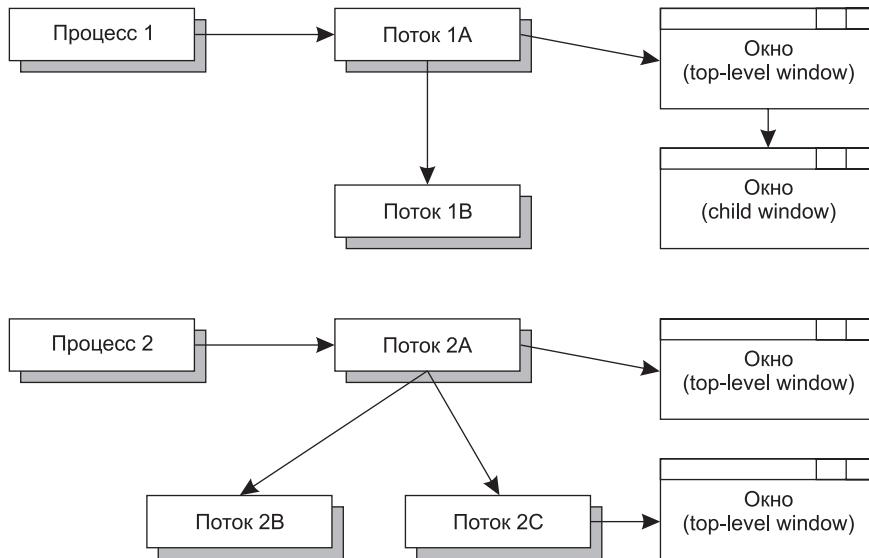


Рис. 1.1. Соотношения между процессами, потоками и окнами

Если на экране находится несколько окон, то в каждый момент времени только одно из них может быть *активным*. Активное окно отличается более ярким фоном на строке заголовка, в то время как пассивные окна имеют более тусклый фон заголовка. Говорят, что активное окно имеет *фокус ввода*. Это означает, что любая информация, вводимая пользователем при помощи клавиатуры или мыши, будет перенаправляться операционной системой именно данному окну.

Управление памятью

Память — это один из важнейших разделяемых ресурсов в операционной системе. Если одновременно запущены несколько приложений, то они должны разделять память, не выходя за пределы выделенного адресного пространства. Так как одни программы запускаются, а другие завершаются, то память фрагментируется. Система должна уметь объединять свободное пространство памяти, перемещая блоки кода и данных.

Система Windows обеспечивает достаточно большую гибкость в управлении памятью. Если объем доступной памяти меньше объема исполняемого файла, то система может загружать исполняемый файл по частям, удаляя из памяти отработавшие фрагменты. Если пользователь запустил несколько копий, кото-

¹ Окно верхнего уровня не имеет родительского окна.

рые также называют отдельными экземплярами приложения, то система размещает в памяти только одну копию исполняемого кода, которая используется этими экземплярами совместно. Программы, запущенные в Windows, могут использовать также функции из других файлов, которые называются *библиотеками динамической компоновки — DLL (dynamic link libraries)*. Система Windows поддерживает механизм связи программ во время их работы с функциями из DLL. Даже сама операционная система Windows, по существу, является набором динамически подключаемых библиотек.

Механизмы управления памятью непрерывно совершенствовались по мере развития Windows. В данный момент Win32 API реализован для 32-разрядной операционной системы с плоским адресным пространством.

Независимость от аппаратных средств

Еще одним преимуществом Windows является независимость от используемой платформы. У программ, написанных для Windows, нет прямого доступа к аппаратной части таких устройств отображения информации, как, например, экран или принтер. Вместо этого они вызывают функции графической подсистемы Win32 API, называемой *графическим интерфейсом устройства (Graphics Device Interface, GDI)*.

Функции GDI реализуют основные графические команды при помощи обращения к программным драйверам соответствующих аппаратных устройств. Одна и та же команда (например, LineTo — нарисовать линию) может иметь различную реализацию в разных драйверах. Эта реализация скрыта от программиста, использующего Win32 API, что упрощает разработку приложений.

Таким образом, приложения, написанные с использованием Win32 API, будут работать с любым типом дисплея и любым типом принтера, для которых имеется в наличии драйвер Windows. То же самое относится и к устройствам ввода данных — клавиатуре, манипулятору «мышь» и т. д. Такая независимость Windows от аппаратных средств достигается благодаря указанию требований, которым должна удовлетворять аппаратура, в совокупности с SDK (Software Development Kit — набор разработки программ) и/или DDK (Driver Development Kit — набор разработки драйверов устройств). Разработчики нового оборудования поставляют его вместе с программными драйверами, которые обязаны удовлетворять этим требованиям.

Вызовы функций и DLL

Win32 API поддерживает вызовы свыше двух тысяч функций, которые можно использовать в приложениях. Все основные функции Windows объявлены в заголовочных файлах. Главным заголовочным файлом является windows.h. В этом файле содержится множество ссылок на другие заголовочные файлы.

Вызовы функций Win32 API в программе осуществляются аналогично вызовам библиотечных функций C/C++. Основное различие заключается в том, что компоновщик связывает код библиотечных функций C/C++ с кодом программы на этапе компоновки (*статическое связывание*), в то время как для функций Windows это связывание откладывается и осуществляется только

на этапе выполнения программы (*динамическое связывание*). Библиотеки динамической компоновки (DLL) содержатся в файлах с расширением .dll. Большая часть этих библиотек расположена в подкаталоге **SYSTEM** каталога установки Windows.

Операционная система позволяет компилятору использовать библиотеки импорта, поставляемые в составе используемой среды программирования. Библиотеки импорта содержат имена всех функций из динамически подключаемых библиотек, а также ссылки на них. Используя эту информацию, компоновщик размещает в исполняемом файле таблицу, по которой в процессе загрузки программы настраиваются адреса вызываемых функций Win32 API.

Объектно-ориентированное программирование

Хотя формально операционная система Windows не является объектно-ориентированной системой, тем не менее, в ней реализована именно объектно-ориентированная идеология. Это наиболее очевидно для базового объекта, с которым имеют дело и операционная система, и программист, и пользователь приложения, — то есть для окна.

Как уже говорилось ранее, окно выглядит как прямоугольная область на экране. Окно получает информацию от клавиатуры или мыши пользователя и выводит графическую информацию на экран. Пользователь рассматривает окна в качестве объектов и непосредственно взаимодействует с ними, перемещая их по экрану, выбирая команды меню, нажимая различные кнопки и передвигая бегунок на полосах прокрутки.

Программист тоже рассматривает окна в качестве объектов, которые получают от пользователя информацию в виде оконных сообщений. Кроме этого окно обменивается сообщениями с другими окнами. Понимание этих сообщений — ключ к осмысленному программированию в среде Windows.

Можно привести и другие примеры объектов в Windows, к которым относятся, например, многочисленные графические объекты, используемые для рисования, такие как перья, кисти, шрифты, палитры и многие другие объекты.

Независимо от своего типа, любой объект в Windows идентифицируется своим *дескриптором*, или *описателем*. Оба этих названия являются переводом английского термина *handle*¹. Дескриптор — это своего рода ссылка на объект. Все взаимоотношения программного кода с объектом осуществляются только через его дескриптор. Система Windows тщательно скрывает свои внутренние секреты и не допускает прямого доступа к внутренним структурам объекта.

Типы данных Win32

Программиста, пишущего на C/C++, первое знакомство с программой для Windows поражает обилием типов данных. На самом деле все они определены посредством директив #define или #typedef в заголовочных файлах Win32. В табл. 1.1 приведены некоторые наиболее часто встречающиеся типы данных Windows.

¹ В среде программистов можно встретить еще один вариант «перевода» — хэндл.

Таблица 1.1. Некоторые часто используемые типы Win32

Тип данных	Описание
BOOL	Булевский тип (эквивалентный bool)
BYTE	Байт (8-битное целое без знака)
DWORD	32-битное целое без знака
HANDLE	Дескриптор объекта
HBITMAP	Дескриптор битмэпа (растрового изображения)
HBRUSH	Дескриптор кисти
HCURSOR	Дескриптор курсора
HDC	Дескриптор контекста устройства
HFONT	Дескриптор шрифта
HICON	Дескриптор иконки (пиктограммы)
HINSTANCE	Дескриптор экземпляра приложения
HMENU	Дескриптор меню
HPEN	Дескриптор пера
HWND	Дескриптор окна
INT	32-битное целое со знаком
LONG	32-битное целое со знаком
LPARAM	Тип, используемый для описания lParam, четвертого параметра оконной процедуры
LPCSTR	Указатель на константную C-строку ¹
LPCTSTR	LPCTSTR, если определен макрос UNICODE, и LPCSTR в противном случае
LPCWSTR	Указатель на константную Unicode-строку ²
LPSTR	Указатель на C-строку
LPTSTR	LPWSTR, если определен макрос UNICODE, и LPSTR в противном случае
LPWSTR	Указатель на Unicode-строку
LRESULT	Значение типа LONG, возвращаемое оконной процедурой
NULL	((void*) 0)
TCHAR	Wchar_t (Unicode-символ), если определен макрос UNICODE, и char в противном случае
UINT	32-битное целое без знака
WPARAM	Тип, используемый для описания wParam, третьего параметра оконной процедуры

Архитектура, управляемая событиями

В основе взаимодействия программы с внешним миром и с операционной системой лежит концепция *сообщений*.

С точки зрения приложения, сообщение является уведомлением о том, что произошло некоторое событие, которое может требовать, а может и не требовать

¹ С-строка обязана завершаться нулевым байтом. В каждом байте содержится один символ в соответствии с кодировкой ANSI. Далее, если это не оговаривается особо, под строками будут подразумеваться именно С-строки.

² Unicode-строка также завершается нулевым байтом, но для хранения каждого символа используется два байта. Чтобы использовать в проекте кодировку UNICODE, необходимо выполнять ряд правил, которые приводятся в разделе «Отображение текста» в главе 2.

выполнения определенных действий. Это событие может быть следствием действий пользователя, например перемещения курсора или щелчка кнопкой мыши, изменения размеров окна или выбора пункта меню. Кроме того, событие может генерироваться приложением, а также операционной системой.

Сообщение — это структура данных, содержащая следующие элементы:

- дескриптор окна, которому адресовано сообщение;
- код (номер) сообщения;
- дополнительную информацию, зависящую от кода сообщения.

Не следует забывать, что Windows — многозадачная операционная среда. Сообщения от внешних источников, например от клавиатуры, адресуются в каждый конкретный момент времени только одному из работающих приложений, а именно — активному окну. Но каким же образом Windows играет роль диспетчера сообщений? Для этого с момента старта операционная система создает в памяти глобальный объект, называемый *системной очередью сообщений*. Все сообщения, генерируемые как аппаратурой, так и приложениями, помещаются в эту очередь. Windows периодически опрашивает эту очередь и, если она не пуста, посыпает очередное сообщение нужному адресату, определяемому при помощи дескриптора окна.

Сообщения, получаемые приложением, могут поступать асинхронно из разных источников. Например, приложение может работать с системным таймером, посылающим ему сообщения с заданным интервалом, и одновременно оно должно быть готовым в любой момент получить любое сообщение от операционной системы. Чтобы не допустить потери сообщений, Windows одновременно с запуском приложения создает глобальный объект, называемый *очередью сообщений приложения*. Время жизни этого объекта совпадает с временем жизни приложения.

Таким образом, путь следования сообщений, вызванных аппаратными событиями, можно представить так:

аппаратное событие → системная очередь сообщений →
очередь сообщений приложения.

Оконная процедура

Для понимания механизма обработки приложением поступающих к нему сообщений сначала нужно объяснить, что такое «оконная процедура».

Оконная процедура — это «функция обратного вызова», предназначенная для обработки сообщений, адресованных любому окну того «оконного класса», в котором содержится ссылка на данную процедуру.

Если вы прочли только что сделанное определение, не споткнувшись ни на одном из терминов, значит, вы — не новичок в программировании для Windows. Для тех же, кто видит эти термины впервые, нужны дополнительные разъяснения.

Начнем с понятия «функция обратного вызова». Так называют функции, которые вызывает сама операционная система. Поэтому в коде приложения вы не найдете прямого вызова такой функции. Компилятор узнает функцию обратного вызова по спецификатору **CALLBACK**. Оконная процедура обычно имеет заголовок со стандартным синтаксисом:

```
LRESULT CALLBACK Имя_функции(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
```

В этом определении LRESULT — тип возвращаемого значения (см. табл. 1.1), hWnd — дескриптор окна, которому адресовано сообщение, uMsg — код сообщения, wParam и lParam — параметры сообщения. Имя функции может быть произвольным, но для главного окна приложения обычно используется имя WndProc.

В теле функции после объявления необходимых локальных переменных обычно размещается оператор switch, внутри которого и происходит обработка нужных сообщений.

Каждому коду сообщения в Windows сопоставлен уникальный символьический идентификатор. Все системные идентификаторы определены при помощи директивы #define в заголовочном файле winuser.h. Это облегчает чтение и понимание программ.

Чаще всего приложение обрабатывает *оконные сообщения* (*window messages*), начинающиеся с префикса WM_, например: WM_PAINT, WM_SIZE, WM_MOVE и многие другие. Другие типы сообщений могут поступать от элементов управления, например: сообщения с префиксом BM_ поступают от кнопок, сообщения с префиксом EM_ — от текстовых полей, сообщения с префиксом LB_ — от списков.

При необходимости разработчик может определить в приложении и потом использовать собственные коды сообщений. Обмен такими сообщениями возможен, конечно, только между окнами данного приложения. Иные приложения не смогут обрабатывать пользовательские сообщения.

Теперь о новом термине «оконный класс». На самом деле все окна создаются на базе того или иного оконного класса. Оконный класс играет роль *типа* для данного окна.

Оконные классы

Оконный класс (*window class*), или *класс окна* — это структура, определяющая основные характеристики окна. К ним относятся стиль окна и связанные с окном ресурсы, такие как пиктограмма, курсор, меню и кисть для закрашивания фона. Кроме того, одно из полей структуры содержит адрес оконной процедуры, предназначенней для обработки сообщений, получаемых любым окном данного класса.

Ссылка на оконный класс передается функции CreateWindow, вызываемой для создания окна.

Использование класса окна позволяет создавать множество окон на основе одного и того же оконного класса и, следовательно, использовать одну и ту же оконную процедуру. Например, все кнопки в программах Windows созданы на основе оконного класса BUTTON. Оконная процедура этого класса, расположенная в динамически подключаемой библиотеке, управляет обработкой сообщений для всех кнопок всех окон. Аналогичные системные классы имеются и для других элементов управления, таких как, например, списки и поля редактирования. В совокупности эти классы называются *предопределеными* или *стандартными оконными классами*.

Windows содержит предопределенный оконный класс также и для диалоговых окон, играющих важную роль в графическом интерфейсе пользователя.

Для главного окна приложения обычно создается собственный класс окна, учитывающий индивидуальные требования к программе.

ВНИМАНИЕ

Не следует путать понятие *оконного класса Windows* с понятием *класса C++* или, в частности, с понятием класса в библиотеке MFC. Дело в том, что исторически концепция оконных классов опередила по времени использование в Windows объектно-ориентированных языков.

Но как приложение узнает, что к нему пришло сообщение? Об этом рассказывается в следующем разделе.

Цикл обработки сообщений

Непременным компонентом всех Windows-приложений является *цикл обработки сообщений*. У приложения всегда есть главная функция `WinMain`. Обычно она содержит вызовы функций для инициализации и создания окон, после чего следует цикл обработки сообщений и необходимый код для закрытия приложения.

Что происходит в цикле обработки сообщений? Как известно, все сообщения, адресованные приложению, Windows записывает в очередь сообщений приложения. Извлечение сообщения из этой очереди осуществляется функцией `GetMessage`.

Если очередное сообщение имеет код `WM_QUIT`, то происходит выход из цикла, после чего приложение завершает свою работу.

Если очередное сообщение не является сообщением `WM_QUIT`, то оно передается функции `DispatchMessage`, которая возвращает сообщение обратно в Windows. Windows отправляет сообщение для его обработки соответствующей оконной процедуре — иными словами, Windows вызывает оконную процедуру. После возврата из оконной процедуры Windows передает управление оператору, который расположен после `DispatchMessage`, и работа цикла продолжается.

Все... Почва вспахана, взбранена, унавожена, увлажнена... Осталось опустить в нее первое зернышко...

Наипростейшая программа для Windows

Если вы еще ни разу не компилировали программу в интегрированной среде Microsoft Visual C++ 6.0, то оставьте, пожалуйста, закладку на этой странице и обратитесь к приложению 1, где изложена технология создания нового проекта, добавления к нему файлов с исходным кодом, компиляции, выполнения и отладки проекта.

OK! Вы уже вернулись?.. Тогда продолжим.

Создайте новый проект типа `Win32 Application` с именем `HelloFromMessageBox`.

Добавьте к проекту новый файл с именем `HelloFromMessageBox.cpp` и введите в него следующий код:

```
#include <windows.h>
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MessageBox(NULL, "Hello, Win32 world!", "Hello from Message Box", MB_OK);
    return 0;
}
```

Откомпилируйте проект. В случае появления ошибок компиляции проверьте, не был ли искажен исходный код при вводе. Если компиляция завершилась успешно, запустите программу на выполнение. Вы должны получить результат, показанный на рис. 1.2.



Рис. 1.2. Окно программы HelloFromMsgBox

Что же можно делать с этим приложением? Прочитать радостное сообщение *Hello, Win32 world!*. Это раз. Перемещать окно по экрану, ухватившись мышью за его заголовок. Это два. Щелкнуть левой кнопкой мыши на кнопке *OK*. Это три. Еще можно щелкнуть мышью на кнопке закрытия окна. В двух последних случаях приложение завершает свою работу, а его окно исчезает с экрана.

Не так уж и много возможностей, верно?.. Ну а что же вы хотите от программы, содержащей всего семь строк кода и всего две инструкции в теле функции *WinMain*?

Рассмотрим теперь код нашей миниатюрной программы. В первой строке находится следующая инструкция:

```
#include <windows.h>
```

Эта директива подключает к программе главный заголовочный файл Windows-приложений.

Далее следует заголовок функции *WinMain*:

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,  
    LPSTR lpCmdLine, int nCmdShow)
```

Эта функция эквивалентна функции *main* для DOS или UNIX, то есть с нее всегда начинается выполнение программы.

Функция возвращает значение типа *int* и принимает следующие параметры:

- **HINSTANCE hInstance** — дескриптор, который Windows присваивает запущенному приложению.
- **HINSTANCE hPrevInstance** — в Win32 этот параметр не используется и поэтому всегда принимает нулевое значение¹.
- **LPSTR lpCmdLine** — указатель на строку, в которую копируются аргументы приложения, если оно запущено в режиме командной строки. Запуск приложения в этом режиме возможен либо с помощью команды стартового меню Пуск ▶ Выполнить, либо из оболочки типа Norton Commander. При этом в командной строке набирается имя приложения, а после пробела указываются аргументы, разделенные также символом пробела. Есть еще один способ запуска приложения в этом режиме — через ярлык EXE-файла. Если у вас создан такой ярлык (являющийся ссылкой на EXE-файл), то щелкните на пиктограмме ярлыка правой кнопкой мыши и выберите в контекстном меню пункт Свойства. Затем в появившемся диалоговом окне Свойства укажите параметры командной строки, введя их в текстовое поле Object.

¹ Этот параметр остался для совместимости версий от Windows 3.x, где он использовался для представления дескриптора предыдущего экземпляра программы.

- ❑ int nCmdShow — целое значение, которое может быть передано функции `ShowWindow`. Этот параметр будет рассмотрен позже.

Спецификатор `WINAPI` определяет *соглашения о вызове*, то есть принятый в Win32 *порядок передачи параметров* при вызове функций. Для нашей дальнейшей работы этой информации достаточно, поэтому нужно просто запомнить, что в заголовке функции `WinMain` он должен быть всегда.

Разобравшись с заголовком функции, перейдем к ее телу. Здесь имеется вызов функции `MessageBox`, которая, собственно, и выполняет всю содержательную работу приложения.

Ну а где же обещанный ранее цикл обработки сообщений? Где текст оконной процедуры?.. Раскроем секрет: написание такой лаконичной программы стало возможно только благодаря использованию функции `MessageBox`.

Прежде чем объяснять этот феномен, ознакомимся с досье на функцию `MessageBox`, которое можно найти в справочной системе MSDN¹. Для этого нужно установить текстовый курсор на любом символе имени функции и нажать клавишу F1. В появившемся диалоговом окне *Найденные разделы* следует выбрать строку `MessageBox` и нажать кнопку *Показать*. На экран будет выведено описание функции `MessageBox`, из которого следует, что функция создает, отображает и обслуживает *окно сообщений*. Окно сообщений — это диалоговое окно, содержащее указанное программистом текстовое сообщение и одну или несколько кнопок.

Функция имеет следующий прототип²:

```
int MessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);
```

Ее параметры интерпретируются следующим образом:

- ❑ `hWnd` — дескриптор родительского окна. Он принимает значение `NULL`, если родительского окна нет.
- ❑ `lpText` — указатель на строку, содержащую текст сообщения.
- ❑ `lpCaption` — указатель на строку, содержащую текст заголовка диалогового окна.
- ❑ `uType` — параметр содержит комбинацию флагов, задающих количество и типы кнопок в диалоговом окне, а также наличие заданной пиктограммы.

Более подробно интерпретация параметров функции `MessageBox` рассматривается в главе 7. Сейчас для нас важно лишь то, что эта функция создает диалоговое окно, относящееся к предопределенному в системе *оконному классу диалоговых окон*. А предопределенный класс диалоговых окон содержит свою собственную оконную процедуру, спрятанную в недрах Windows. Кроме этого, выполняя функцию `MessageBox`, система создает невидимый для программиста цикл обработки сообщений, который обслуживает созданное окно диалога.

¹ Microsoft Developer Network (MSDN) — это набор онлайновых и оффлайновых служб, предназначенных для оказания помощи разработчику в написании приложений с использованием продуктов и технологий фирмы Microsoft. Справочная система MSDN (MSDN Library), обычно устанавливаемая на компьютере вместе со средой Microsoft Visual Studio 6.0, содержит обширную информацию по программированию, в том числе и описание структур данных и функций Win32 API. В дальнейшем вместо термина *Справочная система MSDN* будет употребляться термин *MSDN*.

² Прототип (или предварительное объявление функции) задает тип возвращаемого значения, имя функции и список передаваемых параметров.

Вот почему в программе нет ни оконной процедуры, ни цикла обработки сообщений, и она претендует на титул самой короткой программы для Windows. Но, как уже говорилось, утилитарной пользы от этой суперкороткой программы не очень много.

Зато есть польза, и немалая, — гносеологическая! Так, мы уже знаем, что такое WinMain и как интерпретируются ее параметры. Это раз. Мы уже умеем пользоваться функцией MessageBox. Это два. Анализ текста программы дает также повод поговорить о стиле написания Windows-программ, а точнее, о так называемой *венгерской нотации*, которая была предложена венгерским программистом фирмы Microsoft Чарльзом Симони (Charles Simonyi). Суть этой нотации заключается в том, что имя переменной начинается с префикса, который содержит одну или несколько строчных букв. Этот префикс определяет тип переменной. За префиксом следует само имя, которое может состоять как из прописных, так и из строчных букв, но первая буква имени — всегда прописная. Например, идентификатор nMyVariable обозначает некоторую целую переменную. Основные префиксы венгерской нотации приведены в табл. 1.2.

Таблица 1.2. Префиксы имен переменных для указания типа данных

Префикс	Соответствующий тип данных
b	BOOL
by	BYTE
c	char
dw	DWORD
fn	Функция
h	Дескриптор
i	INT
l	LONG
lp	Дальний указатель
lpsz	Дальний указатель на строку, заканчивающуюся нулевым байтом
n	short или int
p	Указатель
pfn	Указатель на функцию
psz	Указатель на строку, заканчивающуюся нулевым байтом
pv	Указатель на тип void
sz	Строка, заканчивающаяся нулевым байтом
u	UINT
v	void
w	WORD
x	Короткое целое число, используемое в качестве координаты x
y	Короткое целое число, используемое в качестве координаты y

Теперь, уважаемый читатель, вам должно быть понятно, откуда берутся и что означают такие имена переменных, как lpText или uType. Более того, вы можете также придерживаться этой системы обозначений, придумывая имена для ваших собственных переменных. Но на последнем мы не настаиваем, так как вопрос стиля программирования либо решается индивидуально на основе личных вкусов и предпочтений, либо регламентируется правилами той фирмы, в которой работает программист.

Прежде чем перейти к рассмотрению «нормальной» Windows-программы, следует отметить, что в функциях Win32 может использоваться одна из следующих систем координат:

- экранные координаты (screen coordinates);*
- оконные координаты (window coordinates);*
- координаты клиентской области (client coordinates).*

На рис. 1.3 показаны главное окно программы на экране монитора, а также взаимоотношения между этими системами координат.

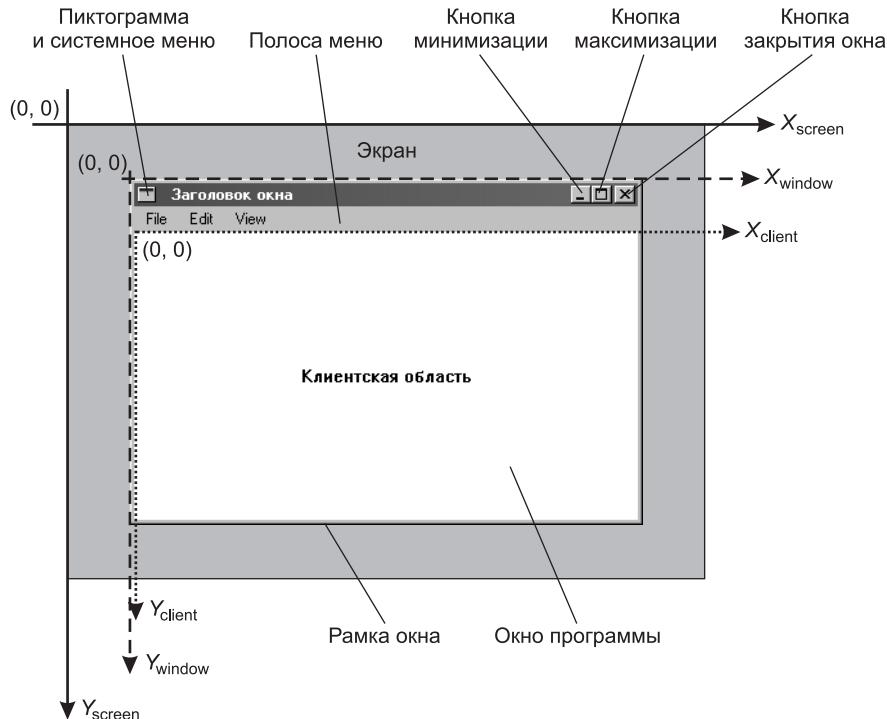


Рис. 1.3. Системы координат Windows

Кроме этого на рисунке показаны основные элементы главного окна программы, на которые мы будем ссылаться в дальнейшем изложении.

Обратите внимание на то, что в верхней части окна программы находится полоса заголовка окна. В левой части полосы заголовка всегда расположена маленькая картинка — *пиктограмма*. Щелчок мышью на изображении пиктограммы вызывает появление всплывающего *системного меню*. Оно обеспечивает доступ к стандартным функциям для окна приложения, например Move (Переместить), Size (Размер), Minimize (Свернуть), Maximize (Развернуть), Close (Закрыть). Последние три функции дублируются кнопками, расположенными в правой части полосы заголовка.

Ниже заголовка окна обычно размещается *полоса меню*, хотя можно создавать приложения, в которых меню отсутствует.

Еще ниже находится *клиентская область* окна. Это главная область вывода приложения. Ответственность за управление клиентской областью окна лежит на приложении.

Следует отметить, что иногда Windows-приложения создают на базе диалогового окна. В таких приложениях полоса заголовка окна может отсутствовать.

Программа «Hello, World!» — первый вариант

Полноценная программа для Win32 должна содержать как минимум две функции:

- `WinMain` — главную функцию, в которой создается основное окно программы и запускается цикл обработки сообщений;
- `WndProc` — оконную процедуру, обеспечивающую обработку сообщений для основного окна программы.

На некотором псевдоязыке каркас Windows-программы можно представить следующим образом:

```
WinMain (список аргументов)
{
    Подготовить и зарегистрировать класс окна с требуемыми характеристиками;
    Создать экземпляр окна зарегистрированного класса;

    Пока не произошло необходимое для выхода событие {
        Извлечь очередное сообщение из очереди сообщений;
        Передать его через Windows оконной функции;
    }
    Возврат из программы;
}

WndProc (список аргументов)
{
    Обработать полученное сообщение;
    Возврат;
}
```

Не будем отступать от традиции и покажем, как написать приложение, которое создает окно и выводит в нем строку «Hello, World!». Весь текст программы, приведенный в листинге 1.1, размещается в файле `Hello1.cpp`.

Листинг 1.1. Проект Hello1

```
///////////////
// Hello1.cpp
#include <windows.h>

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    HWND hMainWnd;
    char szClassName[] = "MyClass";
```

продолжение ↗

Листинг 1.1 (продолжение)

```

MSG msg;
WNDCLASSEX wc;

// Заполняем структуру класса окна
wc.cbSize      = sizeof(wc);
wc.style       = CS_HREDRAW | CS_VREDRAW;
wc.lpfnWndProc = WndProc;
wc.cbClsExtra  = 0;
wc.cbWndExtra  = 0;
wc.hInstance   = hInstance;
wc.hIcon        = LoadIcon(NULL, IDI_APPLICATION);
wc.hCursor      = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
wc.lpszMenuName = NULL;
wc.lpszClassName = szClassName;
wc.hIconSm      = LoadIcon(NULL, IDI_APPLICATION);

// Регистрируем класс окна
if (!RegisterClassEx(&wc)) {
    MessageBox(NULL, "Cannot register class", "Error", MB_OK);
    return 0;
}

// Создаем основное окно приложения
hMainWnd = CreateWindow(
    szClassName, "A Hello1 Application", WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
    (HWND)NULL, (HMENU)NULL,
    (HINSTANCE)hInstance, NULL
);

if (!hMainWnd) {
    MessageBox(NULL, "Cannot create main window", "Error", MB_OK);
    return 0;
}

// Показываем наше окно
ShowWindow(hMainWnd, nCmdShow);
// UpdateWindow(hMainWnd); // См. примечание в разделе "Отображение окна..."

// Выполняем цикл обработки сообщений до закрытия приложения
while (GetMessage(&msg, NULL, 0, 0)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}

return msg.wParam;
}

//=====================================================================
RESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    RECT rect;

    switch (uMsg)

```

```
{  
    case WM_PAINT:  
        hDC = BeginPaint(hWnd, &ps);  
  
        GetClientRect(hWnd, &rect);  
        DrawText(hDC, "Hello, World!", -1, &rect,  
            DT_SINGLELINE | DT_CENTER | DT_VCENTER );  
  
        EndPaint(hWnd, &ps);  
        break;  
  
    case WM_CLOSE:  
        DestroyWindow(hWnd);  
        break;  
  
    case WM_DESTROY:  
        PostQuitMessage(0);  
        break;  
  
    default:  
        return DefWindowProc(hWnd, uMsg, wParam, lParam);  
    }  
}  
////////////////////////////////////////////////////////////////
```

Создайте проект с именем Hello1, добавьте к нему указанный файл. После компиляции и запуска на выполнение на экране должно появиться окно программы, показанное на рис. 1.4.



Рис. 1.4. Окно программы Hello1

А вот теперь придется запастись терпением. Пояснения к программе Hello1 займут примерно в восемь раз больше страниц, чем сам код программы.

Файл исходного текста программы

В файле Hello1.cpp расположены только две функции: WinMain и WndProc. WinMain — это точка входа в программу. WndProc — это оконная процедура для главного окна

программы. Оконная процедура инкапсулирует код, отвечающий за ввод информации, обычно осуществляемый при помощи клавиатуры или мыши, а также за вывод информации на экран.

В Hello1.cpp отсутствуют инструкции для непосредственного вызова WndProc — оконная процедура вызывается только из Windows. Однако в WinMain имеется ссылка на WndProc (адрес оконной процедуры присваивается полю wc.lpfnWndProc), поэтому прототип функции WndProc объявлен в самом начале файла, еще до определения функции WinMain.

Обратите внимание на идентификаторы, написанные полностью прописными буквами. Эти идентификаторы задаются в заголовочных файлах Windows. Некоторые из них содержат двух- или трехбуквенный префикс, завершающийся символом подчеркивания, например CS_HREDRAW, IDI_APPLICATION, IDC_ARROW, WS_OVERLAPPEDWINDOW, CW_USEDEFAULT. Это просто числовые константы. Префикс обозначает основную категорию, к которой принадлежит константа. Некоторые из этих категорий приведены в табл. 1.3.

Таблица 1.3. Префиксы для числовых констант

Префикс	Категория
CS_	Опция стиля класса
CW_	Опция создания окна
DT_	Опция рисования текста
IDC_	Идентификатор предопределенного курсора
IDI_	Идентификатор предопределенной иконки (пиктограммы)
WM_	Сообщение окна
WS_	Стиль окна

Также следует обратить внимание на новые типы данных, специфичные для системы Windows. Некоторые из них были описаны в табл. 1.1.

Регистрация класса окна

Сразу после входа в функцию WinMain создается и регистрируется класс главного окна приложения. Для этого необходимо заполнить структуру типа WNDCLASSEX, а затем передать адрес этой структуры в виде аргумента функции RegisterClassEx.

Структура WNDCLASSEX имеет двенадцать полей:

```
typedef struct tagWNDCLASSEX {
    UINT cbSize;           // размер данной структуры в байтах
    UINT style;            // стиль класса окна
    WNDPROC lpfnWndProc;  // указатель на функцию окна (оконную процедуру)
    int cbClsExtra;        // число дополнительных байтов, которые должны
                          // быть распределены в конце структуры класса
    int cbWndExtra;        // число дополнительных байтов, которые должны
                          // быть распределены вслед за экземпляром окна
    HINSTANCE hInstance;  // дескриптор экземпляра приложения, в котором
                          // находится оконная процедура для этого класса
    HICON hIcon;           // дескриптор пиктограммы
    HCURSOR hCursor;       // дескриптор курсора
    HBRUSH hbrBackground; // дескриптор кисти, используемой для закраски
                          // фона окна
```

```

LPCTSTR lpszMenuName; // указатель на строку, содержащую имя меню,
// применяемого по умолчанию для этого класса
LPCTSTR lpszClassName; // указатель на строку, содержащую имя класса окна
HICON hIconSm; // дескриптор малой пиктограммы
} WNDCLASSEX;

```

ПРИМЕЧАНИЕ

Следует упомянуть о некоторых аспектах использованных в этом определении типов данных, а также о венгерской нотации имен полей.

Предфиксы LP и lp означают «длинный указатель» (long pointer), являющийся пережитком 16-разрядной Windows, в которой различались короткие 16-разрядные указатели и длинные 32-разрядные указатели. В Win 32 API все указатели имеют длину 32 разряда.

Предфикс lpfn означает «длинный указатель на функцию» (long pointer to a function). Префикс cb означает «счетчик байтов» (counter of bytes). А префикс hbr — это «дескриптор кисти» (handle to a brush).

Рассмотрим назначение полей структуры **WNDCLASSEX**.

Значение первого поля, **cbSize**, должно быть равно длине структуры.

Второе поле, **style**, может содержать в своем значении один или несколько стилей, перечисленных в табл. 1.4¹.

Таблица 1.4. Стили класса окна

Стиль	Описание
CS_GLOBALCLASS	Создать класс, доступный всем приложениям. Обычно этот стиль применяется для создания определяемых пользователем элементов управления в DLL
CS_HREDRAW	Перерисовывать все окно, если изменен размер по горизонтали
CS_NOCLOSE	Запретить команду Close в системном меню
CS_OWNDC	Выделить уникальный контекст устройства для каждого окна, созданного при помощи этого класса
CS_VREDRAW	Перерисовывать все окно, если изменен размер по вертикали

В заголовочных файлах Windows идентификаторы, начинающиеся с префикса **CS_**, задаются в виде 32-разрядной константы, в которой только один разряд установлен в единичное значение. Например, константа **CS_VREDRAW** задана как **0x0001**, а **CS_HREDRAW** — как **0x0002**. Подобные константы иногда называют *поразрядными флагами* (bit flags). Они могут объединяться с помощью операции ИЛИ языка C++.

В рассматриваемой программе используется комбинация стилей **CS_HREDRAW | CS_VREDRAW**. Это означает, что все окна этого класса должны целиком перерисовываться при изменении как горизонтального, так и вертикального размеров окна. Если попробовать изменить размеры окна A Hello1 Application, то можно увидеть, что строка текста переместится в новый центр окна. Механизм уведомления оконной процедуры об изменении размеров окна будет рассмотрен позже.

Третье поле, **lpfnWndProc**, содержит адрес оконной процедуры (в нашей программе это — **WndProc**).

¹ В таблице указаны наиболее употребительные стили. Полный список стилей см. в MSDN.

Назначение полей 4–6 представляется очевидным¹.

Седьмое поле, hIcon, содержит дескриптор пиктограммы, которая предназначена для этого класса окна. *Пиктограмма* (синонимы: *иконка*, *значок*) — это маленькая битовая картинка, которая появляется на панели задач Windows и в левой части заголовка окна. Значение hIcon обычно получают вызовом функции LoadIcon, которая имеет следующий прототип:

```
HICON LoadIcon(HINSTANCE hInstance, LPCTSTR lpIconName);
```

Эта функция загружает ресурс пиктограммы, заданный параметром lpIconName, из экземпляра приложения, указанного параметром hInstance². Функцию можно использовать также для загрузки одной из системных (предопределенных) пиктограмм, если передать первому аргументу значение NULL. В этом случае второй аргумент должен содержать константу, идентификатор которой начинается с префикса IDI_ («идентификатор значка» — ID for icon). Возможные значения второго аргумента для предопределенных пиктограмм приведены в табл. 1.5.

Таблица 1.5. Предопределенные идентификаторы пиктограмм

Значение	Описание
IDI_APPLICATION	Пиктограмма приложения по умолчанию
IDI_ASTERISK	То же, что и IDI_INFORMATION
IDI_ERROR	Пиктограмма в виде белого креста на фоне красного круга. Она используется в серьезных предупреждающих сообщениях
IDI_EXCLAMATION	То же, что и IDI_WARNING
IDI_HAND	То же, что и IDI_ERROR
IDI_INFORMATION	Пиктограмма «?», которая используется в информационных сообщениях
IDI_QUESTION	Пиктограмма «?»
IDI_WARNING	Пиктограмма «!», которая используется в предупреждающих сообщениях
IDI_WINLOGO	Логотип Windows

Поле hCursor содержит дескриптор курсора мыши, используемого приложением в клиентской области окна. Значение hCursor обычно получают с помощью вызова функции LoadCursor, имеющей следующий прототип:

```
HCURSOR LoadCursor(HINSTANCE hInstance, LPCTSTR lpCursorName);
```

Эта функция загружает ресурс курсора, заданный вторым параметром (lpCursorName), из экземпляра приложения, заданного первым параметром (hInstance)³.

Функцию можно также использовать для загрузки одного из системных (предопределенных) курсоров, если передать первому аргументу значение NULL. Значение второго аргумента при этом выбирается из табл. 1.6.

Таблица 1.6. Предопределенные идентификаторы курсора

Значение	Описание
IDC_APPSTARTING	Стандартная стрелка и малые песочные часы
IDC_ARROW	Стандартная стрелка

¹ Поля cbClsExtra и cbWndExtra используются крайне редко.

² Вопросы создания и использования пользовательских пиктограмм рассматриваются в главе 5.

³ Вопросы создания и использования пользовательских курсоров рассматриваются в главе 5.

Значение	Описание
IDC_CROSS	Перекрестье
IDC_HELP	Стрелка и вопросительный знак
IDC_IBEAM	Текстовый двутавр
IDC_NO	Перечеркнутый кружок
IDC_SIZEALL	Четырехконечная стрелка
IDC_SIZENESW	Двухконечная стрелка, указывающая на северо-восток и юго-запад
IDC_SIZENS	Двухконечная стрелка, указывающая на север и юг
IDC_SIZENWSE	Двухконечная стрелка, указывающая на северо-запад и юго-восток
IDC_SIZEWE	Двухконечная стрелка, указывающая на запад и восток
IDC_UPARROW	Вертикальная стрелка
IDC_WAIT	Песочные часы

Девятое поле, `hbrBackground`, содержит дескриптор кисти, используемой приложением для закраски фона в клиентской области окна. *Кисть (brush)* — это графический объект, который представляет собой шаблон пикселов различных цветов, используемый для закрашивания области. В Windows имеется несколько стандартных или предопределенных кистей. Вызов функции `GetStockObject` с аргументом `WHITE_BRUSH` возвращает дескриптор белой кисти. Так как возвращаемое значение имеет тип `HGDIOBJ`, то необходимо его преобразование к типу `HBRUSH`.

Десятое поле, `lpszMenuName`, указывает на строку, содержащую имя меню, применяемого по умолчанию для данного класса. Рассматриваемая программа не имеет меню, поэтому поле установлено в `NULL`.

Поле `lpszClassName` указывает на строку, содержащую имя класса. Это имя должно использоваться в параметре `lpClassName` функции `CreateWindow`.

Двенадцатое поле, `hIconSm`, содержит дескриптор малой пиктограммы, которая предназначена для использования в строке заголовка окон, созданных при помощи этого класса. Размеры пиктограммы должны быть 16×16 пикселов. Если поле равно `NULL`, то система ищет ресурс пиктограммы, указанный полем `hIcon`, чтобы сформировать малую пиктограмму из него.

Итак, с заполнением структуры класса окна мы разобрались. Осталось зарегистрировать подготовленный класс, вызвав функцию `RegisterClassEx`. Это расширенная версия¹ функции `RegisterClass` из предыдущих версий Windows. В то же время можно пользоваться и функцией `RegisterClass`, передавая ей адрес структуры `WNDCLASS` (а не `WNDCLASSEX`).

Создание окна

Если регистрация класса окна прошла успешно, то следующий этап — создание окна. Для этого вызывается функция `CreateWindow`, прототип которой приведен ниже:

```
HWND CreateWindow(
    LPCTSTR lpClassName, // имя зарегистрированного класса
    LPCTSTR lpWindowName, // имя окна
    DWORD dwStyle, // стиль окна
    int x, // горизонтальная позиция
```

¹ На это указывает окончание имени `Ex`, то есть *extended* — расширенный.

```

int y,           // вертикальная позиция
int nWidth,      // ширина окна
int nHeight,     // высота окна
HWND hWndParent, // дескриптор родительского окна
HMENU hMenu,      // дескриптор меню окна или идентификатор элемента
//управления
HINSTANCE hInstance, // дескриптор экземпляра приложения
LPVOID lParam    // указатель на данные, передаваемые в сообщении
//WM_CREATE
);

```

Рассматривая назначение параметров, не будем забывать, что функция `CreateWindow` позволяет создавать не только основное окно приложения, но и любые другие окна, включая окна предопределенных классов, с помощью которых реализуются стандартные элементы управления Windows.

Первый параметр, `lpClassName`, — это указатель на строку, содержащую допустимое имя класса окна. Таким именем может быть либо имя класса, зарегистрированного ранее при помощи функции `RegisterClassEx` или `RegisterClass`, либо имя одного из предопределенных классов, перечисленных в табл. 1.7.

Таблица 1.7. Предопределенные классы окон

Класс	Описание	Префикс в обозначении стилей
BUTTON	Прямоугольное окно кнопки, группы, флагка, переключателя или пиктограммы	BS_
COMBOBOX	Элемент управления, объединяющий элементы LISTBOX и EDIT. В поле редактирования отображается выбранная строка из LISTBOX	CBS_
EDIT	Прямоугольное окно (поле редактирования), предназначенное для ввода текста с клавиатуры	ES_
LISTBOX	Прямоугольное окно со списком строк, из которого пользователь может выбрать любую строку	LBS_
MDICLIENT	Клиентское окно многодокументного интерфейса. Это окно получает сообщения, которые управляют дочерними окнами в MDI-приложении	
RICHEDIT	Элемент управления Rich Edit версии 1.0. В дополнение к возможностям элемента EDIT позволяет редактировать текст с разными шрифтами и стилями	ES_
RICHEDIT_CLASS	Элемент управления Rich Edit версии 2.0 или 3.0 ¹ . Усовершенствованная версия элемента RICHEDIT с дополнительными возможностями	ES_
SCROLLBAR	Элемент управления линейкой прокрутки	SBS_
STATIC	Элемент управления статическим текстом. Применяется для размещения в окне текста или рамок	SS_

Второй параметр, `lpWindowName`, — это указатель на строку, содержащую имя окна. Место отображения имени зависит от вида окна. Например, для главного окна приложения оно выводится как заголовок окна (см. рис. 1.3), а для окна предопределенного класса BUTTON размещается по центру кнопки.

¹ В зависимости от версии Windows, установленной на компьютере.

Третий параметр, `dwStyle`, позволяет указывать стиль окна, состоящий из значений, указанных в табл. 1.8. Эти константы могут объединяться в значении параметра с помощью побитовой операции ИЛИ. Для окон предопределенных классов дополнительно могут быть добавлены стили, характерные для данного элемента управления. Идентификаторы таких стилей начинаются с префикса, указанного в табл. 1.7¹.

Таблица 1.8. Стили окна

Обозначение стиля	Описание
<code>WS_BORDER</code>	Создать окно с рамкой в виде тонкой линии
<code>WS_CAPTION</code>	Создать окно, которое имеет область заголовка (включает стиль <code>WS_BORDER</code>)
<code>WS_CHILD</code>	Создать дочернее окно. Окно этого стиля не может иметь полосу меню и не может иметь стиль <code>WS_POPUP</code>
<code>WS_CLIPCHILDREN</code>	Исключить перерисовку дочерних окон, принадлежащих данному родительскому окну
<code>WS_CLIPSIBLINGS</code>	Исключить перерисовку соседних дочерних окон при перерисовке данного дочернего окна
<code>WS_DLGFRADE</code>	Создать окно, имеющее рамку со стилем, типичным для диалоговых окон. Окно этого стиля не может иметь область заголовка
<code>WS_GROUP</code>	Считать данное окно первым в группе элементов управления (обычно группируются переключатели)
<code>WS_HSCROLL</code>	Создать окно с горизонтальной линейкой прокрутки
<code>WS_MAXIMIZE</code>	Создать окно, которое первоначально является развернутым
<code>WS_MAXIMIZEBOX</code>	Создать окно с кнопкой развертывания
<code>WS_MINIMIZE</code>	Создать окно, которое первоначально является свернутым
<code>WS_MINIMIZEBOX</code>	Создать окно с кнопкой свертывания
<code>WS_OVERLAPPED</code>	Создать перекрывающееся окно. Окно этого стиля имеет область заголовка и рамку
<code>WS_OVERLAPPEDWINDOW</code>	Сочетание стилей <code>WS_OVERLAPPED</code> , <code>WS_CAPTION</code> , <code>WS_SYSMENU</code> , <code>WS_THICKFRAME</code> , <code>WS_MINIMIZEBOX</code> и <code>WS_MAXIMIZEBOX</code>
<code>WS_POPUP</code>	Создать всплывающее окно. Этот стиль не может использоваться совместно со стилем <code>WS_CHILD</code>
<code>WS_POPUPWINDOW</code>	Сочетание стилей <code>WS_BORDER</code> , <code>WS_POPUP</code> и <code>WS_SYSMENU</code> . Чтобы сделать системное меню видимым, необходимо добавить стиль <code>WS_CAPTION</code>
<code>WS_SYSMENU</code>	Создать окно с системным меню в области заголовка
<code>WS_TABSTOP</code>	Стиль указывает, что на окне может остановиться фокус ввода, перемещаемый при помощи клавиши TAB
<code>WS_THICKFRAME</code>	Создать окно с рамкой, которая позволяет изменять его размеры
<code>WS_VISIBLE</code>	Создать окно, которое сразу же является видимым
<code>WS_VSCROLL</code>	Создать окно с вертикальной линейкой прокрутки

В рассматриваемой программе используется стандартный стиль для главного окна приложения — `WS_OVERLAPPEDWINDOW`. Это обычное перекрывающееся окно с заголовком, системным меню, расположенным в левой части строки заголовка,

¹ Стили для элементов управления здесь не приводятся. При необходимости эту информацию можно найти в MSDN.

кнопками для сворачивания, разворачивания и закрытия окна справа на строке заголовка и «толстой» рамкой, позволяющей изменять размеры окна.

Четвертый параметр, *x*, задает горизонтальную позицию левого верхнего угла окна. Для главного окна позиции *x* и *y* определяются в экранных координатах, а для дочерних окон и элементов управления координаты отсчитываются относительно левого верхнего угла родительского окна.

Если позиция *x* не важна, то можно установить значение **CW_USEDEFAULT**. В этом случае Windows использует значения *x* и *y* по умолчанию. Это означает, что система располагает следующие друг за другом перекрывающиеся окна, равномерно отступая по горизонтали и вертикали от левого верхнего угла экрана.

Пятый параметр, *y*, задает вертикальную позицию левого верхнего угла окна. Если параметр *x* имеет значение **CW_USEDEFAULT**, то значение параметра *y* игнорируется.

Шестой параметр, *nWidth*, задает ширину окна в пикселях. Если ему присвоено значение **CW_USEDEFAULT**, то система будет использовать значения *nWidth* и *nHeight* по умолчанию.

Седьмой параметр, *nHeight*, задает высоту окна в пикселях. Если параметр *nWidth* установлен в **CW_USEDEFAULT**, то значение *nHeight* игнорируется.

Восьмой параметр, *hWndParent*, в рассматриваемой программе имеет значение **NULL**, поскольку у главного окна программы отсутствует родительское окно. Заметим, что, если между двумя окнами существует связь типа *родительское—дочернее*, дочернее окно всегда появляется только на поверхности родительского.

Девятый параметр, *hMenu*, содержит дескриптор меню окна или идентификатор элемента управления. Интерпретация значения параметра зависит от вида окна.

Если приложение использует меню, определенное в классе окна (поле *lpszMenuName* в структуре *WNDCLASSEX*), то параметру *hMenu* необходимо передать значение **NULL**.

Если создаваемое окно относится к элементу управления, то параметру *hMenu* передается целочисленное значение, которое далее используется как идентификатор созданного элемента. Этот идентификатор, например, будет содержаться в сообщениях *WM_COMMAND*, поступающих от элемента управления.

Десятому параметру, *hInstance*, должен быть присвоен дескриптор экземпляра приложения, переданный программе в качестве аргумента функции *WinMain*.

Последний параметр, *lParam*, в рассматриваемом примере тоже имеет значение **NULL**. При необходимости он может быть использован в качестве указателя на некие дополнительные данные, передаваемые окну в момент его создания с помощью сообщения *WM_CREATE*.

Итак, функция *CreateWindow* вызвана. Отработав, она возвращает дескриптор созданного окна. Если по какой-то причине создать окно не удалось, то функция возвращает значение **NULL**. Поэтому рекомендуется проверять возвращаемое функцией значение и в случае неудачи выдавать соответствующее сообщение, используя уже знакомую функцию *MessageBox*.

Использование функции **CreateWindowEx**

Для создания окна вместо функции *CreateWindow* может быть вызвана функция *CreateWindowEx*, прототип которой приведен ниже:

```
HWND CreateWindowEx(DWORD dwExStyle, LPCTSTR lpClassName,  
LPCTSTR lpWindowName, DWORD dwStyle, int x, int y, int nWidth,
```

```
int nHeight, HWND hWndParent, HMENU hMenu, HINSTANCE hInstance,
LPCVOID lParam);
```

Все параметры этой функции, кроме первого, имеют тот же смысл, что и у функции `CreateWindow`. Первый параметр, `dwExStyle`, задает расширенный стиль окна, применяемый совместно со стилем, определенным в параметре `dwStyle`. Например, в качестве расширенного стиля можно задать один или несколько флагов, приведенных в табл. 1.9. Полный список расширенных стилей можно найти в MSDN.

Таблица 1.9. Расширенные стили окна

Стиль	Описание
<code>WS_EX_ACCEPTFILES</code>	Создать окно, которое принимает перетаскиваемые файлы
<code>WS_EX_CLIENTEDGE</code>	Рамка окна имеет утопленный край
<code>WS_EX_CONTROLPARENT</code>	Разрешить пользователю перемещаться по дочерним окнам с помощью клавиши <code>Tab</code>
<code>WS_EX_MDICHILD</code>	Создать дочернее окно многодокументного интерфейса
<code>WS_EX_STATICEDGE</code>	Создать окно с трехмерной рамкой. Этот стиль предназначен для элементов, которые не принимают ввод от пользователя
<code>WS_EX_TOOLWINDOW</code>	Создать окно с инструментами, предназначенное для реализации плавающих панелей инструментов
<code>WS_EX_TRANSPARENT</code>	Создать прозрачное окно. Любые окна того же уровня, накрываемые этим окном, получат сообщение <code>WM_PAINT</code> в первую очередь, тем самым создавая эффект прозрачности
<code>WS_EX_WINDOWEDGE</code>	Создать окно, имеющее рамку с активизированным краем

Функция `CreateWindowEx` так же, как и функция `CreateWindow`, может быть применена для создания любых окон, в том числе и окон элементов управления Windows. Некоторые из расширенных стилей окна предназначены как раз для дочерних окон элементов управления¹.

Отображение окна на экране

Для отображения на экране созданного окна вызывается функция `ShowWindow`, имеющая следующий прототип:

```
BOOL ShowWindow(HWND hWnd, int nCmdShow);
```

Первым параметром функции является дескриптор окна, а второй параметр определяет, в каком виде будет показано окно.

При начальном отображении главного окна рекомендуется присваивать второму параметру то значение, которое передается приложению через параметр `nCmdShow` функции `WinMain`.

При последующих отображениях можно использовать любое из значений, приведенных в табл. 1.10².

¹ Базовые элементы управления рассматриваются в главе 7, элементы управления общего пользования — в главе 8.

² В таблице приведены наиболее употребляемые значения. Полный список значений см. в MSDN.

Таблица 1.10. Значения параметра nCmdShow функции ShowWindow

Значение	Описание
SW_HIDE	Скрыть окно
SW_MAXIMIZE	Развернуть окно
SW_MINIMIZE	Свернуть окно
SW_SHOW	Активизировать окно и показать в его текущих размерах и позиции
SW_SHOWNOACTIVATE	Окно не выводится, а на панели задач появляются его имя и пиктограмма

ПРИМЕЧАНИЕ

Во всех книгах, посвященных программированию с Win32 API, а также в справочных материалах MSDN рекомендуется после вызова функции ShowWindow вызвать функцию UpdateWindow, которая посылает оконной процедуре сообщение WM_PAINT, заставляющее окно перерисовать свою клиентскую область. Однако эксперименты показали, что в этом нет никакой необходимости. Отладочная трассировка сообщений выявила, что результатом работы функции ShowWindow является генерация сообщений WM_SIZE и WM_MOVE, а обрабатывая сообщение WM_SIZE, система всегда генерирует сообщение WM_PAINT. Поэтому вызов функции UpdateWindow в примере закомментирован.

Обработка сообщений

После вывода окна на экран программа должна подготовить себя для получения информации от пользователя. Эта информация обычно вводится с помощью клавиатуры или мыши. Windows поддерживает «очередь сообщений» (*message queue*) для каждой программы, работающей в системе. Любое действие пользователя система Windows преобразует в «сообщение», которое помещается в указанную очередь.

Программа извлекает сообщения из очереди, выполняя блок команд, известный как «цикл обработки сообщений» (*message loop*):

```
while (GetMessage(&msg, NULL, 0, 0)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

return msg.wParam;

В этом фрагменте кода переменная `msg` — это структура типа `MSG`, которая определена в заголовочных файлах Windows следующим образом:

```
typedef struct tagMSG {
    HWND    hwnd;        // дескриптор окна, которому адресовано сообщение
    UINT    message;     // номер (идентификатор) сообщения
    WPARAM  wParam;      // параметр сообщения wParam
    LPARAM  lParam;     // параметр сообщения lParam
    DWORD   time;        // время отправки сообщения
    POINT   pt;          // позиция курсора (в экранных координатах) в момент
                         // отправки сообщения
} MSG;
```

Интерпретация параметров `wParam` и `lParam` зависит от номера сообщения.

Тип данных `POINT` используется для представления точки парой ее координат:

```
typedef struct tagPoint {
    LONG x;
    LONG y;
} POINT;
```

Извлечение очередного сообщения осуществляется с помощью функции `GetMessage`, имеющей прототип:

```
BOOL GetMessage(LPMSG lpMsg, HWND hWnd, UINT wMsgFilterMin, UINT wMsgFilterMax)
```

- ❑ Параметр `lpMsg` задает адрес структуры типа `MSG`, в которую помещается выбранное сообщение.
- ❑ Параметр `hWnd` содержит дескриптор окна, принимающего сообщение. Обычно значение этого параметра равно `NULL`, что позволяет выбирать сообщения для любого окна приложения.
- ❑ Параметр `wMsgFilterMin` указывает минимальный номер принимаемого сообщения. Обычно он имеет нулевое значение.
- ❑ Параметр `wMsgFilterMax` задает максимальный номер принимаемого сообщения и тоже обычно имеет нулевое значение. Если оба последних параметра равны нулю, то функция выбирает из очереди *любое* очередное сообщение.

Функция `GetMessage` возвращает значение `TRUE` при извлечении любого сообщения, кроме одного — `WM_QUIT`. Получив сообщение `WM_QUIT`, функция возвращает значение `FALSE`. В результате этого происходит немедленный выход из цикла, и приложение завершает работу, возвращая операционной системе код возврата `msg.wParam`.

Но какова дальнейшая судьба принятого сообщения?

В теле цикла обработки сообщений можно увидеть вызов двух функций: `TranslateMessage` и `DispatchMessage`.

Строго говоря, вызов `TranslateMessage` нужен только в тех приложениях, которые должны обрабатывать ввод данных с клавиатуры. Дело в том, что для обеспечения независимости от аппаратных платформ и различных национальных раскладок клавиатуры в Windows реализована двухуровневая схема обработки сообщений от символьных клавиш. Сначала система генерирует сообщения о так называемых *виртуальных клавишиах*, например: сообщение `WM_KEYDOWN` — когда клавиша нажимается, и сообщение `WM_KEYUP` — когда клавиша отпускается. В сообщении `WM_KEYDOWN` содержится также информация о так называемом *скан-коде*¹ нажатой клавиши.

Функция `TranslateMessage` преобразует пару сообщений, `WM_KEYDOWN` и `WM_KEYUP`, в сообщение `WM_CHAR`, которое содержит код символа (`wParam`) в виде значения типа `TCHAR`². Сообщение `WM_CHAR` помещается в очередь, а на следующей итерации цикла функция `GetMessage` извлекает его для последующей обработки.

Наконец, функция `DispatchMessage` передает структуру `msg` обратно в Windows. Очередной ход — за Windows. Windows отправляет сообщение для его обработки соответствующей оконной процедуре, вызывая ее как *функцию обратного вызова*.

¹ Код, идентифицирующий нажатую клавишу и зависящий от аппаратной платформы.

² Тип `TCHAR` интерпретируется как `char`, если макрос `UNICODE` не определен. В ином случае он интерпретируется как `wchar_t` (Unicode-символ).

Следует обратить внимание на то, что сообщение передается оконной процедуре именно того окна, которому оно было адресовано. Это может быть или `WndProc` (если сообщение адресовано главному окну), или оконная процедура некоторого диалогового окна, или оконная процедура одного из стандартных элементов управления, спрятанная в недрах Windows (то есть в DLL).

После возврата из оконной процедуры Windows передает управление оператору, указанному за вызовом `DispatchMessage`, то есть на начало цикла `while`, и работа цикла продолжается.

Как уже говорилось ранее, цикл обработки сообщений функционирует до тех пор, пока из очереди не поступит сообщение `WM_QUIT`. Появление этого сообщения вызывает немедленный выход из цикла и прекращение работы программы.

Оконная процедура

Реальная работа приложения осуществляется в оконной процедуре (`window procedure`). Оконная процедура определяет то, что выводится в клиентскую область окна, и то, как окну реагировать на пользовательский ввод.

Заголовок оконной функции всегда имеет следующий вид:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
```

Четыре параметра оконной процедуры идентичны первым четырем полям структуры `MSG`. Первый параметр функции содержит дескриптор окна, получающего сообщение. Во втором параметре указывается идентификатор сообщения. Для системных сообщений в Win32 зарезервированы номера от 0 до 1024. Третий и четвертый параметры содержат дополнительную информацию, которая распознается системой в зависимости от типа полученного сообщения.

Вместо имени `WndProc` можно использовать любое другое имя, но оно должно совпадать со значением поля `lpfnWndProc` структуры `wc`.

В рассматриваемой программе имеется только одно окно класса `MyClass`. Дескриптор окна `hMainWnd` получен вызовом функции `CreateWindow`. Поэтому при вызове оконной процедуры ее параметру `hWnd` всегда передается значение `hMainWnd`. Если же в программе создается несколько окон на основе одного и того же класса окна (и, следовательно, имеющих одну и ту же оконную процедуру), тогда параметру `hWnd` будет передаваться дескриптор конкретного окна, получающего сообщение.

Обычно программисты используют оператор `switch` для определения того, какое сообщение получено и как его обрабатывать. Если сообщение обрабатывается, то оконная процедура обязана вернуть нулевое значение. Все сообщения, не обрабатываемые оконной процедурой, должны передаваться системной функции `DefWindowProc`. В этом случае оконная процедура должна вернуть то значение, которое возвращает `DefWindowProc`.

Таким образом, программист пишет код только для тех сообщений, которые нуждаются в нестандартной обработке. В программе `Hello1` таких сообщений всего три: `WM_PAINT`, `WM_CLOSE` и `WM_DESTROY`.

Обработка сообщения `WM_PAINT`

Обработка сообщения `WM_PAINT` крайне важна для программирования под Windows. Это сообщение уведомляет программу, что часть или вся клиентская область

окна недействительна (*invalid*) и ее следует перерисовать. Во второй главе тема обработки сообщения WM_PAINT будет рассмотрена более подробно, поэтому здесь излагается тот минимум, который необходим для понимания работы программы Hello1.

В каких ситуациях клиентская область окна становится недействительной? Очевидно, что при создании окна недействительна вся его клиентская область, поскольку в ней еще ничего не нарисовано. Это раз.

Если вы меняете размеры окна, клиентская область также объявляется системой недействительной, что обусловлено комбинацией стилей CS_HREDRAW | CS_VREDRAW в поле *style* структуры класса окна. Это два.

Когда вы минимизируете окно программы Hello1, сворачивая его на панель задач, а затем снова разворачиваете до начального размера, то Windows объявляет клиентскую область окна недействительной. Это три.

Если вы перемещаете окна так, что они перекрываются, а затем закрытая часть окна вновь открывается, то Windows помечает требующую восстановления клиентскую область как недействительную. Это четыре.

Можно было бы продолжить, но оставим подробности для второй главы.

Во всех перечисленных случаях, кроме первого, операционная система автоматически помечает клиентскую область окна как недействительную, что влечет за собой посылку сообщения WM_PAINT. В первом случае, когда окно только что было создано, аналогичный эффект достигается вызовом функции ShowWindow. В процессе ее выполнения генерируются сообщения WM_SIZE и WM_MOVE, а обрабатывая WM_SIZE, система автоматически генерирует сообщение WM_PAINT.

Завершив этот маленький теоретический экскурс, вернемся к анализу нашего кода.

Обработку сообщения WM_PAINT рекомендуется всегда начинать с вызова функции BeginPaint:

```
hDC = BeginPaint(hWnd, &ps);
```

Первый параметр функции содержит дескриптор окна, полученный через аргумент оконной процедуры, а второй — адрес структуры ps типа PAINTSTRUCT. Поля этой структуры, заполняемые в результате выполнения функции BeginPaint, в дальнейшем используются операционной системой¹. Для нас сейчас более важно возвращаемое функцией значение — это дескриптор так называемого контекста устройства.

Контекст устройства (*device context*) описывает физическое устройство вывода информации, например дисплей или принтер. Этот важнейший объект графической подсистемы Windows рассматривается подробно во второй главе. Сейчас достаточно понимания того, что контекст устройства — это некоторая внутренняя структура данных, сохраняющая часто используемые графические атрибуты, такие как цвет фона, перо, кисть, шрифт и им подобные параметры. Эти атрибуты используются при вызове всех рисующих функций, получающих дескриптор hDC в качестве параметра.

Прежде чем перейти к следующей строке программного текста, отметим два побочных эффекта выполнения функции BeginPaint.

¹ Более подробно структура PAINTSTRUCT рассматривается в главе 2.

Во-первых, обрабатывая вызов `BeginPaint`, система Windows обновляет фон клиентской области, если обновляемый регион помечен для стирания¹. По умолчанию для этого используется кисть, заданная в поле `hbrBackground` структуры `WNDCLASSEX`.

Во-вторых, вызов функции `BeginPaint` делает всю клиентскую область *действительной (valid)*, то есть не требующей перерисовки. Это предотвращает повторную генерацию системой сообщения `WM_PAINT` до тех пор, пока вновь не произойдет одно из событий, требующих перерисовки окна.

После вызова `BeginPaint` в рассматриваемой программе следует вызов функции `GetClientRect`, предназначеннной для получения размеров клиентской области окна:

```
GetClientRect(hWnd, &rect);
```

Результат работы функции помещается в переменную `rect` типа `RECT`. Структура `RECT`, описывающая прямоугольник (*rectangle*), определена в заголовочных файлах Windows следующим образом:

```
typedef struct tagRECT {
    LONG left;
    LONG top;
    LONG right;
    LONG bottom;
} RECT;
```

Поля этой структуры задают координаты левого верхнего угла (`left, top`) и правого нижнего угла (`right, bottom`) прямоугольника.

После выполнения функции `GetClientRect` поля `left` и `top` всегда получают нулевое значение, а поля `right` и `bottom` содержат ширину и высоту клиентской области в пикселях. Информация о размерах клиентской области, сохраненная в переменной `rect`, используется далее в «рисующей» функции `DrawText`.

В нашей программе все «рисование» сводится к выводу текста «Hello, World!» при помощи функции `DrawText`, имеющей следующий прототип:

```
BOOL DrawText(
    HDC hdc,           // дескриптор контекста устройства
    LPCTSTR lpString, // указатель на символьную строку
    int nCount,        // длина текста
    LPRECT lpRect,     // указатель на ограничивающий прямоугольник
    UINT uFormat       // флаги форматирования текста
);
```

Функция² выводит текст из строки `lpString` в прямоугольную область, заданную структурой типа `RECT`, используя метод форматирования, заданный параметром `uFormat`. Количество символов в выводимой строке задается параметром `nCount`. Если установить значение `nCount` в `-1`, то система сама определит длину строки `lpString` по завершающему нулевому символу. Но в этом случае программист должен позаботиться о том, чтобы строка действительно завершалась нулевым байтом.

На месте последнего параметра функции задан набор флагов `DT_SINGLELINE | DT_CENTER | DT_VCENTER`, значения которых определяются в заголовочных файлах

¹ Информация об этом факте хранится в одном из полей структуры `ps`.

² Здесь приводятся минимальные сведения о работе функции `DrawText`. Подробно эта функция разбирается в главе 2.

Windows. Флаги показывают, что текст будет выводиться в одну строку, по центру относительно горизонтали и вертикали внутри прямоугольной области, заданной четвертым параметром. Благодаря этому текст «Hello, World!» выводится в центре клиентской области.

Когда клиентская область становится недействительной (например, при изменении размеров окна), WndProc получает новое сообщение WM_PAINT. Обрабатывая его, программа вновь вызывает функцию GetClientRect и поэтому рисует текст опять в центре окна.

После окончания работы с графическими функциями в блоке обработки сообщения WM_PAINT необходимо освободить полученный контекст устройства с помощью функции EndPaint.

Следует отметить, что при выводе текста используются текущие установки для шрифта, цвета фона и цвета текста. О том, как управлять этими атрибутами, будет рассказано во второй главе. Для тех, кто хочет экспериментировать с цветами текста и фона немедленно, ниже приводится краткая справка.

Атрибуты цвета и фона для выводимого текста

В приложении может использоваться функция SetTextColor для установки цвета текста (*text color*) в клиентской области окна. Функция SetBkColor используется для установки цвета фона графического элемента (*background color*), то есть цвета, который отображается позади каждого символа. Еще можно регулировать режим смешивания фона (*background mix mode*), вызывая функцию SetBkMode. Ее второй параметр может принимать значение OPAQUE¹, если цвет фона графического элемента выводится поверх существующего *фона окна*, прежде чем будет выведен символ. Также можно использовать значение TRANSPARENT², при котором цвет фона графического элемента игнорируется, а символ выводится на существующем фоне.

Если эти функции не вызывались, то в контексте устройства будут использованы значения по умолчанию, приведенные в табл. 1.11.

Таблица 1.11. Атрибуты цвета и фона текста по умолчанию

Атрибут	Значение по умолчанию
Цвет текста	Черный
Цвет фона графического элемента	Белый
Режим смешивания фона	OPAQUE

Именно эти параметры и использовались при работе программы Hello1.

Обработка сообщений WM_CLOSE и WM_DESTROY

Сообщение WM_CLOSE появляется, когда пользователь щелкает мышью на кнопке закрытия окна или нажимает комбинацию клавиш Alt+F4. Вообще, обработка этого сообщения не обязательна: если она отсутствует, то функция DefWindowProc вызовет по умолчанию функцию DestroyWindow. Но если вы хотите предусмотреть вывод каких-либо предупреждающих сообщений типа «А вы точно уверены, что

¹ Непрозрачный режим смешивания фона.

² Прозрачный режим смешивания фона.

хотите это сделать?», то данное место — самое подходящее. Ибо окно еще не разрушено, а вот после сообщения WM_DESTROY окна на экране уже нет.

Функция DestroyWindow, оправдывая свое имя, разрушает указанное в ее параметре окно. Для этого она посыпает окну сообщение WM_DESTROY. Если у данного окна есть дочерние окна, то функция посыпает WM_DESTROY сначала ему, а потом уже дочерним окнам. Функция завершает свою работу только после уничтожения всех дочерних окон.

Когда главное окно приложения получает сообщение WM_DESTROY, то оконная процедура должна позаботиться о том, чтобы приложение корректно «покинуло сцену». Для этого вызывается функция PostQuitMessage, посылающая сообщение WM_QUIT. Ну а что происходит в цикле обработки сообщений при появлении WM_QUIT, вы уже знаете.

Разрешите вас поздравить: мы завершили анализ текста программы Hello1 — первого варианта приложения «Hello, World!».

Как, будет еще и второй вариант?.. — Не волнуйтесь, пояснения к нему будут гораздо короче.

Программа «Hello, World!» — второй вариант

Программу «Hello, World!» часто используют как каркас нового Windows-приложения. Иными словами, начало работы над новой программой обычно происходит по такому сценарию: вы создаете новый проект, открываете новый пустой файл для функции WinMain, копируете в него содержимое файла Hello1.cpp¹, а потом вносите необходимые изменения и дополнения.

Но такой способ работы, мне кажется, должен оскорблять чувства программиста, пишущего на C++. Почему бы не воспользоваться известными средствами языка, позволяющими реализовать повторное использование кода гораздо более изящным способом? Вы уже догадались, что речь идет о программировании с классами? — Да, конечно.

Здесь предлагается одна из возможных реализаций этой идеи. Класс KWnd, представленный ниже, инкапсулирует код, отвечающий за регистрацию оконного класса, создание окна и показ его на экране.

Соблюдая традицию [9], будем размещать код для каждого нового класса в двух файлах: интерфейс — в файле с расширением .h, а реализацию — в файле с расширением .cpp. Таким образом, программа реализуется как многофайловый проект, приведенный в листинге 1.2².

Листинг 1.2. Проект Hello2

```
/////////////////////////////
// KWnd.h
#include <windows.h>

class KWnd {
```

¹ Не писать же все заново!

² Если у вас есть вопросы по созданию многофайловых проектов, то обратитесь к приложению 1.

```
public:
    KWnd(LPCTSTR windowName, HINSTANCE hInst, int cmdShow,
          LRESULT (WINAPI *pWndProc)(HWND,UINT,WPARAM,LPARAM),
          LPCTSTR menuName = NULL,
          int x = CW_USEDEFAULT, int y = 0,
          int width = CW_USEDEFAULT, int height = 0,
          UINT classStyle = CS_HREDRAW | CS_VREDRAW,
          DWORD windowStyle = WS_OVERLAPPEDWINDOW,
          HWND hParent = NULL);

    HWND GetHWnd() { return hWnd; }
protected:
    HWND hWnd;
    WNDCLASSEX wc;
};

////////////////////////////////////////////////////////////////
// KWnd.cpp
#include "KWnd.h"

KWnd::KWnd(LPCTSTR windowName, HINSTANCE hInst, int cmdShow,
           LRESULT (WINAPI *pWndProc)(HWND,UINT,WPARAM,LPARAM),
           LPCTSTR menuName, int x, int y, int width, int height,
           UINT classStyle, DWORD windowStyle, HWND hParent)
{
    char szClassName[] = "KWndClass";

    wc.cbSize      = sizeof(wc);
    wc.style       = classStyle;
    wc.lpfnWndProc = pWndProc;
    wc.cbClsExtra  = 0;
    wc.cbWndExtra   = 0;
    wc.hInstance   = hInst;
    wc.hIcon        = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor      = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = menuName;
    wc.lpszClassName = szClassName;
    wc.hIconSm     = LoadIcon(NULL, IDI_APPLICATION);

    // Регистрируем класс окна
    if (!RegisterClassEx(&wc)) {
        char msg[100] = "Cannot register class: ";
        strcat(msg, szClassName);
        MessageBox(NULL, msg, "Error", MB_OK);
        return;
    }

    // Создаем окно
    hWnd = CreateWindow(szClassName, windowName, windowStyle,
                       x, y, width, height, hParent, (HMENU)NULL, hInst, NULL);

    if (!hWnd) {
        char text[100] = "Cannot create window: ";
        strcat(text, windowName);
        MessageBox(NULL, text, "Error", MB_OK);
        return;
```

продолжение ↗

Листинг 1.2 (продолжение)

```
}

// Показываем окно
ShowWindow(hWnd, cmdShow);
}

///////////////
// Hello2.cpp
#include <windows.h>
#include "KWnd.h"

LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    int nCmdShow;
    MSG msg;

    KWnd mainWnd("A Hello2 application", hInstance, nCmdShow, WndProc);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}
=====

LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    RECT rect;
    int userReply;

    switch (uMsg)
    {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &ps);

        GetClientRect(hWnd, &rect);
        DrawText(hDC, "Hello, World!", -1, &rect,
            DT_SINGLELINE | DT_CENTER | DT_VCENTER );

        EndPaint(hWnd, &ps);
        break;

    case WM_CLOSE:
        userReply = MessageBox(hWnd, "А вы уверены в своем желании закрыть приложение?",
            "", MB_YESNO | MB_ICONQUESTION);
        if (IDYES == userReply)
            DestroyWindow(hWnd);
        break;

    case WM_DESTROY:
        PostQuitMessage(0);
        break;

    default:
```

```
        return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }

    return 0;
}
////////////////////////////////////////////////////////////////
```

Обратите внимание на следующие моменты предложенного решения:

- ❑ Заголовок конструктора класса `KWnd` содержит 12 параметров. Первые четыре параметра являются обязательными, а остальные имеют значения по умолчанию.
- ❑ Четвертый параметр является указателем на функцию, принимающую четыре аргумента и возвращающую значение типа `LRESULT`. Здесь обязателен спецификатор `WINAPI` или эквивалентный ему спецификатор `__stdcall`.
- ❑ Код, отвечающий за подготовку и создание окна, размещен в теле конструктора класса `KWnd`.
- ❑ В теле функции `WinMain` определяется объект `mainWnd` класса `KWnd`, после чего остается только записать цикл обработки сообщений.
- ❑ В оконной процедуре усложнена обработка сообщения `WM_CLOSE` по сравнению с первой версией программы. При помощи функции `MessageBox` пользователю предлагается подтвердить его желание закрыть приложение. Благодаря флагу `MB_YESNO` диалоговое окно будет содержать две кнопки: `YES` и `NO`. При щелчке мышью на кнопке `YES` функция возвратит значение `IDYES`, и только в этом случае будет вызвана функция `DestroyWindow`.

Откомпилируйте эту программу (проект `Hello2`), чтобы убедиться в ее работоспособности. Поэкспериментируйте с передачей других значений аргументов конструктору объекта `mainWnd`. Например, замените строку

```
KWnd mainWnd("Hello application", hInstance, nCmdShow, WndProc);
```

на строку

```
KWnd mainWnd("Hello application", hInstance, nCmdShow, WndProc, NULL, 50,
100, 200, 150);
```

и посмотрите на поведение измененной программы.

Разработка второго варианта программы «Hello, World!» завершена. Реализованный в ней класс `KWnd` можно использовать для сокращения кода новых приложений.

ВНИМАНИЕ

Если вы будете использовать класс `KWnd`, то подключите к вашему проекту файлы `KWnd.h` и `KWnd.cpp`, а также добавьте директиву `#include "KWnd.h"` в том файле, где он используется.

Функции поддержки окон

Win32 API предоставляет широкий набор функций, которые позволяют менять размеры, расположение и характеристики отображения окна. В табл. 1.12 приведены наиболее употребительные функции.

Таблица 1.12. Функции поддержки окон

Функция	Назначение
BringWindowToTop	Активизирует окно и переносит его в верхнее положение, если оно находится позади других окон
CloseWindow	Сворачивает окно
EnableWindow	Разрешает или запрещает окну принимать ввод данных от мыши или клавиатуры
FindWindow	Находит окно верхнего уровня по заданным имени класса и имени окна
GetActiveWindow	Выбирает дескриптор активного окна
GetClassInfo	Извлекает информацию о классе окна в структуру типа WNDCLASS
GetClassLong	Выбирает специфицированное по индексу значение из структуры типа WNDCLASSEX
GetClientRect	Получает координаты клиентской области окна
GetFocus	Выбирает дескриптор окна, имеющего фокус ввода
GetParent	Выбирает дескриптор родительского окна
GetWindow	Выбирает дескриптор окна
GetWindowInfo	Извлекает информацию об окне в структуру типа WINDOWINFO
GetWindowLong	Выбирает указанное по индексу значение из данных окна
GetWindowRect	Получает координаты левого верхнего и правого нижнего углов окна
IsChild	Определяет, является ли окно дочерним по отношению к данному родительскому окну
IsIconic	Проверяет, свернуто ли окно
IsZoomed	Проверяет, развернуто ли окно
IsWindowEnabled	Проверяет, разрешено или запрещено окно
MoveWindow	Перемещает и изменяет размеры окна
SetActiveWindow	Делает окно активным
SetClassLong	Замещает указанным 32-битным значением прежнее значение указанного поля в структуре WNDCLASSEX для класса, к которому относится указанное окно
SetFocus	Придает окну фокус ввода
SetWindowLong	Изменяет атрибуты указанного окна (первоначально заданные функцией CreateWindow)
SetWindowPos	Изменяет одновременно размеры, позицию и расположение окна относительно других окон
SetWindowText	Изменяет текст заголовка окна
ShowWindow	Отображает, скрывает или изменяет состояние показа окна
UpdateWindow	Обновляет клиентскую область окна при помощи посылки сообщения WM_PAINT непосредственно оконной процедуре, минуя очередь сообщений приложения

Часто используемые сообщения

В табл. 1.13 приведены сообщения, обработка которых в оконной процедуре встречается наиболее часто.

Таблица 1.13. Часто используемые сообщения

Сообщение	Назначение
WM_CLOSE	Уведомляет окно о том, что оно должно быть закрыто. Сообщение может быть использовано для вывода запроса пользователю с предложением подтвердить завершение работы, прежде чем будет вызвана функция DestroyWindow. По умолчанию (если обработка WM_CLOSE не предусмотрена) DefWindowProc вызывает DestroyWindow
WM_COMMAND	Сообщение посыпается, когда пользователь выбирает команду меню или посыпает команду из элемента управления
WM_CREATE	Посыпается, когда приложение создает окно вызовом функции CreateWindow или CreateWindowEx. Оконная процедура получает это сообщение, когда окно уже создано, но еще не показано на экране. Поэтому, обрабатывая это сообщение, можно изменить характеристики окна либо выполнить некоторые инициализирующие действия, например открыть необходимые файлы. После обработки сообщения приложение должно вернуть нулевое значение для продолжения процесса создания окна. Если приложение вернет значение -1, то окно не будет создано, а функция CreateWindow вернет значение NULL
WM_DESTROY	Посыпается оконной процедуре уничтожаемого окна после того, как окно удалено с экрана
WM_INITDIALOG	Сообщение посыпается оконной процедуре диалогового окна ¹ непосредственно перед тем, как оно будет отображено на экране. Обработка этого сообщения позволяет произвести инициализацию тех объектов, которые связаны с диалоговым окном
WM_MOVE	Посыпается окну, которое переместилось на экране
WM_PAINT	Посыпается окну, содержимое которого требует перерисовки
WM_SIZE	Посыпается окну, размеры которого изменились
WM_TIMER	Уведомляет окно о том, что некоторый системный таймер, установленный функцией SetTimer, отсчитал заданный ему интервал

Модификация характеристик окна

Если окно уже создано, например, в виде объекта класса KWnd, то его характеристики могут быть изменены с помощью одной из двух функций: SetClassLong или SetWindowLong.

Рассмотрим применение функции SetClassLong, предназначеннной для изменения одного из полей структуры WNDCLASSEX. Напомним, что указатель на эту структуру с начальными значениями ее полей передавался функции RegisterClassEx для регистрации класса окна.

Функция SetClassLong имеет следующий прототип:

```
DWORD SetClassLong(
    HWND hWnd,           // дескриптор окна
    int nIndex,          // индекс значения, которое нужно изменить
    LONG dwNewLong       // новое значение
);
```

¹ Диалоговые окна рассматриваются в главе 7.

Здесь nIndex — смещение (с отсчетом от нуля) *дополнительных данных*¹ класса, которые должны быть изменены, или одно из значений, приведенных в табл. 1.14.

Таблица 1.14. Значения параметра nIndex функции SetClassLong

Значение	Описание
GCL_HBRBACKGROUND	Изменить дескриптор кисти цвета фона
GCL_HCURSOR	Изменить дескриптор курсора
GCL_HICON	Изменить дескриптор пиктограммы
GCL_HICONSM	Изменить дескриптор малой пиктограммы
GCL_MENUNAME	Изменить адрес имени ресурса меню
GCL_STYLE	Изменить стиль класса окна
GCL_WNDPROC	Изменить адрес оконной процедуры, связанной с классом

Покажем на примере проекта Hello2, как можно модифицировать характеристики главного окна приложения.

Добавьте в текст функции WndProc, а именно в блок оператора switch, следующие строки:

```
case WM_CREATE:
    SetClassLong(hWnd, GCL_HBRBACKGROUND,
        (LONG) CreateSolidBrush(RGB(200,160,255)));
    break;
```

Посмотрите, как изменится вид окна приложения после сделанной модификации².

А теперь добавьте после строки

```
hDC = BeginPaint(hWnd, &ps);
```

следующую строку:

```
SetBkMode(hDC, TRANSPARENT);
```

и посмотрите на полученный результат.

Особенности программирования для Windows

В разделе «Базовые концепции» уже говорилось о том, что в основе взаимодействия приложения и операционной системы лежит механизм сообщений. Для каждого приложения Windows создает *очередь сообщений приложения* и направляет в эту очередь все сообщения, адресованные окнам приложения. В то же время в описании функции UpdateWindow (см. табл. 1.12) отмечается, что в процессе ее выполнения Windows посыпает сообщение WM_PAINT непосредственно оконной процедуре, минуя очередь сообщений приложения. Внимательный читатель наверняка заметил это противоречие. На самом деле противоречия нет, а есть два типа сообщений, с которыми работает Windows.

¹ Дополнительные данные — это те байты, количество которых задано в поле cbClsExtra.

² Вызов функции CreateSolidBrush (RGB (200, 160, 255)) создает сплошную кисть, цвет которой образован тремя цветовыми составляющими заданной интенсивности: красной (200), зеленои (160) и синей (255). В результате должен получиться сиреневый цвет. Более подробно создание кисти с использованием функции CreateSolidBrush рассматривается в главе 2.

Синхронные и асинхронные сообщения

Асинхронными сообщениями называются сообщения, которые Windows помещает в очередь сообщений приложения. Такие сообщения извлекаются и диспетчеризуются в цикле обработки сообщений.

Синхронные сообщения передаются непосредственно окну, когда Windows вызывает оконную процедуру.

Приведем примеры асинхронных сообщений. Прежде всего, к ним относятся сообщения о событиях пользовательского ввода, таких как нажатие клавиш (`WM_KEYDOWN` и `WM_KEYUP`), перемещение мыши (`WM_MOUSEMOVE`) или щелчок левой кнопкой мыши (`WM_LBUTTONDOWN`). Кроме этого, асинхронными являются сообщения от таймера (`WM_TIMER`), сообщение о необходимости перерисовки клиентской области (`WM_PAINT`) и сообщение о выходе из программы (`WM_QUIT`). Приложение может само направить в очередь асинхронное сообщение, вызвав функцию `PostMessage`.

Остальные сообщения, как правило, являются синхронными. Во многих случаях синхронные сообщения являются результатом обработки асинхронных сообщений. Вообще, когда синхронное сообщение обрабатывается функцией `DefWindowProc`, Windows часто генерирует другие сообщения, направляемые оконной процедуре. Приложение также может послать синхронное сообщение, вызвав функцию `SendMessage`.

Таким образом, оконная процедура должна быть *повторно входимой* (*reentrant program*). Это означает, что Windows часто вызывает функцию `WndProc` с новым сообщением, появившимся в результате вызова `DefWindowProc` из `WndProc` при обработке предыдущего сообщения. В большинстве случаев повторная входимость оконной процедуры не создает каких-то особых проблем, но знать об этом полезно.

Рассмотрим, например, какие события произойдут после щелчка кнопкой мыши на кнопке закрытия окна приложения `Hello1`. Все начнется с того, что Windows отправит синхронное сообщение `WM_SYSCOMMAND` оконной процедуре `WndProc`. Оконная процедура передаст это сообщение на обработку функции `DefWindowProc`. Функция `DefWindowProc` реагирует на него, отправляя сообщение `WM_CLOSE` оконной процедуре. В рассматриваемом примере предусмотрена обработка этого сообщения — вызывается функция `DestroyWindow`. Однако если не предусмотреть эту обработку, то функция `DefWindowProc` сделала бы то же самое, то есть вызвала бы функцию `DestroyWindow`. Функция `DestroyWindow` заставляет Windows отправить оконной процедуре сообщение `WM_DESTROY`. И наконец, `WndProc`, обрабатывая это сообщение, вызывает функцию `PostQuitMessage`, которая посылает асинхронное сообщение `WM_QUIT` в очередь сообщений приложения. Сообщение `WM_QUIT` прерывает цикл обработки сообщений в `WinMain`, и приложение завершает свою работу.

Сообщения не похожи на аппаратные прерывания. Пока оконная процедура обрабатывает одно сообщение, программа не может быть прервана другим сообщением. Только в том случае, когда функция, выполняемая в теле оконной процедуры, генерирует новое синхронное сообщение, оно вызывает повторно оконную процедуру, и только после его обработки выполнение прерванной функции продолжается.

Посылка сообщений из приложения

Основным средством программного взаимодействия между разными окнами приложения и даже между разными приложениями является посылка сообщений. Приведем краткие сведения об основных функциях, используемых для отправки сообщений.

Функция SendMessage

Эта функция посыпает синхронное сообщение указанному окну или нескольким окнам. Она имеет следующий прототип:

```
HRESULT SendMessage(  
    HWND hWnd,           // дескриптор окна-получателя  
    UINT Msg,            // код сообщения  
    WPARAM wParam,       // первый параметр сообщения  
    LPARAM lParam);    // второй параметр сообщения
```

);

Параметры функции те же, что и параметры, передаваемые в оконную процедуру.

Когда приложение вызывает `SendMessage`, Windows, в свою очередь, вызывает оконную процедуру с дескриптором окна `hWnd`, передавая ей эти четыре параметра. После того как оконная процедура заканчивает обработку сообщения, система Windows передает управление инструкции, следующей за вызовом `SendMessage`. Оконная процедура, которой отправляется сообщение, может быть той же самой оконной процедурой, другой оконной процедурой той же программы или даже оконной процедурой другого приложения.

Если первый параметр функции имеет значение `HWND_BROADCAST`, то сообщение посыпается всем окнам верхнего уровня, существующим в настоящий момент в системе.

Параметры `wParam` и `lParam` содержат дополнительную информацию, интерпретация которой зависит от кода сообщения. Чтобы получить справочную информацию в MSDN об интерпретации этих параметров, применяйте поиск по *коду сообщения*, то есть по идентификаторам `WM_PAINT`, `WM_TIMER`, `WM_SETFONT` и им подобным.

Функция SendNotifyMessage

Функция `SendNotifyMessage` имеет те же параметры, что и функция `SendMessage`.

Если окно, которому адресовано сообщение, создано вызывающим потоком, то поведение функции `SendNotifyMessage` не отличается от поведения функции `SendMessage`. Она отправляет сообщение оконной процедуре и не возвращает управление, пока оконная процедура не обработает это сообщение.

Если же окно, которому адресовано сообщение, создано другим потоком, то функция `SendNotifyMessage` передает сообщение указанной оконной процедуре и немедленно возвращает управление, не дожидаясь окончания обработки сообщения оконной процедурой.

Функция PostMessage

Функция `PostMessage` посыпает асинхронное сообщение указанному окну. Она имеет те же параметры, что и функция `SendMessage`.

В отличие от `SendMessage`, функция `PostMessage` не вызывает оконную процедуру, а отправляет сообщение в очередь сообщений потока, создавшего окно `hWnd`.

После этого функция возвращает управление, не дожидаясь, пока поток обрабатывает отправленное сообщение.

Использование глобальных или статических переменных

Иногда бывает нужно, чтобы информация, полученная в процессе обработки какого-то сообщения, была сохранена и использована позже при обработке другого сообщения.

Рассмотрим следующий пример. В программе Hello1 размеры клиентской области окна были получены при помощи функции `GetClientRect`. Но существует и другой способ получить эту информацию — обрабатывая сообщение `WM_SIZE`. Параметр `lParam` этого сообщения содержит в своем младшем слове значение ширины, а в старшем слове — значение высоты клиентской области окна. Допустим, что мы определили две локальные переменные в теле функции `WndProc`:

```
int width, height;
```

и добавили обработку сообщения `WM_SIZE`, обеспечивающую сохранение соответствующих значений:

```
case WM_SIZE:
    width = LOWORD(lParam);
    height = HIWORD(lParam);
    return 0;
```

Здесь `LOWORD` и `HIWORD` — макросы, определенные в заголовочных файлах Windows; первый из них извлекает младшее слово, а второй — старшее слово из 32-разрядного аргумента.

Закомментируем вызов функции `GetClientRect` в блоке обработки сообщения `WM_PAINT` и вместо него вставим следующий код:

```
rect.left = 0;
rect.top = 0;
rect.right = width;
rect.bottom = height;
```

После компиляции и запуска программы будет показано пустое окно, не содержащее никакого текста.

Неудача объясняется тем, что при повторном вызове `WndProc` значения ее локальных переменных от предыдущего вызова не сохраняются.

Но эту ошибку легко исправить. Надо лишь объявить переменные `width` и `height` либо как статические (`static`), либо как глобальные.

ВНИМАНИЕ

Если в оконной процедуре присваивание значения некоторой локальной переменной и дальнейшее использование этого значения разделены по разным ветвям оператора `switch`, то не забудьте объявить эту переменную с классом памяти `static!` Другое возможное решение — использование глобальной переменной.

Получение дескриптора экземпляра приложения

Для многих функций Win32 API необходимо передавать дескриптор экземпляра приложения в качестве одного из параметров. Напомним, что значение этого

дескриптора `hInstance` функция `WinMain` получает от операционной системы через свой первый параметр.

Если значение дескриптора `hInstance` используется в теле функции `WinMain`, проблем никаких нет, если в теле другой функции — возникает вопрос, как получить значение `hInstance`? Есть три способа решения проблемы.

Первый способ (наихудший, с точки зрения стиля программирования на C++) — объявить глобальную переменную

```
HINSTANCE hInst;
```

и в теле функции `WinMain` запомнить значение дескриптора `hInstance`. Естественно, значение глобальной переменной будет доступно во всех других функциях.

Во втором и третьем способах переменная `hInst` объявляется как локальная. Во втором способе ее значение определяют при помощи функции `GetClassLong`:

```
hInst = (HINSTANCE)GetClassLong(hWnd, GCL_HMODULE);
```

В третьем способе для этого используется функция `GetModuleHandle`:

```
hInst = GetModuleHandle(NULL);
```

Предотвращение зависания приложения в случае медленной обработки отдельных событий

Предположим, что в вашей программе обработка какого-то события является очень ресурсоемкой, занимая процессор в течение нескольких минут. Несмотря на то что Windows является многозадачной операционной системой, выделяющей принудительно кванты времени для выполнения других приложений, пользователь будет не в состоянии что-нибудь сделать с этой злополучной программой, пока не завершится указанная обработка. Он не сможет даже перетащить окно в другое место экрана или закрыть приложение. Внешне это выглядит как зависание приложения. Вряд ли пользователю понравится такое поведение программы.

Рассмотрим эту проблему на конкретном примере. Предположим, в приложении имеется обработка некоторой команды меню (например, `Play`) с помощью следующей функции:

```
void OnPlay() {
    while (!fReadData.eof()) {
        fReadData.read(buf, 512);
        DoSomething(); // какая-нибудь обработка
    }
}
```

Здесь `fReadData` — это объект класса `ifstream`. В цикле `while` осуществляются чтение файла `fReadData` блоками по 512 байт и последующая обработка каждого блока с помощью функции `DoSomething`. Допустим, что размер файла достигает 5 Мбайт, а время выполнения функции `DoSomething` составляет 10 мс. Тогда тело цикла будет выполняться $5335040 / 512 = 10420$ раз, и возврат из функции `OnPlay` произойдет примерно через 105 с. В течение этого промежутка времени окно программы будет безвольно висеть на экране.

Для решения проблемы нам нужен способ узнать, имеется ли какое-нибудь сообщение в очереди сообщений приложения или же она пуста. Если сообщение есть, нужно дать возможность Windows обработать его, если нет — можно продолжить выполнение цикла. Win32 API предлагает использовать для этого

функцию PeekMessage. Ее прототип практически идентичен прототипу функции GetMessage:

```
BOOL PeekMessage(  
    LPMMSG lpMsg,           // адрес структуры с сообщением  
    HWND hWnd,              // дескриптор окна  
    UINT wMsgFilterMin,     // первое сообщение  
    UINT wMsgFilterMax,     // последнее сообщение  
    UINT wRemoveMsg         // флаг удаления сообщения  
>);
```

Данная функция возвращает ненулевое значение, если в очереди имеется сообщение. Различие прототипов функций заключается в последнем параметре, который определяет, как именно должно быть выбрано сообщение из очереди. Возможными значениями параметра wRemoveMsg являются:

- PM_NOREMOVE — сообщение остается в очереди;
- PM_REMOVE — сообщение удаляется из очереди.

Покажем, как нужно переделать функцию OnPlay, используя вызов PeekMessage, чтобы приложение не зависало:

```
void OnPlay(HWND hWnd) {  
    MSG message;  
    while (!fReadData.eof()) {  
        fReadData.read(buf, 512);  
  
        // Предотвращение зависания приложения  
        if (PeekMessage(&message, hWnd, 0, 0, PM_REMOVE)) {  
            TranslateMessage(&message);  
            DispatchMessage(&message);  
        }  
  
        DoSomething(); // какая-нибудь обработка  
    }  
}
```

Если функция PeekMessage обнаруживает сообщение в очереди, то оно обрабатывается так же, как и в основном цикле обработки сообщений.

Данный инструментальный прием будет использован в приложении ProgressBar, рассматриваемом в главе 8.

Использование утилиты Spy++

В процессе отладки приложений иногда бывает полезным увидеть, какие сообщения вырабатывает Windows в ответ на те или иные действия пользователя. В составе интегрированной среды Visual Studio 6 есть удобное инструментальное средство для решения данной проблемы — утилита Spy++.

Утилиту можно вызвать при помощи команды меню интегрированной среды: Tools ▶ Spy++.

Работа с утилитой Spy++ описана в приложении 3.

2

GDI — графический интерфейс устройства. Рисование линий, фигур, текста

Графический интерфейс устройства (*graphics device interface (GDI)*) — это составная часть Win32 API, обеспечивающая графический вывод на устройства отображения информации, такие как дисплей или принтер. Windows-приложение не имеет непосредственного доступа к аппаратным устройствам. Вместо этого оно обращается к функциям GDI, а GDI транслирует эти обращения к программным драйверам физических устройств, обеспечивая аппаратную независимость приложений (рис. 2.1). Код библиотеки GDI находится в файле gdi32.dll, то есть библиотека является динамически загружаемой. Драйверы стандартных устройств поставляются как часть Windows, а драйверы специализированных устройств создаются производителями оборудования.

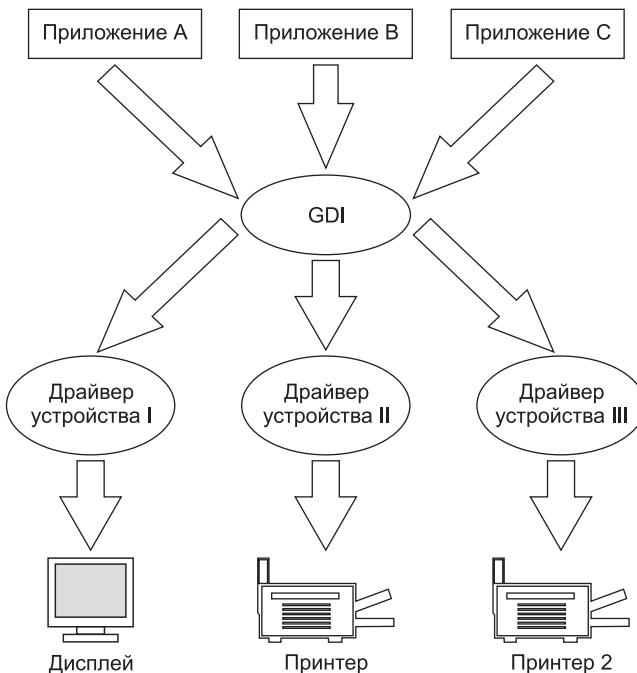


Рис. 2.1. Соотношения между приложениями, GDI и драйверами устройств

Контекст устройства

Взаимодействие приложения с GDI осуществляется при непременном участии еще одного посредника — так называемого контекста устройства.

Контекст устройства (device context) — это внутренняя структура данных, которая определяет набор графических объектов и ассоциированных с ними атрибутов, а также графических режимов, влияющих на вывод.

В следующем списке приведены основные графические объекты:

- ❑ *Перо (pen)* для рисования линий.
- ❑ *Кисть (brush)* для заполнения фона или заливки фигур.
- ❑ *Растровое изображение (bitmap)* для отображения в указанной области окна.
- ❑ *Палитра (palette)* для определения набора доступных цветов.
- ❑ *Шрифт (font)* для вывода текста.
- ❑ *Регион (region)* для отсечения области вывода.

Если необходимо рисовать на устройстве графического вывода (экране дисплея или принтере), то сначала нужно получить *дескриптор контекста устройства*. Возвращая этот дескриптор после вызова соответствующих функций, Windows тем самым предоставляет разработчику право на использование данного устройства. После этого дескриптор контекста устройства передается как параметр в функции GDI, чтобы идентифицировать устройство, на котором должно выполняться рисование.

Контекст устройства содержит много атрибутов, определяющих поведение функций GDI. Благодаря этому списки параметров функций GDI содержат только саму необходимую информацию, например начальные координаты или размеры графического объекта. Все остальное система извлекает из контекста устройства.

Когда в приложении Hello1 вызывалась функция `DrawText`, то в ее параметрах нужно было указать только дескриптор контекста устройства, адрес строки с выводимым текстом, его длину, положение и размеры прямоугольной области для размещения текста, а также способ позиционирования текста внутри этой области. Однако при этом не указывались шрифт, цвет текста, цвет фона, режим смешивания фона, режим рисования и другие атрибуты, потому что все они являются частью контекста устройства и имеют значения по умолчанию. И только если значения по умолчанию не устраивают разработчика, необходимо вызывать функции GDI, изменяющие значения соответствующих атрибутов.

Типы контекстов устройства

Win32 API поддерживает следующие типы контекстов устройства:

- ❑ контекст дисплея;
- ❑ контекст принтера;
- ❑ контекст в памяти (совместимый контекст);
- ❑ метафайловый контекст;
- ❑ информационный контекст.

Конечно, чаще всего используется контекст дисплея. Первое знакомство с ним состоялось, когда рассматривалась обработка сообщения WM_PAINT в программе Hello1 (глава 1). Сообщение WM_PAINT уведомляет программу, что часть или вся клиентская область окна *недействительна* и ее следует перерисовать. Понятие недействительного, или обновляемого, региона очень важно для взаимодействия Windows и приложения, поэтому, прежде чем продолжить более подробное рассмотрение контекста дисплея, мы остановимся кратко на следующей теме.

Регионы Windows. Отсечение

Для повышения эффективности работы Windows оперирует с несколькими типами регионов. Идея заключается в том, чтобы рисовать именно в той части окна, которая требует обновления, а не перерисовывать все окно. Также регионы позволяют отсекать вывод той части графической информации, которая не может быть отображена в данный момент. Вообще полное изучение всей иерархии регионов и их взаимодействия является непростой задачей, требующей пространного изложения¹. В то же время приведенное ниже упрощенное описание достаточно для понимания работы большинства функций Win32 GDI.

Обновляемый регион (update region), или, как его тоже иногда называют, *недействительный регион (invalid region)* — это часть окна, которая требует обновления после возникновения тех или иных событий.

Видимый регион (visible region) — та часть окна, которую в данный момент видит пользователь. Система изменяет видимый регион окна и в том случае, когда окно изменяет размеры, и в том случае, когда перемещение другого окна либо закрывает часть данного окна, либо открывает закрытую прежде часть.

Регион отсечения (clipping region) ограничивает область, внутри которой система разрешает отображение графической информации. Когда приложение получает контекст устройства при помощи функции BeginPaint, система устанавливает регион отсечения путем пересечения видимого региона и обновляемого региона. Приложение может ужесточить регион отсечения и ввести дополнительные ограничения при помощи вызова функций SetWindowRgn, SelectClipPath или SelectClipRgn.

Если при создании окна функцией CreateWindow был использован стиль WS_CLIPCHILDREN или WS_CLIPSIBLINGS, то это вносит дополнительные правила в определение видимого региона, исключая из него любое дочернее или любые «сестринские» окна. Благодаря этому рисование не затрагивает отображаемые области таких окон.

Контекст дисплея

Контекст дисплея создавать не нужно — об этом уже позаботилась операционная система. Нужно лишь получить его дескриптор. Windows предоставляет для этого два метода, применение которых обусловлено (извините за каламбур) *программным контекстом*.

¹ Интересующихся мы можем отослать к известной книге Фень Юаня «Программирование графики для Windows» [6].

Первый метод получения дескриптора контекста устройства

Этот метод используется при обработке сообщения WM_PAINT. Контекст устройства получают вызовом функции BeginPaint, имеющей следующий прототип:

```
HDC BeginPaint(  
    HWND hwnd,           // дескриптор окна  
    LPPAINTSTRUCT lPaint // указатель на структуру типа PAINTSTRUCT  
);
```

В случае успешного завершения функция возвращает дескриптор контекста дисплея для клиентской области окна.

Кроме этого функция заполняет поля структуры PAINTSTRUCT, имеющей следующее определение:

```
typedef struct tagPAINTSTRUCT {  
    HDC hdc;           // контекст устройства  
    BOOL fErase;        // признак стирания фона клиентской области  
    RECT rcPaint;      // границы недействительного прямоугольника  
    BOOL fRestore;  
    BOOL fIncUpdate;  
    BYTE rgbReserved[32];  
} PAINTSTRUCT;
```

Последние три поля используются операционной системой.

Поле rcPaint типа RECT содержит координаты обновляемого прямоугольника в пикселях относительно левого верхнего угла клиентской области окна. Эти координаты либо определяются системой, либо задаются при вызове функции InvalidateRect. Регион отсечения в этом случае определяется посредством пересечения видимого региона и обновляемого прямоугольника.

Поле fErase определяет, будет ли Windows обновлять фон недействительного региона. Чаще всего fErase имеет значение TRUE, что означает стирание (обновление) фона. Когда обновляемый регион формируется вызовом функции InvalidateRect или InvalidateRgn, один из параметров этих функций разрешает или подавляет стирание фона.

Если задано стирание фона, то функция BeginPaint посылает оконной процедуре сообщение WM_ERASEBKGND. Приложение может обработать это сообщение, чтобы отобразить однородный или растровый фон. Однако обычно оно обрабатывается по умолчанию функцией DefWindowProc, которая обновляет фон с использованием кисти, определенной в поле hbrBackground класса окна.

Следует отметить, что функция BeginPaint сбрасывает обновляемый регион в NULL, или, другими словами, превращает недействительный регион в действительный. Это предотвращает повторную генерацию системой сообщения WM_PAINT до тех пор, пока обновляемый регион вновь не изменится.

До своего завершения функция BeginPaint посылает оконной процедуре еще одно сообщение — WM_NCPAINT. Оно заставляет приложение перерисовать так называемую *неклиентскую область*, которая представляет собой остальную часть окна — кроме клиентской области. Как правило, это сообщение обрабатывается также функцией DefWindowProc.

Типовой процесс обработки сообщения WM_PAINT выглядит следующим образом:

```
case WM_PAINT:  
    hdc = BeginPaint(hwnd, &ps);  
    [ использование функций GDI ]  
    EndPaint(hwnd, &ps);  
    return 0;
```

После завершения операций рисования приложение должно вызывать функцию `EndPaint`, чтобы освободить контекст устройства.

Примеры использования контекста дисплея в блоке обработки сообщения `WM_PAINT` были приведены в листингах из первой главы.

Второй метод получения дескриптора контекста устройства

Иногда рисование должно происходить не в блоке обработки сообщения `WM_PAINT`, а при обработке других сообщений. Или, быть может, дескриптор контекста устройства нужен не для рисующих, а для иных функций (например, в качестве аргумента функции `GetTextMetrics`), вызываемых вне блока обработки сообщения `WM_PAINT`. В таких ситуациях контекст устройства получают либо вызовом функции `GetDC`:

```
hdc = GetDC(hWnd);
```

либо вызовом функции `GetWindowDC`:

```
hdc = GetWindowDC(hWnd);
```

Первая функция возвращает дескриптор контекста дисплея для клиентской области окна `hWnd`, а вторая — для всего окна. Если в качестве аргумента `hWnd` передать значение `NULL`, то обе функции вернут дескриптор устройства для всего экрана.

По окончании работы с функциями GDI необходимо освободить контекст устройства с помощью функции `ReleaseDC`, например:

```
hDC = GetDC(hWnd);
[ использование функций GDI ]
ReleaseDC(hWnd, hDC);
```

Заметим, что, в отличие от контекста устройства, полученного при помощи вызова `BeginPaint`, контекст устройства, возвращаемый функцией `GetDC`, работает с регионом отсечения, который равен всей клиентской области. Это значит, что рисование можно производить в любом месте клиентской области, а не только в недействительном прямоугольнике, если он вообще определен. В отличие от функции `BeginPaint`, функция `GetDC` не делает действительными какие-либо недействительные зоны.

Обычно вторым методом получения дескриптора контекста дисплея пользуются программы текстовых редакторов, обрабатывая сообщения от клавиатуры, а также программы рисования, обрабатывая сообщения от мыши. Это позволяет обновлять клиентскую область непосредственно в ответ на ввод информации с клавиатуры или от мыши и при этом не объявлять часть окна недействительной для генерации сообщения `WM_PAINT`. Тем не менее, подобные программы должны уметь обновлять окно в любой момент, когда они получают сообщение `WM_PAINT`.

Но на этом наше знакомство с контекстом дисплея еще не закончилось. Win32 API предлагает к использованию *два подтипа* контекста дисплея, различающиеся механизмами выделения и освобождения памяти¹.

¹ На самом деле есть еще и третий подтип — *классовый* контекст устройства, но он реализован лишь для совместимости с 16-разрядными версиями Windows и не рекомендован к применению в 32-разрядных приложениях.

Подтипы контекста дисплея

Любая из функций, `BeginPaint`, `GetDC`, `GetWindowDC`, возвращает либо *общий* (*common*), либо *частный* (*private*) контекст устройства в зависимости от стиля, указанного в классе данного окна (поле `style` в структуре типа `WNDCLASS`).

Общий контекст дисплея

Система предоставляет *общий контекст* дисплея по умолчанию, если в классе окна не указан явно стиль контекста устройства. Обычно общий контекст используется, если рисование в приложении не требует интенсивных изменений атрибутов контекста, которые могли бы занимать много процессорного времени.

Система Windows получает все общие контексты устройств из кэша экранных контекстов. Поэтому приложение должно освобождать общий контекст устройства сразу же после окончания рисования. При освобождении контекста теряются все сделанные изменения атрибутов.

При повторном вызове общего контекста дисплея все значения атрибутов, отличающиеся от значений по умолчанию, должны быть установлены заново.

Частный контекст дисплея

Частный контекст задается посредством указания флага `CS_OWNDC` для поля `style` в структуре `WNDCLASS` перед регистрацией класса окна. В противоположность общему контексту дисплея в частном контексте все изменения атрибутов сохраняются даже после освобождения контекста.

Частный контекст устройства не является частью системного кэша, и память для него выделяется системой специально для данного приложения. Система автоматически освобождает частный контекст устройства после того, как уничтожается последнее окно данного класса.

Частный контекст используется в приложениях с высокой интенсивностью применения операций рисования, например в программах компьютерного проектирования (CAD), издательских системах, графических редакторах и им подобных приложениях.

Использование сообщения WM_PAINT

Обычно приложение рисует что-либо в окне, реагируя на сообщение `WM_PAINT`. Система посыпает это сообщение окну во всех случаях, требующих перерисовки клиентской области окна. Например, типичными причинами генерации этого сообщения могут быть следующие события:

- ❑ изменились размеры или местоположение окна;
- ❑ клиентская область была частично или полностью закрыта другим окном или выпадающим меню, а теперь закрывающий объект исчез;
- ❑ приложение вызвало одну из функций работы с полосами прокрутки.

Кроме того, приложение может само инициировать посылку сообщения `WM_PAINT` посредством вызова одной из функций, `InvalidateRect`, `InvalidateRgn` или `UpdateWindow`.

Функция `UpdateWindow` посыпает сообщение `WM_PAINT` непосредственно в оконную процедуру, минуя очередь приложения.

Работа с сообщением `WM_PAINT` требует от разработчика понимания общей парадигмы отображения графики на экране, которая принята в системах семейства

Windows. Желательно структурировать программу таким образом, чтобы информация, необходимая для рисования в клиентской области, готовилась там, где это удобно с точки зрения реализуемого алгоритма. Но само рисование должно выполняться только тогда, когда появляется сообщение WM_PAINT. Впрочем, иногда решаемая задача диктует и другие подходы к рисованию: например, оно может выполняться при обработке сообщений от мыши или от таймера.

Использование функций InvalidateRect и InvalidateRgn

Если в каком-то месте алгоритма необходимо сгенерировать сообщение WM_PAINT для перерисовки некоторой части окна, то приложение может воспользоваться функциями InvalidateRect и InvalidateRgn.

Функция InvalidateRect имеет следующий прототип:

```
BOOL InvalidateRect(HWND hWnd, CONST RECT* lprRect, BOOL bErase);
```

Параметры этой функции перечислены в следующем списке:

- ❑ hWnd — дескриптор окна, у которого изменился обновляемый регион. Если этот параметр равен NULL, то система обновляет и перерисовывает все окна приложения, а также посыпает сообщения WM_ERASEBKGND и WM_NCPAINT оконной процедуре до возврата из функции.
- ❑ lprRect — указатель на структуру типа RECT, содержащую клиентские координаты прямоугольника, который добавляется к обновляемому региону. Если этот параметр имеет значение NULL, то к обновляемому региону добавляется вся клиентская область.
- ❑ bErase — флаг, определяющий, будет ли стираться фон обновляемого региона. Если этот параметр равен TRUE, то фон стирается, когда вызывается функция BeginPaint. Если указано значение FALSE, то фон остается без изменения.

Последний параметр очень важен. Нужно быть очень внимательным, определяя его значение. Ни в коем случае нельзя передавать значение TRUE, если можно обойтись значением FALSE, так как это может оказаться причиной очень неприятного мерцания фона окна. Если же стирание фона все-таки необходимо, следует постараться минимизировать обновляемый регион. Иными словами, если можно ограничить стираемую область некоторым прямоугольником rcUpdate, то в качестве второго параметра следует передавать ссылку на этот прямоугольник, а не значение NULL.

Функция InvalidateRgn имеет следующий прототип:

```
BOOL InvalidateRgn(HWND hWnd, HRGN hRgn, BOOL bErase);
```

В этой функции все параметры, кроме второго, имеют тот же смысл, что и в предыдущей функции. Параметр hRgn содержит дескриптор региона, добавляемого к обновляемому региону. Этот дескриптор можно получить при помощи функций типа CreateRectRgn, CreatePolygonRgn и других функций, рассматриваемых в разделе «Регионы».

Таким образом, и функция InvalidateRect, и функция InvalidateRgn изменяют обновляемый регион так, что в любом случае он уже не является пустым. Если же обновляемый регион не пуст, то система отправляет окну hWnd сообщение WM_PAINT.

Типичная ошибка

Если не нужно что-либо выводить в клиентскую область окна, то можно доверить обработку сообщения WM_PAINT функции DefWindowProc. Но не следует писать такой код:

```
case WM_PAINT:  
    return 0; // ОШИБКА !!!
```

Дело в том, что, как только появится непустой обновляемый регион, Windows поставит сообщение **WM_PAINT** в очередь. А оконная процедура в приведенном примере *не вызывает* функцию **BeginPaint**, которая делает обновляемую область *действительной*. Поэтому Windows снова отправит сообщение **WM_PAINT**. И система будет отсылать это сообщение постоянно, создавая бесконечный цикл.

Контекст принтера

При необходимости вывода информации на принтер приложение должно создать контекст устройства, вызвав функцию **CreateDC**. В качестве аргументов функции передаются имя драйвера устройства, имя устройства и, при необходимости, данные для инициализации устройства. Используя полученный дескриптор контекста, можно подготовить устройство для печати и вывести необходимую информацию. Завершив работу с принтером, приложение должно удалить контекст принтера с помощью функции **DeleteDC**.

Контекст в памяти (совместимый контекст)

Рассмотренные выше типы контекстов устройства позволяют выводить графику только на физические устройства, такие как видеoadаптеры или принтеры. Работа этих устройств обеспечивается драйверами, взаимодействующими с ними на низком уровне. Но в некоторых ситуациях бывает удобно осуществлять вывод на виртуальное графическое устройство, имитируемое в памяти в виде раstra. *Контекст устройства в памяти* (или *совместимый контекст*) позволяет системе работать с указанным виртуальным устройством.

Такой контекст создается функцией, имеющей следующий прототип:

```
HDC CreateCompatibleDC (HDC hdc);
```

Параметр **hdc** позволяет указывать дескриптор существующего контекста устройства, с которым должен быть совместим создаваемый контекст. Если этот параметр равен **NULL**, то функция создает контекст в памяти, совместимый с текущим экраном приложения.

Если функция завершается успешно, то она возвращает дескриптор совместимого контекста устройства. Совместимый контекст устройства можно создавать только для тех устройств, которые поддерживают растровые операции.

Когда совместимый контекст устройства создан, с ним по умолчанию связано растровое изображение из одного пикселя. Поэтому прежде чем использовать этот контекст, необходимо связать с ним растр (*bitmap*) с необходимыми шириной и высотой. Это осуществляется при помощи функции **SelectObject** или **CreateCompatibleBitmap**.

После создания совместимого контекста устройства он используется как обычный контекст: вы можете устанавливать его атрибуты, выбирать перья, кисти и иные необходимые параметры.

Когда совместимый контекст устройства перестанет быть нужным, необходимо его удалить, вызвав функцию **DeleteDC**.

Пример использования совместимого контекста приведен в главе 3.

Метафайловый контекст

Другой разновидностью контекста, не соответствующего реальному физическому устройству, является метафайловый контекст устройства.

Совместимый контекст позволяет сформировать растр с использованием графических команд GDI. В противоположность этому *метафайловый контекст устройства* позволяет сохранить команды GDI в виде списка графических команд. Этот список обычно сохраняется в файле на диске, который затем может быть прочитан и воспроизведен для восстановления сформированного изображения.

Существенное различие между этими типами контекстов состоит в том, что совместимый контекст работает с растром, имеющим фиксированные размеры и разрешение, в то время как метафайловый контекст сохраняет векторные и растровые команды, которые при последующем использовании точно масштабируются по заданным размерам.

Вопросы использования метафайлового контекста будут рассматриваться в главе 3.

Информационный контекст

Иногда потребности приложения ограничиваются простым получением атрибутов графического устройства. В таких ситуациях можно использовать усеченный контекст устройства, называемый *информационным контекстом*, причем под графическим устройством здесь понимается не только экран, но и любое устройство, поддерживаемое GDI.

Информационный контекст создается функцией `CreateIC`, прототип которой приведен ниже:

```
HDC CreateIC(
    LPCTSTR lpszDriver,           // имя драйвера
    LPCTSTR lpszDevice,          // имя устройства
    LPCTSTR lpszOutput,          // имя порта или файла
    CONST DEVMODE* lpDvm        // указатель на структуру DEVMODE
);
```

В Win32-приложениях третий параметр всегда игнорируется и при вызове функции должен быть равным `NULL`.

Четвертый параметр задает адрес структуры типа `DEVMODE`, в которую записывается извлекаемая информация.

Когда информационный контекст устройства перестанет быть нужным, не забудьте его удалить, вызвав функцию `DeleteDC`.

Системы координат и преобразования

Экранные, оконные и клиентские координаты

Система координат для окна базируется на координатной системе дисплея. Основной единицей измерения служит пиксел. Точки на экране задаются парой координат (x, y) . При этом x -координаты возрастают слева направо, а y -координаты — сверху вниз. Расположение начала координат зависит от режима отображения.

Позиция окна на экране задается в приложениях в так называемых *экранных координатах* (*screen coordinates*), для которых началом координат является левый верхний угол экрана. Часто эта позиция описывается структурой типа `RECT`, содержащей экранные координаты левого верхнего угла и правого нижнего угла окна.

Позиция точки в окне задается либо в *оконных координатах* (*window coordinates*), либо в *клиентских координатах* (*client coordinates*) (см. рис. 1.3). Выбор системы координат зависит от используемых функций Win32 GDI, поэтому следует обращать внимание на спецификацию системы координат, читая описание функции в MSDN. Оконные и клиентские координаты гарантируют надлежащее позиционирование точки в окне независимо от положения окна на экране.

Типы координатных систем

Приложения используют координатные системы и геометрические преобразования для масштабирования, вращения, перемещения, сдвига и зеркального отражения графических объектов. Каждая из используемых систем является некоторой разновидностью декартовой прямоугольной системы координат. По практическим соображениям в Win32 GDI реализована поддержка четырех координатных систем, функционирующих подобно некоторому конвейеру, по которому движется геометрическая модель, описывающая размеры и местонахождение объектов.

- *Мировая система координат* (*world space*) может использоваться как начальная система координат, обеспечивающая любое преобразование.
- *Страницчная система координат* (*page space*) используется или как следующая система координат после мировой системы координат, или как стартовая система, которая поддерживает ограниченные преобразования. В этой системе могут устанавливаться режимы отображения.
- *Система координат устройства* (*device space*) используется после страницной системы. В ней осуществляется только перемещение начала координат, чтобы обеспечить надлежащее положение изображения в физической системе координат.
- *Физическая система координат* (*physical device space*) — последняя система в конвейере геометрических преобразований. Она используется драйвером графического устройства.

Начальное размещение изображения на уровне приложения происходит в мировой системе координат, если приложение вызвало функцию `SetWorldTransform`. В ином случае начальное размещение изображения производится в страницной системе координат. На рис. 2.2 показаны типичные преобразования систем координат, формируемые системой после вызова функции `SetWorldTransform`.

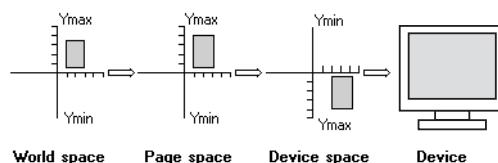


Рис. 2.2. Типичные преобразования систем координат для вывода изображения

Физическая система координат

Физическая система координат представляет собой матрицу пикселов фиксированной ширины и высоты. Начало координат находится в левом верхнем углу. Ось *X* направлена слева направо, а ось *Y* — сверху вниз. Максимальный размер физического устройства равен $2^{27} \times 2^{27}$ пикселов.

Для дисплея физические координаты называют также *экранными координатами*¹. Экранные координаты обычно используются в операциях управления окнами. Например, функция `GetWindowRect` возвращает ограничивающий прямоугольник окна в экранных координатах. Также экранные координаты содержатся в параметрах таких сообщений, как `WM_NCMOUSEMOVE`. Это сообщение генерируется при перемещении курсора мыши в пределах неклиентской части окна.

Следует учитывать, что бывают случаи, когда не все пиксели физической поверхности могут отображаться устройством. Например, для принтеров существуют механические ограничения, не позволяющие печатать у краев страниц. В таких ситуациях приложение должно получить информацию об отображаемой части поверхности при помощи функции `GetDeviceCap`.

Система координат устройства

Координаты устройства (*device coordinates*) используются при работе с контекстами устройств. В общем случае система координат устройства является подмножеством соответствующей физической системы координат.

Для контекстов устройств, созданных функциями `CreateDC`, `CreateIC` и `CreateCompatibleDC`, система координат устройства идентична физической системе координат.

Для контекстов устройств, связанных с конкретными окнами (то есть возвращаемых функциями `GetDC`, `GetWindowDC` и `BeginPaint`), система координат устройства определяется прямоугольником окна или его клиентской области.

Как и в физической системе координат, левый верхний угол системы координат устройства имеет координаты $(0, 0)$, ось *X* направлена слева направо, а ось *Y* — сверху вниз. Зная дескриптор контекста устройства, можно узнать относительную позицию системы координат устройства в его физических координатах при помощи функции `GetDCOrgEx`.

В интерактивных графических приложениях, использующих сообщения мыши для работы с графическими объектами, возникает необходимость преобразования между физическими экранными координатами и координатами устройства (клиентскими координатами). Win32 API предлагает соответствующие функции для решения этой задачи:

```
BOOL ClientToScreen(HWND hWnd, LPOINT lpPoint);
BOOL ScreenToClient(HWND hWnd, LPOINT lpPoint);
```

В области управления окнами координаты устройства чаще всего интерпретируются как *клиентские координаты*. Клиентские координаты содержатся также в сообщениях, генерируемых мышью, таких как `WM_MOUSEMOVE` и `WM_LBUTTONDOWN`.

¹ См. выше раздел «Экранные, оконные и клиентские координаты».

Страницная система координат и режимы отображения

Обе рассмотренные системы координат ограничиваются представлением в виде аппаратно-зависимого массива пикселов. Размер окна на экране с высоким разрешением отличается от размера окна при низком разрешении, а толщина напечатанной линии из трех пикселов будет зависеть от разрешения принтера. Чтобы графическое программирование в меньшей степени зависело от устройства, Win32 GDI позволяет приложениям создавать собственные *логические* системы координат, приближенные к их геометрическим моделям. В таких системах координат удобнее работать, и они гораздо меньше зависят от оборудования.

Одной из двух логических систем координат в Win32 API является *страницная система координат*. Это единственная логическая система координат, которая поддерживается 16-разрядными ОС семейства Windows и даже реализацией Win32 в Windows 95/98. Мировые координаты, которые являются второй логической системой координат, поддерживаются только в Windows NT/2000. По историческим причинам в Windows под логической системой координат обычно понимается именно страницная система.

При отображении страницных координат в координаты устройства используются два базовых понятия компьютерной графики:

- окно (window);*
- область просмотра (viewport).*

Иногда можно встретить другие термины для обозначения этих же понятий, когда окно называют *логической областью вывода*, а область просмотра — *физической областью вывода*.

Окном называется любая прямоугольная область в страницной системе координат.

Областью просмотра называется прямоугольная область в системе координат устройства. Таким образом, окно определяет отображаемую часть геометрической модели, а область просмотра — ее расположение на поверхности устройства. Соотношение между размерами определяет масштаб вывода.

Для формализации рассматриваемых преобразований следует ввести несколько понятий.

Окно (*window*) определяется четырьмя переменными в страницных координатах:

- xW0* — абсцисса начала координат (*x-origin*).
- yW0* — ордината начала координат (*y-origin*).
- xWE* — горизонтальные габариты окна (*x-extent*).
- yWE* — вертикальные габариты окна (*y-extent*).

Область просмотра (*viewport*) определяется четырьмя переменными в координатах устройства:

- xV0* — абсцисса начала координат (*x-origin*).
- yV0* — ордината начала координат (*y-origin*).
- xVE* — горизонтальные габариты области просмотра (*x-extent*).
- yVE* — вертикальные габариты области просмотра (*y-extent*).

Точка (x, y) в страничной системе координат отображается на точку (x', y') в координатах устройства по следующим формулам:

$$x' = (x - \text{xWO}) \cdot \text{xVE} / \text{xWE} + \text{xVO};$$

$$y' = (y - \text{yWO}) \cdot \text{yVE} / \text{yWE} + \text{yVO}.$$

Существует несколько вариантов отображений логической области вывода в физическую область вывода.

- **Тождественное отображение.** Окно и область просмотра задаются квартетами $(0, 0, 1, 1)$. В этом случае $x' = x$, $y' = y$, а страничные координаты идентичны координатам устройства.
- **Смещение.** Окно определяется квартетом $(0, 0, 1, 1)$, а область просмотра — квартетом $(dx, dy, 1, 1)$. В этом случае $x' = x + dx$, $y' = y + dy$. Каждая точка страничного пространства при отображении в систему координат устройства смещается на величину (dx, dy) .
- **Масштабирование.** Окно определяется квартетом $(0, 0, 1, 1)$, а область просмотра — $(0, 0, mx, my)$. В этом случае $x' = x \cdot mx$, $y' = y \cdot my$. Каждая точка страничного пространства при отображении в систему координат устройства масштабируется с коэффициентами (mx, my) .

- **Отражение.** Окно определяется квартетом $(0, 0, w, h)$, а область просмотра — квартетом $(w, h, -w, -h)$. В этом случае $x' = w - x$, а $y' = h - y$. При отображении из страничной системы координат в координаты устройства рисунок может подвергаться зеркальному отражению относительно горизонтальной и вертикальной осей. Отражение позволяет использовать в страничной системе координат направления осей, которые отличаются от фиксированных направлений системы координат устройства.

Эти варианты отображений можно комбинировать в необходимой последовательности.

В Win32 API имеются соответствующие функции для настройки страничной системы координат при помощи задания параметров окна и области просмотра:

```
BOOL SetWindowOrgEx (
    HDC hdc,           // дескриптор контекста устройства
    int X,             // новая x-координата начала координат окна
    int Y,             // новая y-координата начала координат окна
    LPPOINT lpPoint // указатель на структуру типа POINT для
                     // сохранения прежнего начала координат окна
);
BOOL SetWindowExtEx (
    HDC hdc,           // дескриптор контекста устройства
    int nXExtent,     // новый размер окна по оси X
    int nYExtent,     // новый размер окна по оси Y
    LPSIZE lpSize    // указатель на структуру типа SIZE для
                     // сохранения прежних размеров окна
);
```

В приведенных функциях, настраивающих логическую область вывода, все координаты задаются в логических единицах. Но есть также функции, настраивающие физическую область вывода. В их вызовах все координаты задаются в физических единицах:

```
BOOL SetViewportOrgEx (
    HDC hdc,           // дескриптор контекста устройства
```

```

int X,           // новая X-координата начала координат области просмотра
int Y,           // новая Y-координата начала координат области просмотра
LPOINT lpPoint // прежнее начало координат области просмотра
);
BOOL SetViewportExtEx (
    HDC hdc,          // дескриптор контекста устройства
    int nXExtent,     // новый размер области просмотра по оси X
    int nYExtent,     // новый размер области просмотра по оси Y
    LPSIZE lpSize    // прежние размеры области просмотра
);

```

Для всех четырех функций в качестве последнего параметра можно передать значение `NULL`, если сам параметр не используется.

Использование функций `SetWindowOrgEx` и `SetViewportOrgEx` требует дополнительных пояснений. Первая функция устанавливает, какая именно точка логического пространства отображается в точку $(0, 0)$ физического пространства¹. Соответственно, вторая функция устанавливает, какая точка физического пространства отображается в точку $(0, 0)$ логического пространства. Но учтите, что одновременное использование обеих функций запрещено! Остановите свой выбор на одной из них, так как обе функции позволяют добиться одного и того же результата.

Пример использования приведенных функций для настройки страничной системы координат можно найти в листинге 2.3.

Для упрощения работы программиста Win32 API содержит несколько заготовок страничных систем координат, называемых *режимами отображения* (*mapping modes*).

В большинстве режимов отображения устанавливаются заранее выбранные габариты окна и области просмотра. Эти габариты определяют размер единицы измерения в страничной системе координат и коэффициент масштабирования при переходе к системе координат устройства. Впрочем, приложение может изменить положение начала координат окна и области просмотра, что позволяет выводить различные фрагменты геометрической модели в разных частях экрана.

Режим отображения контекста устройства выбирается следующей функцией:

```
int SetMapMode (HDC hDC, int iMapMode);
```

В этой функции параметр `iMapMode` задает один из восьми идентификаторов режимов отображения, приведенных в табл. 2.1.

Таблица 2.1. Режимы отображения

Режим отображения	Направление по оси x	Направление по оси y	Логические единицы
MM_TEXT	Вправо	Вниз	Пиксель
MM_LOMETRIC	Вправо	Вверх	0,1 мм
MM_HIMETRIC	Вправо	Вверх	0,01 мм
MM_LOENGLISH	Вправо	Вверх	0,01 дюйма
MM_HIENGLISH	Вправо	Вверх	0,001 дюйма
MM_TWIPS	Вправо	Вверх	1 / 1440 дюйма
MM_ISOTROPIC	Любое	Любое	Произвольные единицы ($x == y$)
MM_ANISOTROPIC	Любое	Любое	Произвольные единицы ($x != y$)

¹ Это эквивалентно такому сдвигу осей, в результате которого точка $(0, 0)$ логического пространства уже не относится к левому верхнему углу окна.

Если функция `SetMapMode` не вызывалась, то по умолчанию используется режим отображения `MM_TEXT`.

Таким образом, страничная система координат позволяет приложению строить свою геометрическую модель в произвольных координатах с произвольно выбранными направлениями осей и физическим масштабом.

Следует отметить, что реальность, однако, не столь радужна, как может показаться из последнего утверждения. Все получается великолепно до тех пор, пока вы отображаете отдельные точки при помощи функции `SetPixel` либо используете другие графические функции с первом толщиной 1 пиксел. Для этого при вызове функции `CreatePen` аргумент `nWidth` должен иметь нулевое значение. Но при попытке рисовать линии разной толщины вы столкнетесь с неприятными проблемами. Эти проблемы и пути их решения рассматриваются ниже в разделе «Вывод временной диаграммы напряжения переменного электрического тока» (листинг 2.3).

Мировая система координат

Страничная система координат имеет несколько недостатков, причиной которых являются следующие особенности архитектуры Win16 GDI API:

- ❑ Дробные коэффициенты при отображении окна на область просмотра, что может приводить к потере точности.
- ❑ Неполная реализация API. Так, вывод текста не соответствует семантике отражения относительно оси *Y*. Другими словами, если приложение направляет ось *X* справа налево, то текстовые строки все равно выводятся слева направо.
- ❑ Ограниченный набор преобразований. Отображение окна на область просмотра позволяет выполнить масштабирование, перемещение и зеркальное отражение. Вращение и сдвиг не поддерживаются, а без прямой поддержки со стороны GDI реализовать их очень трудно.

В Windows NT/2000 для решения этих проблем была создана новая логическая система координат — *мировые координаты*. В этой системе координаты задаются по-прежнему в виде 32-разрядных целых чисел, однако появляется возможность использования более общих преобразований.

Аффинные преобразования

Преобразования, поддерживаемые в Windows NT/2000, относятся к классу двумерных аффинных преобразований. Аффинное преобразование отображает параллельные линии в параллельные линии, а конечные точки — в конечные точки.

Двумерное аффинное преобразование определяется шестью числами, образующими матрицу 2×3 . В Win32 API такие матрицы определяются структурой `XFORM`:

```
typedef struct _XFORM {  
    FLOAT eM11;  
    FLOAT eM12;  
    FLOAT eM21;  
    FLOAT eM22;  
    FLOAT eDx;  
    FLOAT eDy;  
} XFORM;
```

Аффинное преобразование, определяемое этими числами, преобразует точку (x, y) в точку (x', y') следующим образом:

$$x' = eM11 \cdot x + eM21 \cdot y + eDx;$$

$$y' = eM12 \cdot x + eM22 \cdot y + eDy.$$

Аффинные преобразования позволяют выполнить следующие операции:

- *Тождественное отображение.* Оно определяется матрицей $(1, 0, 0, 1, 0, 0)$. При этом $x' = x$, а $y' = y$.
- *Смещение.* Это преобразование определяется матрицей $(1, 0, 0, 1, dx, dy)$. При этом $x' = x + dx$, а $y' = y + dy$.
- *Масштабирование.* Задается матрицей $(mx, 0, 0, my, 0, 0)$. Выполняются соотношения $x' = mx \cdot x$, $y' = my \cdot y$.
- *Зеркальное отражение.* Преобразование задается матрицей $(-1, 0, 0, -1, 0, 0)$. При этом $x' = -x$, а $y' = -y$.
- *Поворот.* Определяется матрицей $(\cos(\theta), \sin(\theta), -\sin(\theta), \cos(\theta), 0, 0)$. При этом существуют соотношения $x' = \cos(\theta) \cdot x - \sin(\theta) \cdot y$ и $y' = \sin(\theta) \cdot x + \cos(\theta) \cdot y$.
- *Сдвиг.* Определяется матрицей $(1, s, 0, 1, 0, 0)$ ¹. При этом выполняется соотношения $x' = x + s \cdot y$ и $y' = y$.
- *Комбинированные операции.* Матрицы нескольких аффинных преобразований объединяются операцией матричного умножения и образуют новое аффинное преобразование.

Функции мировых преобразований в Win32 API (Windows NT/2000)

По умолчанию контекст устройства работает в так называемом *совместимом режиме*. При этом имеется в виду совместимость с 16-разрядной семантикой GDI. В совместимом режиме мировые координаты не поддерживаются, а единственной логической системой координат является страничная система. Для использования мировой системы координат приложение должно переключить контекст устройства в графический режим вызовом функции `SetGraphicsMode(hDC, GM_ADVANCED)`. В результате вызова будет обеспечена поддержка двух уровней логического координатного пространства — мировых и страничных координат, а также матрицы преобразования. Чтобы вернуться к совместимому режиму работы, следует заполнить матрицу данными тождественного преобразования и вызвать функцию `SetGraphicsMode(hDC, GM_COMPATIBLE)`. Кроме того, можно воспользоваться функциями `SaveDC` и `RestoreDC`.

В контексте устройства преобразование из мировых координат в страничные по умолчанию инициализируется тождественной матрицей. Для изменения текущего преобразования используются следующие функции:

```
BOOL SetWorldTransform(HDC hDC, CONST XFORM* lpXform);
```

```
BOOL ModifyWorldTransform(HDC hDC, CONST XFORM* lpXform, DWORD iMode);
```

Функция `SetWorldTransform` просто заменяет атрибут преобразования в контексте устройства новым преобразованием, заданным структурой `XFORM`.

¹ Координаты x смещаются на величину, пропорциональную y .

При вызове функции `ModifyWorldTransform` параметр `iMode` может принимать одно из трех предопределенных значений:

- ❑ `MTW_IDENTITY` — преобразование приводится к тождественной форме;
- ❑ `MTW_LEFTMULTIPLY` — текущее преобразование умножается на матрицу параметра `lpXform` слева;
- ❑ `MTW_RIGHTMULTIPLY` — текущее преобразование умножается на матрицу параметра `lpXform` справа.

Для получения информации о текущем преобразовании используется функция `GetWorldTransform`.

Пример использования мировой системы координат приведен в главе 12.

Получение информации о возможностях устройства

Иногда бывает необходимо уточнить характеристики конкретного графического устройства. Для этого можно воспользоваться функцией `GetDeviceCaps`, которая возвращает информацию об атрибутах или возможностях графического устройства на основе целочисленного индекса, который передается в качестве параметра:

```
int GetDeviceCaps(HDC hdc, int nIndex);
```

Некоторые возможные значения индекса `nIndex` приведены в табл. 2.2¹.

Таблица 2.2. Значения параметра `nIndex`

Индекс	Значение
<code>DRIVERVERSION</code>	Версия драйвера устройства
<code>TECHNOLOGY</code>	Технология устройства (DT_PLOTTER, DT_RASDISPLAY, DT_RASPRINTER и другие значения)
<code>HORZSIZE</code>	Ширина физического экрана в миллиметрах
<code>VERTSIZE</code>	Высота физического экрана в миллиметрах
<code>HORZRES</code>	Ширина экрана в пикселях
<code>VERTRES</code>	Высота экрана в пикселях
<code>LOGPIXELSX</code>	Число пикселов на 1 логический дюйм по горизонтали
<code>LOGPIXELSY</code>	Число пикселов на 1 логический дюйм по вертикали
<code>BITSPIXEL</code>	Число смежных битов цвета для каждого пикселя
<code>PLANES</code>	Число цветовых плоскостей
<code>NUMBRUSHES</code>	Число кистей, зависящих от устройства
<code>NUMPENS</code>	Число перьев, зависящих от устройства
<code>NUMFONTS</code>	Число шрифтов, зависящих от устройства
<code>NUMCOLORS</code>	Число входов в таблице цветов устройства
<code>CLIPCAPS</code>	Флажок, который указывает возможности отсечения устройства
<code>SIZEPALETTE</code>	Число входов в системной палитре

¹ Полный список возможных значений индекса см. в MSDN.

Индекс	Значение
COLORRES	Фактическая разрешающая способность устройства по цвету в битах на пиксель
PHYSICALWIDTH	Ширина физической страницы (для принтеров) в пикселях
PHYSICALHEIGHT	Высота физической страницы (для принтеров) в пикселях
PHYSICALOFFSETX	Смещение в пикселях от левой границы страницы, где фактически начинается печать
PHYSICALOFFSETY	Смещение в пикселях от верхней границы страницы, где фактически начинается печать
RASTERCAPS	Растровые возможности устройства. Возвращаемое значение может представлять собой комбинацию следующих флагов: RC_BITBLT — способно передавать растровые изображения; RC_BITMAP64 — способно поддерживать растровые изображения, превышающие по размерам 64 Кбайт; RC_DEVBITS — способно поддерживать аппаратно-зависимые изображения; RC_NONE — устройство не имеет растровых возможностей; RC_PALETTE — указывает на устройство, действующее на основе палитры; RC_SCALING — способно к масштабированию; RC_STRETCHBLT — способно выполнять функцию StretchBlt; RC_STRETCHDIB — способно выполнять функцию StretchDibits
CURVECAPS	Возможности вывода кривых. Возвращаемое значение может представлять собой комбинацию следующих флагов: CC_CHORD — устройство может выводить дуги хорды; CC_CIRCLES — может выводить окружности; CC_ELLIPSWS — может выводить эллипсы; CC_INTERIORS — может закрашивать внутренние области; CC_NONE — устройство не поддерживает вывод кривых
LINECAPS	Возможности вывода линий. Возвращаемое значение может представлять собой комбинацию следующих флагов: LC_STYLED — устройство может выводить стилизованные линии; LC_WIDE — может выводить широкие линии; LC_POLYLINE — может выводить ломаную линию; LC_INTERIORS — может закрашивать внутренние области; LC_NONE — устройство не поддерживает линии
POLYGONALCAPS	Возможности вывода многоугольников. Возвращаемое значение может представлять собой комбинацию следующих флагов: PC_STYLED — устройство может выводить стилизованные рамки; PC_WIDE — может выводить широкие рамки; PC_POLYGON — может выводить многоугольники; PC_INTERIORS — может закрашивать внутренние области; PC_NONE — устройство не поддерживает многоугольники
TEXTCAPS	Текстовые возможности устройства. Возвращаемое значение может представлять собой комбинацию следующих флагов: TC_CR_90 — устройство способно поворачивать символы на 90°; TC_CR_ANY — способно поворачивать символы на любой угол; TC_EA_DOUBLE — может выводить символы с двойным весом; TC_IA_ABLE — может выводить символы курсивом; TC_RA_ABLE — может выводить растровые шрифты; TC_VA_ABLE — может выводить векторные шрифты

Например, если необходимо определить количество пикселов на 1 логический дюйм экрана по вертикали, чтобы рассчитать необходимые размеры создаваемого шрифта, то это можно сделать в блоке обработки сообщения WM_CREATE:

```
HDC hdc = GetDC(hwnd);
int logPixelY = GetDeviceCaps(hdc, LOGPIXELY);
// используем значение logPixelY ...
ReleaseDC(hwnd, hdc);
```

Управление цветом. Вывод пикселя

При внимательном рассмотрении можно увидеть что экран дисплея составлен из тонких горизонтальных и вертикальных линий. На пересечении каждой горизонтальной и вертикальной линии находится точка, называемая Пикселом. *Пиксель* — это минимальный по размерам изобразительный элемент, которым может управлять приложение.

В свою очередь, пиксель образован тремя микроточками, не различимыми невооруженным глазом, которые имеют, соответственно, красный, зеленый и синий цвета. Микроточки могут светиться с интенсивностью от 0 (отсутствие излучения) до 255 (максимальная яркость). Совокупность их свечения образует текущий цвет пикселя.

Таким образом, цвет пикселя можно рассматривать как некоторую точку в трехмерном RGB-пространстве, образованном тремя цветовыми осями: Red (красная цветовая составляющая), Green (зеленая составляющая) и Blue (синяя составляющая).

В компьютерной графике используются и другие цветовые пространства. Например, для цветных принтеров чаще используется цветовое пространство CMYK, в котором каждый цвет является комбинацией голубой, малиновой, желтой и черной составляющих.

Для человеческого глаза более привычно различать цвета по их оттенку, яркости и насыщенности. Поэтому в графических пакетах более распространенным является использование пространства HLS.

Цветовое пространство HLS

В пространстве HLS цвет описывается через значения оттенка (*Hue*), яркости (*Lightness*) и насыщенности (*Saturation*).

Оттенок измеряется в угловых величинах от 0 до 360°. Нулевое значение соответствует синему цвету, 60 — малиновому, 120 — красному, 180 — желтому, 240 — зеленому, 300 — голубому.

Яркость описывает интенсивность чистого цвета. Значение яркости находится в промежутке от нуля до единицы. Условно говоря, увеличение яркости добавляет белый цвет к чистому цвету, а уменьшение яркости добавляет черный цвет.

Насыщенность, также изменяемая от нуля до единицы, является мерой «чистоты» цвета. Если насыщенность уменьшается, то к чистому цвету добавляется больше серого цвета. Нулевое значение насыщенности приводит к шкале серого цвета.

Цветовое пространство RGB

Система Windows поддерживает цветовое пространство RGB и пространство индексов палитры, соответствующее базовым возможностям видеоадаптера.

Цвет кодируется в любом из этих пространств при помощи типа данных COLORREF, представляющего четырехбайтное значение. Старший байт задает один из трех возможных форматов: 0 — RGB, 1 — PALETTEINDEX, 2 — PALETTERGB.

Интерпретация формата RGB показана на рис. 2.3.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Blue								Green								Red														

Рис. 2.3. Тип данных COLORREF, формат RGB

Минимальное значение (0) всех трех составляющих цвета задает черный цвет. Максимальное значение (255) всех трех составляющих цвета задает белый цвет. Комбинируя все возможные значения RGB-составляющих, можно получить 2^{24} комбинаций, или примерно 16,7 миллиона возможных цветов.

Win32 API содержит несколько макросов, позволяющих объединять три составляющие RGB в одно 32-разрядное значение типа COLORREF либо, наоборот, разделять данные COLORREF на составляющие RGB-модели.

Эти макросы, определенные в файле wingdi.h, можно представить в виде следующих подставляемых (*inline*) функций:

```
COLORREF RGB(BYTE byRed, BYTE byGreen, BYTE byBlue);
BYTE GetRValue(COLORREF rgb);
BYTE GetGValue(COLORREF rgb);
BYTE GetBValue(COLORREF rgb);
```

Следующий фрагмент кода демонстрирует использование макроса RGB для определения констант, представляющих наиболее часто применяемые «чистые» цвета:

```
const COLORREF black = RGB(0, 0, 0);           // RGB(0x0, 0x0, 0x0);
const COLORREF red = RGB(255, 0, 0);            // RGB(0xFF, 0x0, 0x0);
const COLORREF green = RGB(0, 255, 0);           // RGB(0x0, 0xFF, 0x0);
const COLORREF blue = RGB(0, 0, 255);            // RGB(0x0, 0x0, 0xFF);
const COLORREF yellow = RGB(255, 255, 0);         // RGB(0xFF, 0xFF, 0x0);
const COLORREF magenta = RGB(255, 0, 255);        // RGB(0xFF, 0x0, 0xFF);
const COLORREF cyan = RGB(0, 255, 255);           // RGB(0x0, 0xFF, 0xFF);
const COLORREF white = RGB(255, 255, 255);         // RGB(0xFF, 0xFF, 0xFF);
```

Определение цвета при работе с палитрой

К сожалению, не все устройства способны воспроизводить 16 777 216 цветов из RGB-пространства типа COLORREF. Например, все еще встречаются дисплеи старых типов, которые могут поддерживать только 256 цветов. В таких случаях для приемлемого качества рисования нужно работать с цветовой палитрой. Вопросы работы с палитрой рассматриваются в главе 3. Здесь же приводятся только форматы данных для типа COLORREF, которые позволяют работать с палитрой.

Win32 API содержит макросы PALETTEINDEX и PALETTERRGB, формирующие значения одноименных форматов для типа данных COLORREF. Интерпретация этих форматов показана на рис. 2.4.

PALETTEINDEX (index)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0								index																						

PALETTERRGB (Red, Green, Blue)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
2	Blue								Green								Red														

Рис. 2.4. Тип данных COLORREF, форматы PALETTEINDEX и PALETTERRGB

Вывод пикселов

Для работы с пикселями предусмотрены следующие функции:

```
COLORREF GetPixel(HDC hdc, int X, int Y);  
BOOL SetPixelV(HDC hdc, int X, int Y, COLORREF color);  
COLORREF SetPixel(HDC hdc, int X, int Y, COLORREF color);
```

Параметры *X* и *Y* определяют позицию пикселя в логических единицах. Параметр *color*, устанавливающий цвет пикселя, обычно задается при помощи макрона *RGB*¹.

Функция *GetPixel* возвращает цветовое значение пикселя с заданными координатами.

Функции *SetPixelV* и *SetPixel* устанавливают заданное цветовое значение пикселя, различаясь возвращаемым значением. Первая функция возвращает логическое значение, указывающее, успешно ли прошла операция. Вторая функция в случае успешного выполнения возвращает старый цвет пикселя, а если произошла какая-то ошибка, то возвращается значение *-1*.

Реализация несложных, на первый взгляд, функций *SetPixelV* и *SetPixel* на самом деле связана с существенными затратами процессорного времени. Обе функции инициируют вызов системной функции *NtGdiSetPixel*, которая, в свою очередь, осуществляется довольно много действий:

- ❑ блокировку контекста устройства;
- ❑ отображение логических координат в физические;
- ❑ преобразование в случае необходимости значения типа *COLORREF* в индекс;
- ❑ вызов функции драйвера *DrvBitBlt*;
- ❑ разблокировку контекста устройства.

Хронометраж процесса выполнения этих функций [6] показал, что процесс вывода одного пикселя требует более 1000 тактов работы процессора. Как ни странно, функция *GetPixel* работает еще медленнее и требует более 6000 тактов работы процессора.

Поэтому, если рисование ведется посредством вывода отдельных пикселов, могут возникнуть проблемы с быстродействием программы. Решение таких проблем рассматривается в главе 12.

Атрибуты контекста устройства, влияющие на рисование

В этом разделе будут рассматриваться атрибуты рисования, к которым в первую очередь относятся режим рисования, режим смещивания фона и цвет фона графических элементов.

¹ Если приложение работает с логической палитрой, то вместо макрона *RGB* следует использовать макрос *PALETERGB*.

Режим рисования. Бинарные растровые операции

Любую операцию рисования можно разложить на элементарные акты рисования отдельных пикселов. В предыдущем разделе упоминалась функция `SetPixel(hDC, x, y, color)`, которая изменяет текущий цвет пикселя с координатами x и y на цвет `color`. Но такое определение функции `SetPixel` не является абсолютно корректным. Точнее говоря, оно корректно, если приложение не изменяет режим рисования, установленный по умолчанию в контексте устройства.

На самом деле в Windows реализована более гибкая технология установки цвета пикселя при рисовании. Эту технологию можно представить как процесс передачи цвета из *источника S* (инструмента рисования) в *приемник D* («холст» или поверхность устройства). Независимо от разновидности инструмента рисования (перо, кисть или функция `SetPixel`) результат такой передачи определяется как функция $f(D, S)$, то есть представляет собой некоторую комбинацию цвета источника и цвета приемника:

$$D = f(D, S).$$

Теоретически, число таких функций является бесконечным, но в GDI реализованы только поразрядные логические операции. Слово «поразрядные» означает, что к битам двух аргументов, находящимся в одинаковой позиции, применяется одна и та же логическая операция. В Windows поразрядная логическая операция, применяемая к пикселям, носит название *растровой операции* (*raster operation* (ROP)). А поскольку эта операция применяется к двум аргументам, то ее называют *бинарной растровой операцией* (*binary raster operation* (ROP2)).

В MSDN при описании бинарных растровых операций в качестве источника рассматривается только один инструмент рисования — перо `P`, поэтому приведенная выше формула имеет следующий вид:

$$D = f(D, P).$$

Однако все бинарные растровые операции, приведенные в табл. 2.3, имеют аналогичное действие и для альтернативных инструментов рисования, таких как кисть или функция `SetPixel`.

Таблица 2.3. Бинарные растровые операции

P: D:	1 1 0 0 1 0 1 0	Формула	ROP2	Описание
Р е з у л ь т а т ы	0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 1 0 1 0 0 0 1 0 1 0 1 1 0 0 1 1 1 1 0 0 0	D = 0 D = ~(D P) D = D & ~P D = ~P D = P & ~D D = ~D D = D ^ P D = ~(D & P) D = D & P	R2_BLACK R2_NOTMERGESEN R2_MASKNOTPEN R2_NOTCOPYPEN R2_MASKPENNOT R2_NOT R2_XORPEN R2_NOTMASKPEN R2_MASKPEN	Всегда 0 (черный цвет) Инверсия R2_MERGESEN Конъюнкция приемника с инвертированным пером Инверсия цвета пера Конъюнкция пера с инвертированным приемником Инверсия приемника Исключающее «ИЛИ» для приемника и пера Инверсия R2_MASKPEN Конъюнкция приемника с пером

продолжение ➔

Таблица 2.3 (продолжение)

P:	1 1 0 0	Формула	ROP2	Описание
D:	1 0 1 0			
е	1 0 1 0	$D = D$	R2_NOP	Приемник не изменяется
з	1 0 1 1	$D = D \mid \sim P$	R2_MERGENOTPEN	Дизъюнкция приемника с инвертированным пером
у				
л				
ь	1 1 0 0	$D = P$	R2_COPYPEN	Цвет пера (значение по умолчанию)
т	1 1 0 1	$D = P \mid \sim D$	R2_MERGEPPENNOT	Дизъюнкция пера с инвертированным приемником
а				
т				
ы	1 1 1 0	$D = P \mid D$	R2_MERGEPPEN	Дизъюнкция пера с приемником
	1 1 1 1	$D = 1$	R2_WHITE	Всегда 1 (белый цвет)

Контекст устройства GDI содержит атрибут, называемый *режимом рисования* (*drawing mode*). Он определяет текущую операцию ROP2.

Режим рисования можно установить при помощи функции SetROP2:

```
int SetROP2(HDC hdc, int fnDrawMode);
```

При вызове функции параметру fnDrawMode передается одно из значений ROP2, приведенных в табл. 2.3. Функция возвращает предыдущий режим рисования. Для получения текущего режима рисования можно воспользоваться функцией GetROP2.

По умолчанию в контексте устройства установлен режим R2_COPYPEN, при котором пикселу приемника просто присваивается цвет пера.

При использовании бинарных растровых операций следует помнить о том, что если в устройстве используется цветовое пространство RGB, то операции применяются к каждой из трех составляющих RGB-модели. Поэтому результат всегда предсказуем, но не всегда может быть оправдан с точки зрения логики цветового восприятия. Например, если в контексте устройства выбрано перо красного цвета (RGB(0xFF, 0, 0)) и установлен режим рисования R2_XORPEN, то при рисовании на белом фоне (RGB(0xFF, 0xFF, 0xFF)) результирующая линия будет иметь цвет морской волны (субан), поскольку операция «Исключающее ИЛИ» для приемника и пера даст результат RGB(0, 0xFF, 0xFF). Интересно, что при повторном выводе линии на том же месте будет нарисована линия белого цвета, то есть восстановится первоначальное состояние приемника. На этом полезном свойстве режима рисования R2_XORPEN основан вывод «эластичных прямоугольников».

Для устройств с палитрой растровые операции применяются к цветовым индексам, поэтому результат зависит от упорядочения цветов в палитре.

Бинарные растровые операции играют важную роль в компьютерной графике. Режим R2_BLACK используется для закраски пикселов черным цветом, а режим R2_WHITE — белым цветом. Режим R2_NOTCOPYPEN меняет цвет пера на противоположный. Режим R2_NOP полностью подавляет вывод линий и кривых — это удобно, если вы не хотите обводить прямоугольник рамкой.

Режимы R2_XORPEN и R2_NOTXORPEN часто используются в интерактивной компьютерной графике для вывода эластичных контуров или движущихся изображений. Эластичные прямоугольники, например, используются для выделения с помощью мыши некоторых областей на изображении. В процессе перемещения мыши построение фигуры считается не законченным, поэтому приложение должно быстро стереть прежнюю версию контура и нарисовать его в новой позиции. Бинарные операции R2_XORPEN и R2_NOTXORPEN позволяют легко реализовать эту функциональность, поскольку при повторном выводе линии восстанавливается

исходное содержимое изображения. Пример вывода эластичных прямоугольников приведен в четвертой главе.

Режим смешивания фона и цвет фона графического элемента

Некоторые графические примитивы GDI содержат пиксели двух видов: основные (*foreground*) и фоновые (*background*). Например, при выводе текстового символа пиксели, образующие глиф¹ символа, считаются основными, а остальные пиксели из прямоугольной области вокруг символа² считаются фоновыми. При выводе пунктирных линий пиксели отрезков считаются основными, а пиксели промежутков — фоновыми. Для шаблона штриховой кисти штриховые линии рисуются основными пикселями, а промежутки между линиями — фоновыми пикселями.

Фоном графического элемента будем в дальнейшем считать совокупность его фоновых пикселов. Цвет фона графических элементов (*background color*) является атрибутом контекста устройства, который можно устанавливать с помощью функции `SetBkColor`.

По умолчанию этот атрибут имеет значение белого цвета. Но не следует путать понятие цвета фона графических элементов, который относится к контексту устройства, с понятием цвета фона окна, который является атрибутом класса окна. Чаще всего фон класса окна также имеет белый цвет. Именно поэтому при выводе текста в программе `Hello1` нельзя было заметить цвет фона для букв текста. Он просто слился с фоном окна.

Если основные пиксели графического элемента выводятся всегда, то вывод фоновых пикселов зависит от еще одного атрибута контекста устройства, который называется *режимом смешивания фона* (*background mix mode*)³.

По умолчанию режим смешивания фона имеет значение `OPAQUE`⁴. Этот режим указывает, что Windows выводит фон графического элемента (цветом *background color*) поверх фона окна. Разработчик может изменить режим смешивания фона, запретив отображение фона графических элементов. Для этого нужно вызвать функцию `SetBkMode`, передав ей в качестве второго параметра константу `TRANSPARENT`⁵.

Чтобы узнать, какой режим смешивания фона является текущим, следует вызвать функцию `GetBkMode`.

Общие операции с графическими объектами

Как уже упоминалось ранее, GDI содержит набор графических объектов, обеспечивающих выполнение графических операций. К таким объектам относятся пе-

¹ Глиф (*glyph*) — графическая форма символа.

² Эту область иногда называют *ячейкой символа*.

³ В литературе встречается другой перевод этого термина — *режим заполнения фона*.

⁴ Непрозрачный режим смешивания фона.

⁵ Прозрачный режим смешивания фона.

рья, кисти, растровые изображения, палитры, шрифты. Использование любого графического объекта предполагает выполнение следующей последовательности операций:

1. Создать графический объект.
2. Выбрать созданный объект в контекст устройства.
3. Вызвать графическую функцию, работающую с объектом.
4. Удалить объект из контекста устройства, вернув предшествующий объект.
5. Уничтожить объект.

Для создания GDI-объектов предназначены соответствующие функции `Create...`, которые в случае успешного завершения возвращают дескриптор объекта.

Выбор объекта в контекст устройства осуществляется при помощи функции `SelectObject` (палитры выбираются с помощью функции `SelectPalette`). Функция `SelectObject` имеет следующий прототип:

```
HGDIOBJ SelectObject(  
    HDC hdc,           // дескриптор контекста устройства  
    HGDIOBJ hgdiobj   // дескриптор GDI-объекта  
) ;
```

В результате ее выполнения новый объект `hgdiobj` заменяет предшествующий объект того же типа в контексте устройства `hdc`.

Функция возвращает дескриптор предшествующего объекта. Для корректной работы приложение должно запомнить этот дескриптор и после окончания рисования с новым объектом (шаг 3) вернуть в контекст устройства предшествующий объект (шаг 4).

Для уничтожения объекта, ставшего ненужным, вызывается функция `DeleteObject`.

Следует отметить, что не всегда для создания GDI-объекта нужно вызывать соответствующую функцию типа `Create....`. В системе имеется набор *предопределенных (стандартных) графических объектов*, и если параметры предопределенного объекта (перо, кисть, шрифт и т. д.) подходят для решаемой задачи, то приложение может получить такой объект с помощью функции `GetStockObject`. После окончания работы с предопределенным объектом его не нужно удалять при помощи функции `DeleteObject`. Предопределенные объекты существуют в системе постоянно.

Линии и кривые

Теоретически, рисование любой линии можно было бы свести к многократному вызову функции `SetPixel` с соответствующим изменением координат `x` и `y`. Однако, во-первых, это не очень удобно для программиста, а во-вторых, подобный способ рисования был бы очень медленным. Значительно более эффективной в графических системах является реализация функций рисования отрезков и других сложных графических операций на уровне драйвера устройства, который содержит код, оптимизированный для этих операций. Поэтому Win32 GDI предоставляет соответствующий набор графических функций.

Рисование отрезков

Любая линия рисуется в Windows с использованием графического объекта, называемого *пером*. Контекст устройства содержит перо по умолчанию — сплошное перо черного цвета толщиной 1 пиксель. Многие графические функции начинают рисование с так называемой текущей позиции пера.

Текущая позиция пера

Текущая позиция пера является одним из атрибутов контекста устройства. Она определяется значением типа POINT. По умолчанию текущая позиция пера установлена в точку (0, 0). Если нужно изменить текущую позицию, вызывается функция MoveToEx:

```
BOOL MoveToEx(
    HDC hdc,           // дескриптор контекста устройства
    int X,             // x-координата новой текущей позиции
    int Y,             // y-координата новой текущей позиции
    LPOINT lpPoint    // предыдущая позиция пера
);
```

Если предыдущая позиция пера вас не интересует, передавайте последнему параметру значение NULL. В случае успешного завершения функция возвращает ненулевое значение.

В любой момент можно получить значение текущей позиции пера, вызвав функцию

```
GetCurrentPositionEx(hdc, &pt);
```

Результат выполнения функции помещается в переменную pt типа POINT.

Рисование прямой линии

Для создания прямой линии используется функция LineTo:

```
BOOL LineTo(
    HDC hdc,           // дескриптор контекста устройства
    int xEnd,          // x-координата конечной точки
    int yEnd           // y-координата конечной точки
);
```

Эта функция рисует отрезок, начиная с точки, в которой находится текущая позиция пера, до точки (xEnd, yEnd), не включая последнюю точку в отрезок. Координаты конечной точки задаются в логических единицах. Если функция завершается успешно, то она возвращает ненулевое значение, а текущая позиция пера устанавливается в точку (xEnd, yEnd).

Последнее свойство функции можно использовать, если требуется нарисовать ряд связанных отрезков. Например, можно определить массив из пяти точек, задающих контур прямоугольника, в котором последняя точка совпадает с первой, и нарисовать прямоугольник, как показано в следующем фрагменте кода:

```
{
    POINT pt[5] = {{100,100}, {200,100}, {200,200},
                    {100,200}, {100,100}};
    MoveToEx(hdc, pt[0].x, pt[0].y, NULL);
    for (int i = 0; i < 5; ++i)
        LineTo(hdc, pt[i].x, pt[i].y);
}
```

Вставьте этот фрагмент в код программы Hello2 (глава 1) после вывода текста функцией `DrawText`, чтобы можно было быстро посмотреть, как он работает. Наружные фигурные скобки в данном фрагменте делают возможным объявление локальных переменных внутри блока оператора `switch`.

Рисование связанных отрезков (ломаной линии)

Последовательность связанных отрезков гораздо проще нарисовать с помощью функции `Polyline`:

```
BOOL Polyline(HDC hdc, CONST POINT* lppt, int cPoints);
```

Второй параметр здесь — это адрес массива точек, а третий — количество точек.

Предыдущий пример рисования прямоугольника теперь можно переписать так:

```
{
    POINT pt[5] = {{100,100}, {200,100}, {200,200},
                    {100,200}, {100,100}};
    Polyline(hdc, pt, 5);
}
```

Хотя результат выполнения этого фрагмента будет таким же, следует заметить, что функция `Polyline` не использует текущую позицию пера и не изменяет ее.

Функция `PolylineTo` предназначена для рисования последовательности связанных отрезков:

```
BOOL PolylineTo(HDC hdc, CONST POINT* lppt, DWORD cPoints);
```

Она использует текущую позицию пера для начальной точки и после каждого своего выполнения устанавливает текущую позицию в конец нарисованного отрезка. Если вы захотите применить ее в рассмотренном выше примере, то вызов функции `Polyline` надо заменить двумя инструкциями:

```
MoveToEx(hdc, pt[0].x, pt[0].y, NULL);
PolylineTo(hdc, pt + 1, 4);
```

Функция `PolyPolyline` позволяет нарисовать несколько ломаных линий за один вызов. Она имеет следующий прототип:

```
BOOL PolyPolyline(
    HDC hdc,           // дескриптор контекста устройства
    CONST POINT* lppt, // массив точек для нескольких ломаных линий
    CONST DWORD* lpPolyPoints, // массив с числом точек в каждой ломаной
    DWORD cCount       // количество ломаных линий
);
```

Эта функция не использует и не изменяет текущей позиции пера.

Дуги

Дуги в Windows рисуются как часть эллипса. Размеры и расположение эллипса определяются ограничивающим прямоугольником. Ограничивающий прямоугольник задается координатами левой верхней и правой нижней вершин. Если обозначить эти координаты как $(xLeft, yTop)$ и $(xRight, yBottom)$, тогда центром эллипса будет точка $(x0, y0)$, где $x0 = xLeft + (xRight - xLeft)/2$, а $y0 = yTop + (yBottom - yTop)/2$.

Для рисования дуг предназначены функции `Arc`, `ArcTo` и `AngleArc`. Первые две функции имеют одинаковый набор параметров, поэтому рассмотрим прототип функции `Arc`:

```
BOOL Arc(HDC hdc, int xLeft, int yTop, int xRight, int yBottom,
         int xStart, int yStart, int xEnd, int yEnd);
```

Параметры со второго по пятый задают вершины ограничивающего прямоугольника. Начало и конец дуги определяются начальным и конечным углами, которые задаются косвенно через две дополнительные точки с координатами ($xStart, yStart$) и ($xEnd, yEnd$). Начало дуги — это пересечение эллипса с лучом, который начинается в центре эллипса и проходит через точку ($xStart, yStart$). Конец дуги — это пересечение эллипса с лучом, который начинается в центре эллипса и проходит через точку ($xEnd, yEnd$). Все координаты задаются в логических единицах.

В Windows 95/98 дуга рисуется против часовой стрелки. В Windows NT/2000 направление дуги определяется соответствующим атрибутом в контексте устройства, значение которого можно получить вызовом функции `GetArcDirection` или установить вызовом функции `SetArcDirection`. Когда вы пользуетесь функцией `SetArcDirection`, в качестве второго параметра можно передать одно из значений: `AD_COUNTERCLOCKWISE`, устанавливающее режим рисования против часовой стрелки, либо `AD_CLOCKWISE`, устанавливающее режим рисования по часовой стрелке. По умолчанию в контексте устройства используется значение `AD_COUNTERCLOCKWISE`.

На рис. 2.5 показана дуга, нарисованная при помощи вызова функции

```
Arc(hdc, 100, 100, 400, 300, 50, 50, 300);
```

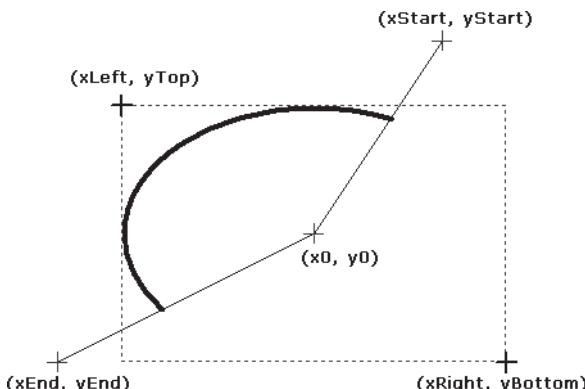


Рис. 2.5. Дуга, нарисованная с использованием функции `Arc` (направление по умолчанию — против часовой стрелки)

Если перед обращением к функции `Arc` вызвать функцию

```
SetArcDirection(hdc, AD_CLOCKWISE);
```

то функция `Arc` нарисует дугу, показанную на рис. 2.6.

Функция `Arc` не использует текущую позицию пера и не обновляет ее.

Функция `ArcTo` отличается от функции `Arc` тем, что она проводит линию из текущей позиции пера в заданную начальную точку дуги, и только после этого рисует дугу. После завершения рисования функция перемещает текущую позицию пера в конечную точку дуги.

Функция `AngleArc`, поддерживаемая только в Windows NT/2000, существенно отличается от двух предыдущих функций способом определения рисуемой дуги, что выражено и в ее прототипе:

```
BOOL AngleArc(HDC hdc, int X, int Y, DWORD radius, FLOAT startAngle,
               FLOAT sweepAngle);
```

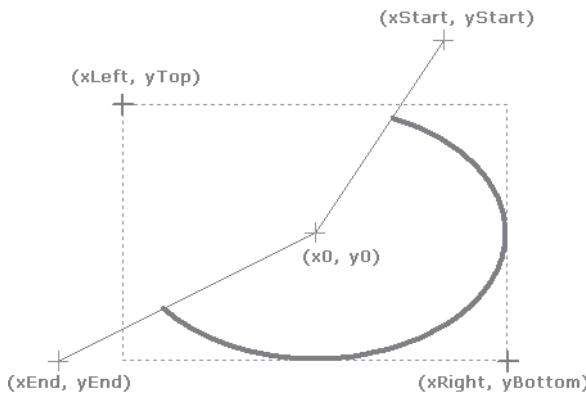


Рис. 2.6. Дуга, нарисованная с использованием функции Arc (направление по часовой стрелке)

Параметры X и Y задают центр круга, а параметр radius — его радиус. Дуга, рисуемая этой функцией, является частью круга. Чтобы нарисовать часть эллипса, приложение должно определить соответствующее преобразование или отображение.

Параметр startAngle определяет начальный угол дуги в градусах, а параметр sweepAngle — длину дуги в градусах. Атрибут контекста устройства, отвечающий за направление рисования дуг, в этой функции не используется. При положительном значении sweepAngle дуга рисуется против часовой стрелки, при отрицательном значении — по часовой стрелке.

Функция AngleArc, так же как и ArcTo, проводит линию из текущей позиции пера в заданную начальную точку дуги, после чего рисует дугу и перемещает текущую позицию пера в конечную точку дуги.

Кривые Безье

В дополнение к рисованию кривых, являющихся частью эллипса, Windows позволяет рисовать нерегулярные кривые, называемые кривыми Безье. *Кривая Безье (сплайн Безье)* — это кубическая кривая, положение и кривизна которой задаются четырьмя определяющими точками p1, p2, p3 и p4. Точка p1 является стартовой точкой, точка p4 — конечной точкой. Точки p2, p3 называются контрольными точками — именно они определяют форму кривой, играя роль «магнитов», оттягивающих линию от прямой, соединяющей p1 и p4. Пример кривой Безье показан на рис. 2.7.

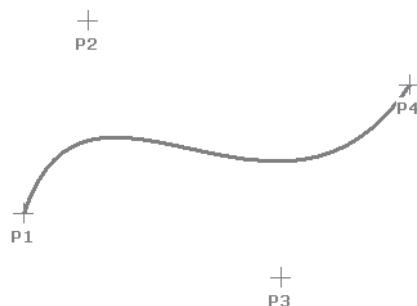


Рис. 2.7. Кривая Безье

Математический аппарат для рисования таких кривых предложил П. Безье (P.Bezier) в ходе разработки систем автоматизированного проектирования для компаний «Ситроен» и «Рено». Впоследствии оказалось, что кривые Безье – очень удобный инструмент для рисования кривых в компьютерной графике.

Win32 GDI содержит две функции, позволяющие рисовать набор связанных кривых Безье за один вызов:

```
BOOL PolyBezier(HDC hdc, CONST POINT* lppr, DWORD cPoints);  
BOOL PolyBezierTo(HDC hdc, CONST POINT* lppr, DWORD cCount);
```

Для рисования n кривых функция `PolyBezier` получает адрес массива `lppr`, содержащего $3n + 1$ точек, при этом параметр `cPoints` должен быть равен $3n + 1$. Первые четыре точки, `lppr[0], lppr[1], lppr[2], lppr[3]`, задают первую кривую. Точка `lppr[3]` вместе со следующими тремя точками определяет вторую кривую и т. д. Функция `PolyBezier` не использует текущую позицию пера и не обновляет ее.

В отличие от `PolyBezier`, функция `PolyBezierTo` получает $3n$ точек в массиве `lppr`, при этом параметр `cCount` должен быть равен $3n$. Функция рисует первую кривую, начиная с текущей позиции пера, до позиции, заданной третьей точкой, используя первые две точки в качестве контрольных. Каждая последующая кривая рисуется так же, как и функцией `PolyBezier`. По окончании рисования текущая позиция пера переводится в последнюю точку из массива `lppr`.

Более подробную информацию об использовании кривых Безье можно найти в издании [6].

Перья

Любая функция рисования линий и кривых, а также контуров замкнутых фигур использует *перо* (*pen*), выбранное в контексте устройства в данный момент. Если вы не выбирали никакого пера, то используется перо по умолчанию `BLACK_PEN`. Оно рисует сплошные черные линии толщиной 1 пиксель независимо от режима отображения. Возможно, вы захотите большего разнообразия при рисовании линий, например, захотите придать линиям разный цвет, разную толщину и даже разный стиль!

Для удовлетворения таких запросов Win32 GDI позволяет создавать объекты логических перьев. *Логическое перо* представляет собой описание требований к перу со стороны приложения. Эти требования могут в каких-то деталях и не соответствовать тому, как будут выводиться линии на поверхности физического устройства. Драйвер графического устройства может поддерживать собственные структуры данных, определяющие реализацию логического пера, — такие внутренние объекты называются *физическими перьями*.

Структура данных логического пера находится под управлением GDI, как и иные логические объекты. Дескриптор созданного пера возвращается приложению и используется для ссылок на перо во время последующей работы.

Дескрипторы объектов GDI описываются общим типом `HGDIOBJ`; для дескрипторов логических перьев зарезервирован тип `HPEN` (*handle to a pen*). Следует заметить, что тип `HGDIOBJ` определен в заголовочных файлах Windows как указатель на тип `void`, а `HPEN` и другие специализированные типы указателей — как указатели

на совершенно разные структуры. Таким образом, типом **HPEN** можно заменить тип **HGDIOBJ**, но попытка использования **HGDIOBJ** вместо **HPEN** требует обязательного преобразования типа.

Итак, объект пера с дескриптором **hPen** объявляется следующим образом:

```
HPEN hPen;
```

Значение дескриптора получают вызовом соответствующей функции. Вид вызываемой функции зависит от типа пера.

Объект логического пера, как и другие объекты GDI, поглощает ресурсы операционной системы. Следовательно, когда необходимость в нем отпадает, рекомендуется исключить его из контекста устройства и удалить функцией **DeleteObject**¹.

Стандартные перья

В Win32 GDI определено четыре типа стандартных перьев, которые приведены в табл. 2.4.

Таблица 2.4. Стандартные перья

Индекс пера	Описание	Примечание
BLACK_PEN	Сплошное черное перо толщиной 1 пиксел	Установлено по умолчанию
WHITE_PEN	Сплошное белое перо толщиной 1 пиксел	
NUL_PEN	Пустое перо (ничего не рисует)	Может использоваться для вывода фигур без внешнего контура
DC_PEN	Перо DC — сплошное перо толщиной 1 пиксел. По умолчанию оно имеет черный цвет. Этот цвет может быть изменен функцией SetDCPenColor	Реализовано только в Windows 2000

Для получения дескриптора стандартного объекта нужно вызвать функцию **GetStockObject**, передав ей индекс стандартного объекта, как показано в следующей инструкции:

```
hPen = (HPEN)GetStockObject(WHITE_PEN);
```

Функция **GetStockObject** возвращает значение типа **HGDIOBJ**, поэтому для корректного присваивания выполняется преобразование типа.

Получив значение дескриптора, нужно выбрать объект пера в контекст устройства:

```
SelectObject(hDC, hPen);
```

После этого все функции, рисующие линии, будут использовать **WHITE_PEN** до тех пор, пока вы не выберете другое перо в контекст устройства или не освободите контекст устройства.

Вместо того чтобы определять переменную **hPen**, можно совместить вызовы **GetStockObject** и **SelectObject** в одной инструкции:

```
SelectObject(hDC, GetStockObject(WHITE_PEN));
```

¹ Стандартные перья удалять не нужно.

Хорошим стилем считается сохранение предшествующего дескриптора объекта и возврат его в контекст устройства после завершения операций рисования с новым объектом, например, как показано в следующем фрагменте кода:

```
HPEN hOldPen = (HPEN)SelectObject(hdc, GetStockObject(WHITE_PEN));
// ... рисуем с пером WHITE_PEN
// возврат в контекст устройства предыдущего пера
SelectObject(hdc, hOldPen);
```

Стандартное перо DC, возвращаемое вызовом `GetStockObject(DC_PEN)`, является представителем «нового поколения» объектов GDI. Обычные объекты GDI «намерть» фиксируются при создании. Их можно использовать, можно удалять, но их нельзя изменять. Если вам понадобился слегка отличающийся объект GDI, придется создавать новый объект и удалять старый. Это может приводить к снижению быстродействия в определенных ситуациях.

Концептуальная новизна пера DC заключается в том, что после его выбора в контекст устройства вы можете *изменять его цвет*. Для этого предусмотрена функция `SetDCPenColor`, имеющая следующий прототип:

```
COLORREF SetDCPenColor(HDC hdc, COLORREF crColor);
```

Параметр `crColor` задает новый цвет пера DC. Функция возвращает предшествующий цвет пера DC.

Следующий фрагмент кода показывает, как можно нарисовать градиентную заливку всего одним пером:

```
{
    HGDIOBJ hOld = SelectObject(hdc, GetStockObject(DC_PEN));
    for (int i = 0; i < 256; ++i) {
        SetDCPenColor(hdc, RGB(255-i, 128, i));
        MoveToEx(hdc, 10, i+10, NULL);
        LineTo(hdc, 266, i+10);
    }
    SelectObject(hdc, hOld);
}
```

Вставьте этот фрагмент в код программы Hello2 (глава 1) после вывода текста функцией `DrawText`, чтобы посмотреть на его работу¹.

Компиляция модифицированной программы, скорее всего², будет неудачной. Компилятор сообщит, что идентификатор DC_PEN ему неизвестен. Чтобы исправить эту ошибку, необходимо добавить в самом начале файла (еще до директивы `#include <windows.h>`) следующую строку:

```
#define _WIN32_WINNT 0x500
```

После этого компиляция должна пройти успешно.

Простые перья

Все стандартные перья имеют сплошной цвет и единичную толщину. Чтобы рисовать прерывистые или более толстые линии, нужно использовать нестандартные

¹ Наружные фигурные скобки в данном фрагменте делают возможным объявление локальных переменных внутри блока оператора switch.

² При использовании компилятора Visual C++ 6.0.

объекты логического пера. Простые перья создаются вызовом функции `CreatePen` или `CreatePenIndirect`.

Функция `CreatePen` имеет следующий прототип:

```
HPEN CreatePen(int fnPenStyle, int nWidth, COLORREF crColor);
```

Первый параметр задает стиль пера, определяющий порядок следования пикселов и расположение линии. Возможные значения этого параметра приведены в табл. 2.5.

Таблица 2.5. Стили простых перьев

Стиль	Вид линии	Выравнивание	Ограничения на толщину
PS_SOLID	Сплошная	По центру	Нет
PS_DASH	Пунктирная	По центру	<code>nWidth <= 1</code>
PS_DOT	Точечная	По центру	<code>nWidth <= 1</code>
PS_DASHDOT	Пунктирно-точечная	По центру	<code>nWidth <= 1</code>
PS_DASHDOTDOT	Отрезок и две точки	По центру	<code>nWidth <= 1</code>
PS_NULL	Не рисуется		
PS_INSIDEFRAME	Сплошная	Внутри контура	<code>nWidth > 1</code>

На рис. 2.8 показано, как выглядят линии этих стилей. Все линии имеют толщину 1 пикселя.

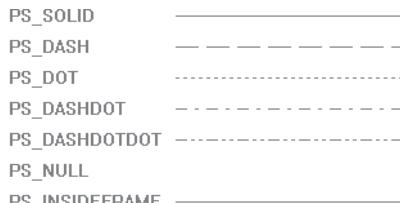


Рис. 2.8. Стили простых перьев

Второй параметр функции `CreatePen` задает толщину линии в *логических единицах*. Напомним, что если в контексте устройства используется режим отображения по умолчанию `MM_TEXT`, то логические единицы совпадают с физическими единицами и выражаются в пикселях.

Если логические единицы не совпадают с физическими, то физическая толщина пера устанавливается в соответствии с масштабом, который определяется режимом отображения и мировыми преобразованиями, если они есть. При этом толщина пера будет всегда одинаковой только в том случае, когда масштабы по обеим осям одинаковы. Если же масштабы различаются, то вертикальные и горизонтальные линии будут иметь разную толщину. То же касается и наклонных линий, причем их толщина будет зависеть от угла наклона.

Параметру `nWidth` можно передать нулевое значение. В этом случае будет использоваться перо толщиной 1 пиксель независимо от режима отображения.

Если задать точечный или пунктирный стиль (`PS_DASH`, `PS_DOT`, `PS_DASHDOT`, `PS_DASHDOTDOT`) с физической толщиной более единицы, то Windows будет использовать перо со стилем `PS_SOLID`. Таким образом, стилевые (прерывистые) линии можно рисовать только с толщиной 1 пикселя. Обратите внимание на то, что точки в стилевых линиях изображаются тремя пикселями.

Третьему параметру функции `CreatePen` передается цвет пера в виде значения типа `COLORREF`. Обычно это значение задается либо с помощью макроса `RGB`, либо с помощью макроса `PALETTERGB`¹. Первый вариант используется, если устройство вывода поддерживает полный диапазон цветов, определяемый 24-битным RGB-значением, и следовательно, приложению нет необходимости работать с палитрой. Второй вариант (макрос `PALETTERGB`) необходимо использовать, если приложение работает с логической палитрой, например, для моделей дисплеев, которые поддерживают только 256 цветов. В последнем случае система Windows преобразует запрошенный RGB-цвет в наиболее подходящий индекс палитры.

Характеристика пера, названная в таблице «выравниванием», проявляется наиболее существенно, когда линия используется для обводки замкнутых фигур или для рисования дуг, являющихся частью эллипса. Для всех стилей, кроме `PS_INSIDEFRAME`, линия центрируется таким образом, что часть линии оказывается за пределами ограничивающего контура. Для стиля `PS_INSIDEFRAME` вся линия рисуется внутри ограничивающего контура. Эту разницу можно заметить, только если параметр `nWidth` имеет значение большее, чем единица. Например, рисование дуг на рис. 2.5 и 2.6 выполнено пером со стилем `PS_INSIDEFRAME`. Если же параметр `nWidth` имеет единичное значение, то стиль `PS_INSIDEFRAME` совпадает со стилем `PS_SOLID`.

Приведем пример использования функции `CreatePen`. Следующий вызов создает простое сплошное (однородное) перо красного цвета толщиной 5 логических единиц:

```
hPen = CreatePen(PS_SOLID, 5, RGB(255, 0, 0));
```

Следует отметить, что утолщенные линии рисуются всегда с закругленными окончаниями. Это можно заметить для линий, у которых параметр `nWidth` больше или равен трем единицам.

Второй способ создания простых перьев связан с вызовом функции `CreatePenIndirect`:

```
HPEN CreatePenIndirect(CONST LOGPEN* lplgpn);
```

Этой функции в качестве параметра передается адрес структуры типа `LOGPEN`:

```
typedef struct tagLOGPEN {  
    UINT     style; // стиль пера  
    POINT    width; // толщина в логических единицах  
    COLORREF color; // цвет  
} LOGPEN;
```

Член структуры `width` имеет тип `POINT`, но Windows использует только величину `width.x` как толщину пера и игнорирует значение `width.y`.

Таким образом, сначала определяется переменная типа `LOGPEN`, например:

```
LOGPEN logpen;
```

Затем полям этой переменной присваиваются нужные значения, например:

```
logpen.style = PS_SOLID;  
logpen.width.x = 10;  
logpen.color = RGB(255, 0, 0);
```

И только после этого вызывается функция `CreatePenIndirect`:

```
hPen = CreatePenIndirect(&logpen);
```

¹ См. раздел «Управление цветом. Вывод пикселя».

Простые перья, созданные с помощью функции `CreatePenIndirect`, обладают точно такими же характеристиками, что и перья, созданные функцией `CreatePen`.

Обратите внимание на то, что функции `CreatePen` и `CreatePenIndirect` не требуют дескриптора контекста устройства. Они создают логические перья, которые никак не связаны с контекстом устройства, пока не будет вызвана функция `SelectObject`.

Покажем возможный сценарий использования нескольких перьев в программе. Допустим, приложению требуются три нестандартных пера: красное толщиной 2, зеленое толщиной 5 и синее пунктирное. Сначала нужно определить переменные для хранения дескрипторов этих перьев:

```
static HPEN hPen1, hPen2, hPen3;
```

Сами перья могут быть созданы в процессе обработки сообщения `WM_CREATE`:

```
hPen1 = CreatePen(PS_SOLID, 2, RGB(255, 0, 0));
hPen2 = CreatePen(PS_SOLID, 5, RGB(0, 255, 0));
hPen3 = CreatePen(PS_DASH, 1, RGB(0, 0, 255));
```

В процессе обработки сообщения `WM_PAINT` можно выбрать одно из этих перьев в контекст устройства и рисовать с его помощью:

```
SelectObject(hdc, hPen1);
// ... функции рисования линий
SelectObject(hdc, hPen2);
// ... функции рисования линий
```

В процессе обработки сообщения `WM_DESTROY` рекомендуется удалить эти перья:

```
DeleteObject(hPen1);
DeleteObject(hPen2);
DeleteObject(hPen3);
```

Это наиболее общий подход. Но возможны и другие сценарии. Например, можно создать перо в блоке обработки сообщения `WM_PAINT` и удалить его перед вызовом функции `EndPaint` или даже после ее вызова.

Кроме того, можно создавать перья «на лету», объединяя вызовы функций `CreatePen` и `SelectObject` в одну инструкцию:

```
SelectObject(hdc, CreatePen(PS_DOT, 0, RGB(255, 255, 0)));
```

Теперь последующее рисование линий будет выполняться желтым пером с точечным стилем. Закончив рисовать, нужно удалить выбранное перо. Но как это сделать, если его дескриптор не сохранен? В этом случае на выручку приходит функция `SelectObject` с ее свойством возвращать дескриптор объекта, выбранного в контекст устройства ранее. Таким образом, можно удалить перо при помощи выбора стандартного пера `BLACK_PEN` в контекст устройства и удаления значения, возвращенного функцией `SelectObject`:

```
DeleteObject>SelectObject(hdc, GetStockObject(BLACK_PEN));
```

Рассмотрим слегка модифицированную версию последнего примера. Дело в том, что можно сохранить дескриптор пера при первом вызове `SelectObject`:

```
hPen = SelectObject(hdc, CreatePen(PS_DOT, 0, RGB(255, 255, 0)));
```

Если это первый вызов `SelectObject` после получения дескриптора контекста устройства, то `hPen` примет значение дескриптора стандартного пера `BLACK_PEN`. Тогда после окончания рисования выбранным пером его удаление можно совместить с возвратом в контекст устройства стандартного пера:

```
DeleteObject>SelectObject(hdc, hPen);
```

Расширенные перья

Простые перья, рассмотренные в предыдущем разделе, обладают ограниченными возможностями. Так, стилевые линии можно рисовать только толщиной 1 пикселя, а сплошные линии рисуются любой толщиной, но всегда имеют закругленные окончания.

Для преодоления этих ограничений в Win32 API появилась новая функция ExtCreatePen, создающая перья с расширенным набором атрибутов:

```
HPEN ExtCreatePen(
    DWORD dwPenStyle,           // тип, стиль и атрибуты пера
    DWORD dwWidth,              // толщина
    CONST LOGBRUSH* lpLb,       // атрибуты кисти
    DWORD dwStyleCount,         // длина массива lpStyle
    CONST DWORD* lpStyle        // массив, задающий правила чередования пикселов
);
```

Параметр *dwPenStyle* может принимать значения в виде комбинации флагов, определяющих *тип пера*, его *стиль*, *тип завершения* линий и *тип соединения* линий. Возможные значения этих флагов приведены в табл. 2.6–2.9.

Таблица 2.6. Типы расширенных перьев

Флаг типа	Описание
PS_COSMETIC	Косметическое перо. Толщина равна 1 пиксел
PS_GEOMETRIC	Геометрическое перо. Толщина задается в логических единицах

Из табл. 2.6 видно, что расширенные перья могут быть косметическими и геометрическими.

Таблица 2.7. Стили расширенных перьев

Флаг стиля	Описание	Ограничения
PS_ALTERNATE	Чередование «пиксель — промежуток»	Поддерживается только в Windows NT/2000 и только для косметических перьев
PS_SOLID	Непрерывная линия	
PS_DASH	Пунктирная линия	Windows 95: не поддерживается для геометрических перьев. Windows 98: не поддерживается
PS_DOT	Точечная линия	Windows 95/98: не поддерживается для геометрических перьев
PS_DASHDOT	Пунктирно-точечная линия	Windows 95: не поддерживается для геометрических перьев. Windows 98: не поддерживается
PS_DASHDOTDOT	Отрезок и две точки	Windows 95: не поддерживается для геометрических перьев. Windows 98: не поддерживается
PS_NULL	Линия не рисуется	
PS_INSIDEFRAME	Непрерывная линия. Выравнивание внутри контура	Только для геометрических перьев
PS_USERSTYLE	Чередование отрезков и промежутков задается параметрами dwStyleCount и lpStyle	Поддерживается только в Windows NT/2000

Все стили, кроме PS_INSIDEFRAME, имеют выравнивание по центру. Интерпретация этого параметра была приведена при описании простых перьев.

Таблица 2.8. Тип завершения, используемый только для геометрических перьев

Флаг завершения	Описание
PS_ENDCAP_ROUND	Закругленное завершение, когда к линии добавляется половина круга
PS_ENDCAP_SQUARE	Квадратное завершение, когда к линии добавляется половина квадрата
PS_ENDCAP_FLAT	Плоское завершение

Таблица 2.9. Тип соединения, используемый только для геометрических перьев

Флаг соединения	Описание
PS_JOIN_BEVEL	Усеченное соединение
PS_JOIN_MITER	Заостренное соединение
PS_JOIN_ROUND	Закругленное соединение

Косметические перья

Косметические перья можно использовать только с толщиной 1 пиксел. При попытке задать большую толщину Windows использует стиль PS_SOLID.

Для стилей PS_DASH, PS_DOT, PS_DASHDOT и PS_DASHDOTDOT косметические перья рисуют точно такие же линии, как и простые перья. Однако, в отличие от простых перьев, они всегда рисуют в прозрачном режиме смешивания фона, даже если в контексте устройства установлен режим OPAQUE.

Для стиля PS_ALTERNATE косметическое перо рисует «настоящую» точечную линию, которую образуют точки размером 1 пиксел и промежутки между точками также размером 1 пиксел.

Для использования стиля PS_USERSTYLE необходимы два дополнительных параметра: dwStyleCount задает количество элементов в массиве с адресом lpStyle. Первый элемент массива содержит длину первого отрезка, второй — длину промежутка, третий — длину второго отрезка и т. д. При этом одна единица длины соответствует трем пикселям вместо одного.

Геометрические перья

Геометрические перья рисуют линии, которые могут различаться толщиной, стилем, заливкой, типом завершения и типом соединения.

Толщина геометрического пера задается в логических координатах. Если в простых перьях единица длины, соответствующая изображению точки, равна 3 пикселя, то в геометрических перьях единица длины в режиме отображения MM_TEXT равна 1 пиксел. Но с увеличением толщины линии пропорционально увеличивается и длина точки, равно как и длина других отрезков, составляющих линию. Также эти размеры могут изменяться в соответствии с мировыми преобразованиями или режимом отображения.

В отличие от простых перьев, утолщенные геометрические перья рисуют линии в соответствии со своим стилем, как показано на рис. 2.9.

Из рисунка также видно, что геометрические перья по умолчанию имеют завершение, определяемое стилем PS_ENDCAP_ROUND. Но тип завершения можно изменить, если при вызове функции ExtCreatePen передать первый параметр с добавлением флага

из табл. 2.8. На рис. 2.10 показано, как выглядит буква Z, состоящая из трех утолщенных линий с разными завершениями. Линии нарисованы при помощи функции LineTo. Для большей четкости картины в местах стыковки вторая линия нарисована более светлым пером, чем первая и третья линии. Тонкие белые линии, нанесенные поверх фигуры, обозначают положение базовых осей каждой линии.

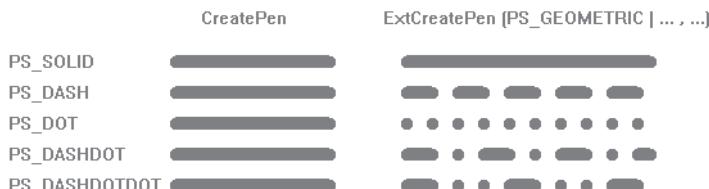


Рис. 2.9. Простые и расширенные геометрические перья толщиной 10 пикселов



Рис. 2.10. Стыковка линий с разными завершениями

Учтите, что если вы будете рисовать при помощи функции LineTo все три линии пером одного и того же цвета в режиме рисования R2_XORPEN, то на стыках из-за повторной прорисовки образуются «провалы», как показано на рис. 2.11, слева. Та же Z-образная фигура, нарисованная функцией Polyline (на рис. 2.11 в центре и справа), таких провалов не имеет. При этом фигура справа демонстрирует возможность заливки линий с помощью кисти. В данном случае перед созданием геометрического пера поле lbStyle структуры LOGBRUSH было установлено в значение BS_HATCHED, а поле lbHatch — в значение HS_DIAGCROSS¹.



Рис. 2.11. Рисование геометрическим пером в режиме рисования R2_XORPEN

Функция Polyline относится к группе функций GDI, позволяющих выполнить рисование нескольких линий и кривых за один вызов. Функции этой группы обладают тем замечательным свойством, что обеспечивают плавную стыковку линий. Именно для них имеет значение тип соединения геометрического пера, устанавливаемый флагом из табл. 2.9. По умолчанию в контексте устройства используется флаг PS_JOIN_ROUND. На рис. 2.12 показаны различные типы соединений в сочетании с разными типами завершения геометрического пера. Рисование производилось при помощи функции Polyline.

¹ Работа с объектом логической кисти рассматривается далее в разделе «Кисти».

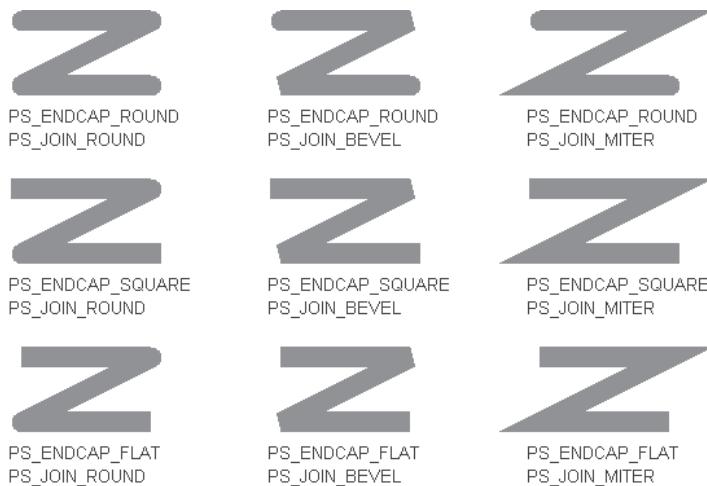


Рис. 2.12. Рисование с помощью Polyline для разных типов соединений и разных типов завершения геометрического пера

Перейдем теперь к рассмотрению рисования замкнутых фигур.

Любая замкнутая фигура, например прямоугольник или эллипс, рисуется в Windows с использованием двух графических объектов, выбранных в контекст устройства. Первым объектом является текущее перо, которым обводится контур фигуры, а вторым объектом — текущая кисть, используемая для заливки (закрашивания) внутренней области фигуры. Контекст устройства содержит кисть по умолчанию, которой является сплошная кисть белого цвета. Обычно фон окна тоже имеет белый цвет, поэтому, рисуя замкнутые фигуры с кистью по умолчанию, нельзя увидеть эффект заливки. В связи с этим стоит сначала научиться создавать и использовать различные кисти, а уже потом знакомиться с функциями, рисующими замкнутые фигуры.

Кисти

Кисть — это растр размером 8×8 пикселов, который при закрашивании области дублируется в горизонтальном и вертикальном направлении. Когда Windows использует смещивание (*dithering*) для отображения большего числа цветов, чем доступно на дисплее, то на самом деле для этого используется кисть. На монохромном дисплее, например, смещивание черных и белых пикселов позволяет получить 64 разных оттенка серого цвета. На цветных видеосистемах полутона тоже реализуются при помощи подобных растровых образов, но с гораздо более широким набором доступных цветов.

Win32 GDI предоставляет несколько функций для создания объектов логических кистей. Логическая кисть описывает требования, предъявляемые к заливке со стороны приложения. Эти требования не всегда совпадают с возможностями физических устройств. Драйверы устройств поддерживают собственные структуры данных, определяющие реализацию логической кисти. Такие внутренние объекты называются физическими кистями.

Для дескрипторов логических кистей зарезервирован тип **HBRUSH** (*handle to a brush*), поэтому новая кисть объявляется следующим образом:

```
HBRUSH hBrush;
```

Значение дескриптора получают вызовом соответствующей функции. Вид вызываемой функции зависит от типа кисти. Так же как и при работе с перьями, созданные кисти выбираются в контекст устройства с помощью функции **SelectObject**, после чего заливка всех замкнутых фигур осуществляется выбранной кистью. Когда кисть перестает быть нужной, рекомендуется вернуть в контекст устройства прежнюю кисть, а ненужную кисть удалить при помощи функции **DeleteObject**. В разделе «Простые перья» были приведены возможные сценарии использования нескольких перьев в одном приложении. Аналогичные сценарии можно использовать и для работы приложения с несколькими кистями.

Стандартные кисти

Стандартные кисти, предоставляемые GDI, приведены в табл. 2.10.

Таблица 2.10. Стандартные кисти

Индекс кисти	Описание
BLACK_BRUSH	Черная кисть
DKGRAY_BRUSH	Темно-серая кисть
DC_BRUSH	Кисть DC — сплошная цветная кисть; по умолчанию имеет белый цвет; цвет может быть изменен функцией SetDCBrushColor
GRAY_BRUSH	Серая кисть
HOLLOW_BRUSH	Пустая кисть (заливки нет)
LTGRAY_BRUSH	Светло-серая кисть
NULL_BRUSH	То же, что и HOLLOW_BRUSH
WHITE_BRUSH	Белая кисть, которая используется по умолчанию

Все кисти, приведенные в таблице, — сплошные, то есть все 8×8 пикселов раstra имеют один и тот же цвет. Кисть DC, реализованная только в Windows 2000, относится к числу новых средств GDI. Так же как и перо DC, кисть DC позволяет изменять свой цвет после ее выбора в контекст устройства. Для этого предназначена функция **SetDCBrushColor**.

Чтобы получить дескриптор стандартной кисти, достаточно вызвать функцию **GetStockObject** с одной из констант, приведенных в табл. 2.10, например:

```
hBrush = (HBRUSH) GetStockObject(GRAY_BRUSH);
```

Пользовательские кисти

Стандартные кисти хороши простотой своего использования, но их возможности явно недостаточны для нужд многих приложений. Поэтому Windows содержит функции, создающие пользовательские кисти следующих типов: *сплошные кисти*, *штриховые кисти* и *растровые кисти*.

Сплошные кисти

Сплошная кисть создается вызовом функции **CreateSolidBrush**:

```
HBRUSH CreateSolidBrush(COLORREF crColor);
```

Ее единственному параметру `crColor` передается цвет кисти в виде значения типа `COLORREF`. Обычно это значение задается при помощи макроса `RGB` или `PALETERGB`¹. Первый вариант используется, если устройство вывода поддерживает полный диапазон цветов, определяемый 24-битным RGB-значением. В этом случае приложению нет необходимости работать с палитрой. Второй вариант необходимо использовать, если приложение работает с логической палитрой. В последнем случае Windows преобразует запрошенный в макросе `PALETERGB` цвет в наиболее подходящий индекс палитры. Если подходящий индекс не найден, то устройство имитирует нужный цвет, комбинируя доступные цвета при помощи смешения (*dithering*).

Примеры использования сплошной кисти можно найти в листингах глав 3 и 6.

Штриховые кисти

Штриховая кисть создается вызовом функции `CreateHatchBrush`:

```
HBRUSH CreateHatchBrush(int fnStyle, COLORREF clrref);
```

Параметр `fnStyle`, задающий стиль штриховки, может принимать значения `HS_HORIZONTAL`, `HS_VERTICAL`, `HS_BDIAGONAL`, `HS_FDIAGONAL`, `HS_CROSS` и `HS_DIAGCROSS`.

На рис. 2.13 показано, как выглядят эти стили при заливке прямоугольников размером 64×64 пиксела (верхний ряд). В нижней части рисунка приведены в увеличенном виде минимальные блоки, соответствующие этим штриховым кистям. Каждый минимальный блок представляет собой растр размером 8×8 пикселов.

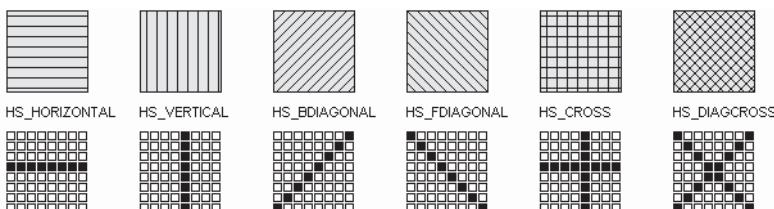


Рис. 2.13. Стили штриховых кистей

В штриховых кистях пиксели делятся на *основные* и *фоновые*. На рис. 2.13 основные пиксели показаны более темным цветом. Цвет основных пикселов задается параметром `clrref`, а цвет фоновых пикселов определяется атрибутом цвета фона графических элементов в контексте устройства, который можно устанавливать при помощи функции `SetBkColor`. Фоновые пиксели выводятся только в том случае, если установлен режим смешивания фона `OPAQUE`.

GDI позволяет также устанавливать *базовую точку* штриховой кисти с помощью функции `SetBrushOrgEx`:

```
BOOL SetBrushOrgEx(HDC hdc, int nXOrg, int nYOrg, LPPOINT lppr);
```

Базовая точка (`nXOrg`, `nYOrg`), задаваемая в системе координат устройства, определяет привязку левого верхнего пикселя штрихового узора. Остальные блоки пикселов выстраиваются соответствующим образом. Параметр `lppr` содержит адрес структуры типа `POINT`, в которой запоминается прежняя базовая точка. По умолчанию координаты базовой точки кисти равны (0, 0).

¹ См. раздел «Управление цветом. Вывод пикселя».

Предположим, что вы рисуете прямоугольник, имея перо толщиной 1 пиксель, и используете штриховую кисть для заливки. Если вы хотите, чтобы левый верхний угол первого пиксельного блока штриховой кисти в точности совпадал с левым верхним углом заливаемой области, то этого можно достичь вызовом функции

```
SetBrushOrgEx(hDC, xLeft + 1, yTop + 1, NULL);
```

где `xLeft` и `yTop` – координаты левого верхнего угла прямоугольника. Обратите внимание на то, что смещение на единицу обусловлено единичной толщиной используемого пера.

Растровые кисти

Растровая (или *узорная*) кисть создается при помощи функции `CreatePatternBrush`:

```
HBRUSH CreatePatternBrush(HBITMAP hBmp);
```

Параметр `hBmp` содержит дескриптор растрового объекта GDI.

Поскольку растровый объект GDI (*bitmap*), называемый также аппаратно-зависимым растром (DDB)¹, может содержать произвольное изображение, то программист получает неограниченные возможности создания рисунков для заливки замкнутых областей.

Давайте проведем следующий эксперимент с растровой кистью. При помощи графического редактора MS Paint создайте рисунок размером 32 × 32 пикселя, содержащий изображение пятиконечной звезды, как показано на рис. 2.14. Это изображение следует сохранить в файле `Star.bmp`.

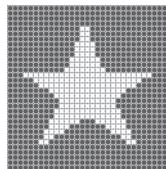


Рис. 2.14. Изображение звезды, созданное в редакторе MS Paint

Затем вставьте в код программы `Hello2`, как мы это делали раньше, следующий фрагмент:

```
{  
    HBITMAP hBmp = (HBITMAP)LoadImage(NULL, "Star.bmp", IMAGE_BITMAP, 0,  
    0, LR_LOADFROMFILE);  
    HBRUSH hBrush = CreatePatternBrush(hBmp);  
    HBRUSH hOldBrush = (HBRUSH)SelectObject(hDC, hBrush);  
    Rectangle(hDC, 32, 32, 288, 160);  
    DeleteObject(SelectObject(hDC, hOldBrush));  
}
```

Функция `LoadImage` в этом фрагменте загружает аппаратно-независимый растр (DIB)² из файла `Star.bmp`, затем преобразует его к формату растрового объекта GDI и возвращает дескриптор этого объекта `hBmp`. Полученный дескриптор передается функции `CreatePatternBrush`, создающей растровую кисть. Кисть `hBrush`

¹ Вопросы использования растров рассмотриваются в главе 3.

² Аппаратно-зависимые и аппаратно-независимые раstry рассматриваются в главе 3.

выбирается в контекст устройства и после этого используется для заливки внутренней области прямоугольника, рисуемого функцией `Rectangle`¹.

Чтобы программа смогла загрузить файл `Star.bmp`, нужно поместить его в папку с проектом `Hello2`. После компиляции и запуска на выполнение программа должна нарисовать прямоугольник, показанный на рис. 2.15.



Рис. 2.15. Заливка прямоугольника растровой кистью

В системах Windows 95/98 накладывается ограничение на применение растровых кистей. Битовые образы кистей не могут превышать размер 8×8 пикселов. В случае превышения размера используется квадрат 8×8 из левого верхнего угла изображения.

Пример использования растровой кисти приведен в главе 11.

Структура `LOGBRUSH` и функция `CreateBrushIndirect`

Win32 GDI предоставляет еще одну функцию — `CreateBrushIndirect`, позволяющую создать логическую кисть любого из трех рассмотренных выше типов:

```
HBRUSH CreateBrushIndirect(CONST LOGBRUSH* lpb);
```

Параметр `lpb` — это адрес структуры типа `LOGBRUSH`, в которой задаются параметры кисти:

```
typedef struct tagLOGBRUSH {
    UINT    lbStyle;
    COLORREF lbColor;
    LONG    lbHatch;
} LOGBRUSH;
```

Поле `lbStyle` задает стиль кисти и может иметь одно из значений, приведенных в табл. 2.11.

Таблица 2.11. Возможные значения поля `lbStyle`

Значение	Описание
<code>BS_SOLID</code>	Сплошная кисть
<code>BS_HATCHED</code>	Штриховая кисть
<code>BS_HOLLOW</code>	Пустая кисть (заливки нет)
<code>BS_NULL</code>	То же, что и <code>BS_HOLLOW</code>
<code>BS_PATTERN</code>	Растровая кисть, рисунок которой определен растровым изображением в памяти
<code>BS_PATTERN8X8</code>	То же, что и <code>BS_PATTERN</code>
<code>BS_DIBPATTERN</code>	Растровая кисть, рисунок которой задается аппаратно-независимым растровым изображением (DIB — device-independent bitmap), дескриптор которого содержится в поле <code>lbHatch</code>

¹ Функции, рисующие замкнутые фигуры, рассматриваются в следующем разделе.

Значение	Описание
BS_DIBPATTERN8X8	То же, что и BS_DIBPATTERN
BS_DIBPATTERNPNT	Растровая кисть, рисунок которой задается аппаратно-независимым растровым изображением (DIB). Поле lbHatch содержит указатель на DIB-изображение

Поле lbColor задает цвет кисти. Его интерпретация зависит от значения поля lbStyle и поясняется в табл. 2.12.

Таблица 2.12. Интерпретация поля lbColor

Значения поля lbStyle	Интерпретация поля lbColor
BS_HOLLOW или BS_PATTERN	Игнорируется
BS_HATCHED или BS_SOLID	Значение типа COLORREF
BS_DIBPATTERN или BS_DIBPATTERNPNT	Младшая часть слова lbColor должна иметь значение DIB_PAL_COLORS или DIB_RGB_COLORS. Первое значение указывает на то, что в массиве bmiColors, являющемся членом структуры BITMAPINFO, содержатся 16-разрядные индексы цветов в логической палитре, второе значение — на то, что в массиве bmiColors содержатся явные RGB-коды цветов

Поле lbHatch задает стиль штриховки. Его интерпретация также зависит от значения поля lbStyle и поясняется в табл. 2.13.

Таблица 2.13. Интерпретация поля lbHatch

Значения поля lbStyle	Интерпретация поля lbHatch
BS_SOLID или BS_HOLLOW	Игнорируется
BS_PATTERN	Дескриптор растрового объекта GDI (bitmap'a)
BS_HATCHED	Одно из значений HS_HORIZONTAL, HS_VERTICAL, HS_BDIAGONAL, HS_FDIAGONAL, HS_CROSS, HS_DIAGCROSS
BS_DIBPATTERN	Дескриптор упакованного DIB-растра
BS_DIBPATTERNPNT	Указатель на упакованный DIB-растр

Структура LOGBRUSH также используется при создании расширенных перьев, которые рассматривались ранее. Например, третья фигура на рис. 2.11 была нарисована с использованием данной структуры.

Замкнутые фигуры

Напомним, что любая замкнутая фигура рисуется в Windows с использованием текущего пера, которым обводится контур фигуры, и текущей кисти, которая используется для заливки внутренней области фигуры. По умолчанию в контексте устройства выбрана сплошная кисть белого цвета.

Прямоугольники

Наиболее важной геометрической фигурой в Windows является прямоугольник. Прямоугольники применяются при определении окон и клиентских

областей, различных фигур с прямоугольным ограничивающим контуром, при отсечении и при форматировании текста. Поэтому в Win32 API определены специальная структура данных для представления прямоугольников и многочисленные функции как для работы с этой структурой, так и для рисования прямоугольников.

Прямоугольник как структура данных

Эта структура данных уже рассматривалась в главе 1, но все же напомним ее определение:

```
typedef struct tagRECT {
    LONG left;
    LONG top;
    LONG right;
    LONG bottom;
} RECT;
```

Поля структуры задают координаты левого верхнего угла (`left, top`) и правого нижнего угла (`right, bottom`) прямоугольника.

Допустим, что в программе определена переменная этого типа:

```
RECT rect;
```

Для установки всех полей структуры `rect` при обычной записи вам пришлось бы написать примерно следующий код:

```
rect.left = x_left;
rect.top = y_top;
rect.right = x_right;
rect.bottom = y_bottom;
```

Но если воспользоваться функцией `SetRect`, имеющей прототип:

```
BOOL SetRect(LPRECT lprc, int xLeft, int yTop, int xRight, int yBottom);
```

то четыре строчки кода можно заменить одной:

```
SetRect(&rect, x_left, y_top, x_right, y_bottom);
```

Кроме этой функции Win32 API предлагает еще восемь функций для работы со структурами типа `RECT`. В табл. 2.14 показано использование этих функций.

Таблица 2.14. Функции для работы со структурами типа `RECT`

Функция	Описание
<code>SetRect(&rect, xLeft, yTop, xRight, yBottom);</code>	Установить координаты прямоугольника в заданные значения
<code>OffsetRect(&rect, dX, dY);</code>	Переместить прямоугольник по оси <i>X</i> на величину <i>dX</i> и по оси <i>Y</i> на величину <i>dY</i>
<code>InflateRect(&rect, dX, dY);</code>	Уменьшить или увеличить ширину и высоту прямоугольника
<code>SetRectEmpty(&rect);</code>	Установить все поля структуры <code>rect</code> в нуль
<code>CopyRect(&DestRect, &SrcRect);</code>	Скопировать один прямоугольник в другой
<code>IntersectRect(&DestRect, &SrcRect1, &SrcRect2);</code>	Получить пересечение двух прямоугольников
<code>UnionRect(&DestRect, &SrcRect1, &SrcRect2);</code>	Получить объединение двух прямоугольников
<code>bEmpty = IsRectEmpty(&rect);</code>	Определить, является ли прямоугольник пустым
<code>bInRect = PtInRect(&rect, point);</code>	Определить, содержится ли точка <i>point</i> внутри прямоугольника <code>rect</code>

Рисование прямоугольников

Для рисования прямоугольников предусмотрены функции `Rectangle`, `FillRect`, `FrameRect`, `InvertRect` и `DrawFocusRect`.

Rectangle

Функция `Rectangle` имеет следующий прототип:

```
BOOL Rectangle(HDC hdc, int xLeft, int yTop, int xRight, int yBottom);
```

Она рисует прямоугольник, определяемый четырьмя координатами. Параметры `xLeft` и `yTop` задают положение левого верхнего угла, а `xRight` и `yBottom` — положение правого нижнего угла. Все координаты определяются в логических единицах. Стороны прямоугольника всегда параллельны горизонтальной и вертикальной сторонам экрана.

Хотя в MSDN утверждается, что необходимо выполнять условия `xLeft < xRight` и `yTop < yBottom`, экспериментальная проверка показывает, что если второй и третий параметры задают правый нижний угол, а четвертый и пятый параметры — левый верхний угол, то Windows справляется с этой «головоломкой» и рисует требуемый прямоугольник.

На процесс рисования влияют многие атрибуты, содержащиеся в контексте устройства:

- ❑ графический режим, который может быть совместимым (`GM_COMPATIBLE`) или расширенным (`GM_ADVANCED`);
- ❑ стиль и цвет пера;
- ❑ толщина пера;
- ❑ режим рисования (текущая бинарная раstralная операция).

Рисунок 2.16 иллюстрирует результаты применения функции `Rectangle` при разных атрибутах контекста устройства.

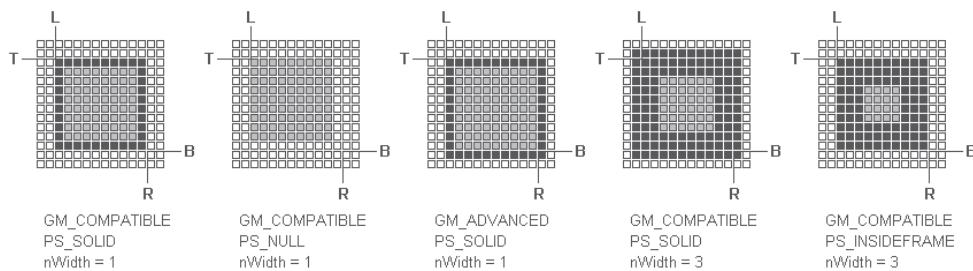


Рис. 2.16. Рисование прямоугольников функцией `Rectangle`

Рисование выполнялось черным пером, за исключением второго прямоугольника, нарисованного пустым пером, и серой стандартной кистью (`GRAY_BRUSH`), выбранной в контексте устройства.

Прежде всего отметим, что в совместимом графическом режиме, который установлен по умолчанию¹, прямоугольник рисуется так, что действительное положение его правого нижнего угла определяется точкой (`xRight × 1, yBottom - 1`).

¹ См. раздел «Системы координат и преобразования».

В этом режиме нарисованы первый, второй, четвертый и пятый прямоугольники. И только в расширенном графическом режиме, в котором был нарисован третий прямоугольник, действительное положение правого нижнего угла определяется точкой `(xRight, yBottom)`.

Когда прямоугольник рисуется пустым пером `PS_NULL`, то его размеры уменьшаются на один пиксел по ширине и на один пиксел по высоте.

Назовем *базовой линией* ту линию, которая рисуется при единичной толщине пера. Если толщина пера `nWidth` больше единицы, то один пиксел рисуется на базовой линии, $(nWidth - 1)/2$ пикселов рисуются снаружи от этой линии и еще столько же пикселов рисуются внутри прямоугольника. Но что происходит, когда величина `nWidth` является четным числом и $(nWidth - 1)$ не делится нацело на два? Остаток в виде половины пикселя трансформируется в утолщение на один пиксел сверху для горизонтальных линий и на один пиксел слева — для вертикальных линий.

Все сказанное выше о толстых линиях справедливо для всех стилей пера, кроме стиля `PS_INSIDEFRAME`. Этот стиль задает выравнивание внутри контура. На крайнем справа прямоугольнике хорошо виден механизм его работы. Один пиксел рисуется на базовой линии, а все остальные — внутри прямоугольника.

Текущая растровая операция в контексте устройства распространяется как на периметр, так и на внутреннюю часть прямоугольника.

FillRect

Функция `FillRect` имеет следующий прототип:

```
BOOL FillRect(HDC hdc, CONST RECT* lprc, HBRUSH hbr);
```

Она закрашивает прямоугольник, определяемый структурой типа `RECT`, адрес которой указан в параметре `lprc`. Для закрашивания используется кисть, дескриптор которой передается через параметр `hbr`.

Действительное положение правого нижнего угла прямоугольника определяется точкой `(lprc.right - 1, lprc.bottom - 1)`, независимо от используемого графического режима.

Следует обратить внимание на три важных свойства этой функции:

- ❑ кисть передается функции непосредственно через третий параметр, поэтому ее не нужно задавать в контексте устройства;
- ❑ перо в этой функции вообще не используется;
- ❑ функция не использует атрибут бинарной растровой операции в контексте устройства.

FrameRect

Функция `FrameRect` имеет следующий прототип:

```
BOOL FrameRect(HDC hdc, CONST RECT* lprc, HBRUSH hbr);
```

Она закрашивает периметр прямоугольника `lprc` кистью `hbr`. Толщина рисуемой рамки равна одной логической единице.

Обратите внимание на то, что рамка рисуется кистью, а не пером. Это позволяет получать интересные эффекты в случае использования растровой кисти. Например, можно создать растровую кисть с «шахматным» узором, когда в растре черные и белые пиксели чередуются, как на шахматной доске. Если передать

дескриптор этой кисти функции `FrameRect`, то периметр прямоугольника будет нарисован «настоящей» точечной линией¹.

InvertRect

Функция `InvertRect` имеет следующий прототип:

```
BOOL InvertRect(HDC hdc, CONST RECT* lprc);
```

Она инвертирует цвет каждого пикселя внутри заданного прямоугольника. Если графическое устройство использует палитру, то инвертируются индексы палитры. В устройствах без палитры черный цвет переходит в белый, белый цвет — в черный, а RGB-значение каждого пикселя инвертируется.

При двукратном вызове функции `InvertRect` с одинаковыми параметрами восстанавливается первоначальное изображение.

DrawFocusRect

Функция `DrawFocusRect` имеет следующий прототип:

```
BOOL DrawFocusRect(HDC hdc, CONST RECT* lprc);
```

По своему действию она напоминает функцию `FrameRect`, рисуя периметр прямоугольника шахматной узорной кистью с применением растровой операции «Исключающее ИЛИ». Повторный вызов функции с теми же параметрами восстанавливает первоначальное изображение.

Функция активно используется теми модулями Win32 API, которые отвечают за управление окнами. Например, в диалоговых окнах функция `DrawFocusRect` рисует точечный контур прямоугольника на кнопке, получающей фокус ввода с клавиатуры. Когда фокус переходит к другой кнопке, прямоугольник стирается повторным вызовом функции `DrawFocusRect`.

Функция `DrawFocusRect` также может использоваться при выводе «эластичных» прямоугольников.

Эллипсы, сегменты, секторы и закругленные прямоугольники

Win32 GDI содержит несколько функций для рисования эллипса, его частей и даже гибрида прямоугольника с эллипсом, который выглядит как прямоугольник с закругленными углами.

Эллипсы

Для рисования эллипсов предназначена функция `Ellipse`, прототип которой приведен ниже:

```
BOOL Ellipse(HDC hdc, int xLeft, int yTop, int xRight, int yBottom);
```

Функция рисует эллипс в ограничивающем прямоугольнике, заданном координатами `xLeft`, `yTop`, `xRight`, `yBottom`.

Все сказанное выше о влиянии атрибутов в контексте устройства на процесс рисования прямоугольников распространяется также и на рисование эллипсов функцией `Ellipse`.

¹ См. замечания по поводу стиля `PS_ALTERNATE` для косметических перьев.

Сегменты

В разделе «Линии и кривые» уже рассматривалось рисование дуги эллипса с помощью функции `Arc`. Сегмент эллипса образуется из дуги, если ее концы соединить отрезком, называемым хордой.

Для рисования сегментов используется функция `Chord`:

```
BOOL Chord(HDC hdc, int xLeft, int yTop, int xRight, int yBottom,
           int xStart, int yStart, int xEnd, int yEnd);
```

Функция принимает такой же набор параметров, что и функция `Arc`. Начальный и конечный углы дуги задаются двумя точками с координатами (`xStart, yStart`) и (`xEnd, yEnd`). Направление рисования дуги, а следовательно, и вид получаемой фигуры определяются так же, как и для функции `Arc`. На рис. 2.17 показано применение функции `Chord`.

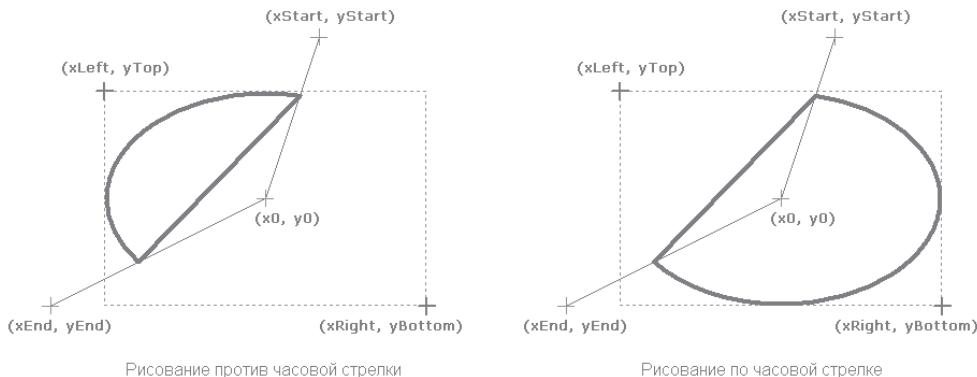


Рис. 2.17. Применение функции `Chord`

Секторы

Сектор эллипса — это фигура, ограниченная дугой и двумя радиусами, проведенными к концам дуги.

Секторы рисуются функцией `Pie`, прототип которой приведен ниже:

```
BOOL Pie(HDC hdc, int xLeft, int yTop, int xRight, int yBottom,
         int xStart, int yStart, int xEnd, int yEnd);
```

Функция принимает такой же набор параметров, что и функция `Arc`. Направление рисования дуги и вид получаемой фигуры определяются так же, как и для функции `Arc`. На рис. 2.18 показано применение функции `Pie`.

Закругленные прямоугольники

Прямоугольник с закругленными углами создается при помощи функции `RoundRect`, имеющей следующий прототип:

```
BOOL RoundRect(HDC hdc, int xLeft, int yTop, int xRight, int yBottom,
                int roundWidth, int roundHeight);
```

Функция позволяет нарисовать прямоугольник, эллипс или любую промежуточную фигуру. Первые пять параметров функции совпадают с параметрами функции `Rectangle`. Последние два параметра определяют ширину и высоту маленьких эллипсов, используемых для закругления углов прямоугольника, как это показано на рис. 2.19.

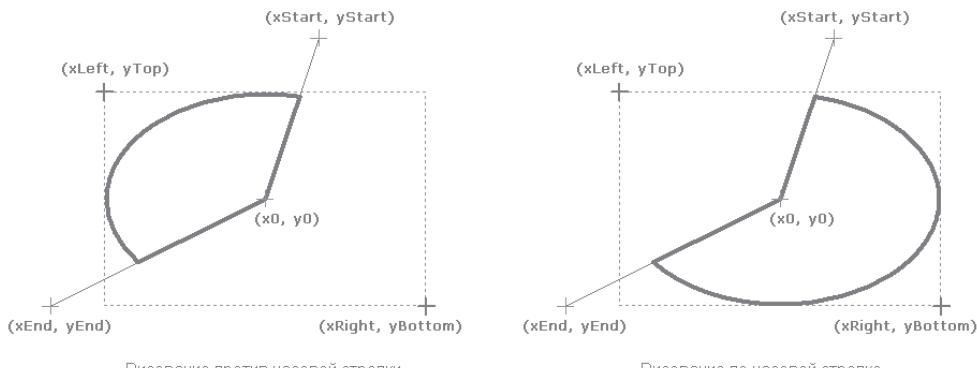


Рис. 2.18. Применение функции Pie

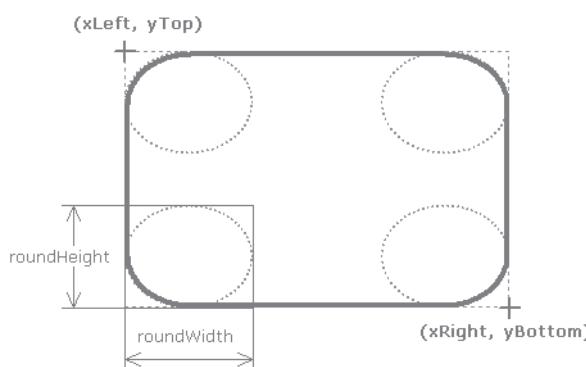


Рис. 2.19. Прямоугольник, нарисованный функцией RoundRect

Если параметры `roundWidth` и `roundHeight` имеют нулевое значение, то функция `RoundRect` рисует прямоугольник. Если параметр `roundWidth` равен `xRight` – `xLeft` и одновременно параметр `roundHeight` равен `yBottom` – `yTop`, то функция `RoundRect` нарисует эллипс.

Многоугольники

Прямоугольник является частным случаем многоугольника. Для рисования произвольного многоугольника предназначена функция `Polygon`:

```
BOOL Polygon(HDC hdc, CONST POINT* lPoints, int nCount);
```

Работа этой функции напоминает рисование ломаных линий функцией `Polyline`. Второй параметр функции принимает адрес массива точек, а третий — количество точек. В отличие от `Polyline`, функция `Polygon` автоматически замыкает фигуру. В случае использования геометрического пера функция `Polygon`, так же как и `Polyline`, оформляет каждую вершину в соответствии с атрибутом соединения.

Для выпуклых многоугольников внутренняя область определяется достаточно четко. Однако невыпуклый многоугольник может состоять из нескольких частей, как, например, пятиконечная звезда, что затрудняет определение его

внутренней области. Такие же проблемы могут возникнуть при перекрытии одного многоугольника другим. В связи с этим Win32 GDI предусматривает два режима заполнения многоугольников: режим **ALTERNATE** и режим **WINDING**. Контекст устройства содержит соответствующий атрибут, который можно изменять, вызывая функцию *SetPolyFillMode*. По умолчанию установлен режим заполнения многоугольников **ALTERNATE**. На рис. 2.20 показано рисование пятиконечной звезды при разных режимах заполнения многоугольников.



Рис. 2.20. Фигуры, нарисованные в двух режимах закрашивания многоугольника: слева — **ALTERNATE**; справа — **WINDING**

В режиме **ALTERNATE** принадлежность некоторой точки внутренней области определяется при помощи довольно простого алгоритма:

1. Проведем мысленно горизонтальную линию сканирования через интересующую нас точку.
2. Пронумеруем точки пересечения этой линии с встречающимися на ее пути линиями многоугольника.
3. Если интересующая нас точка лежит между нечетным и четным пересечениями, то она принадлежит внутренней области многоугольника.

Это правило иллюстрируется левой фигурой, показанной на рис. 2.21.

В режиме **WINDING** принадлежность некоторой точки внутренней области определяется с учетом направления рисования сторон многоугольника по следующему алгоритму:

1. Проведем мысленно горизонтальную линию сканирования через интересующую нас точку.
2. Установим нулевое значение счетчика **count** до первого пересечения этой линии с многоугольником.
3. При каждом следующем пересечении линии многоугольника анализируем направление рисования этой линии по отношению к линии сканирования. Если это направление по часовой стрелке, то значение счетчика **count** увеличивается на единицу, если против часовой стрелки, то уменьшается на единицу.
4. Если текущее значение счетчика **count** больше нуля, то точка принадлежит внутренней области, в противном случае точка считается внешней.

Это правило иллюстрируется правой фигурой, показанной на рис. 2.21.

Для рисования серии многоугольников за один вызов предназначена функция **PolyPolygon**:

```
BOOL PolyPolygon(HDC hdc, CONST POINT* lppoints, CONST int* lpPolyCounts,
int nCount);
```

Второй параметр функции содержит адрес массива точек для всех многоугольников. Многоугольники определяются последовательно один за другим. Каждый

многоугольник замыкается автоматически проведением линии от последней вершины к первой.

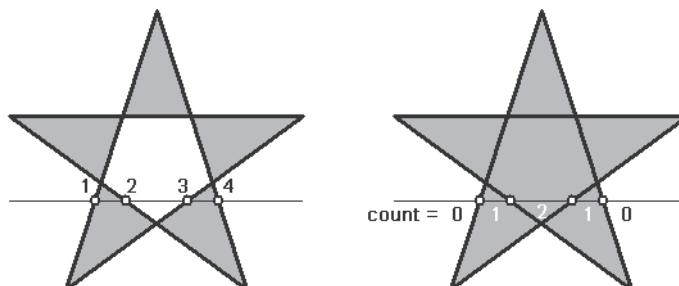


Рис. 2.21. Определение принадлежности точки многоугольнику при разных режимах закрашивания многоугольника: слева — ALTERNATE; справа — WINDING

Третий параметр функции — адрес массива целых чисел, задающих количество вершин в соответствующем многоугольнике. Каждое число должно быть не менее двух. Если количество вершин равно двум, то многоугольник вырождается в прямую линию. Четвертый параметр функции задает количество рисуемых многоугольников.

Регионы и отсечение

Регион — это совокупность точек в координатном пространстве. Эта совокупность может быть пустой, а может занимать все координатное пространство. Она может иметь прямоугольную или любую неправильную форму. Регионы так же, как пе-рья, кисти, битовые образы и многие другие элементы, являются объектами GDI. После создания объекта региона приложение получает его дескриптор, который относится к типу *HRGN*. Когда объект региона становится ненужным, его следует удалить функцией *DeleteObject*. Регионы могут использоваться как для рисования, так и для отсечения.

Создание регионов

Простейший тип региона — это прямоугольник. После объявления объекта региона:

`HRGN hRgn;`

вы можете создать *прямоугольный регион* одним из двух способов:

`hRgn = CreateRectRgn(xLeft, yTop, xRight, yBottom);`

или

`hRgn = CreateRectRgnIndirect(&rect);`

Вы можете также создать *эллиптический регион*, используя следующие вызовы:

`hRgn = CreateEllipticRgn(xLeft, yTop, xRight, yBottom);`

или

`hRgn = CreateEllipticRgnIndirect(&rect);`

Прямоугольный регион с закругленными углами тоже формируется довольно просто:

```
hRgn = CreateRoundRectRgn(xLeft, yTop, xRight, yBottom, roundWidth,
                           roundHeight);
```

Создание многоугольного региона похоже на использование функции **Polygon**:

```
hRgn = CreatePolygonRgn(point, iCount, iPolyFillMode);
```

Аргумент **point** содержит адрес массива структур типа **POINT**. В аргументе **iCount** задается количество точек, а аргумент **iPolyFillMode** принимает значение **ALTERNATE** или **WINDING**.

Также можно создать сложный регион, состоящий из нескольких многоугольников при помощи функции **CreatePolyPolygonRgn**:

```
hRgn = CreatePolyPolygon(point, polyCounts, iCount, iPolyFillMode);
```

Использование этой функции аналогично использованию функции **PolyPolygon**.

Операции с объектами регионов

Регион представляет собой множество точек двумерного пространства, поэтому многие операции над регионами выглядят как операции над множествами. В табл. 2.15 приведены функции, реализующие эти операции.

Таблица 2.15. Функции для работы с регионами

Функция	Назначение
BOOL PtInRegion(HRGN hrgn, int X, int Y);	Проверяет, принадлежит ли точка (X, Y) региону hrgn
BOOL RectInRegion(HRGN hrgn, CONST RECT *lprc);	Проверяет, принадлежит ли хотя бы одна точка прямоугольника lprc региону hrgn
BOOL EqualRgn(HRGN hRgn1, HRGN hRgn2);	Проверяет, содержат ли регионы hRgn1 и hRgn2 одинаковые множества точек
int GetRgnBox(HRGN hrgn, LPRECT lprc);	Возвращает через lprc ограничивающий прямоугольник региона
int CombineRgn(HRGN hrgnDest, HRGN hrgnSrc1, HRGN hrgnSrc2, int fnCombineMode);	Комбинирует два региона, hrgnSrc1 и hrgnSrc2, в соответствии с операцией fnCombineMode. Результат помещается в hrgnDest
int OffsetRgn(HRGN hrgn, int nXOffset, int nYOffset);	Осуществляет смещение региона hrgn на nXOffset единиц по оси X и на nYOffset единиц по оси Y
DWORD GetRegionData(HRGN hRgn, DWORD dwCount, LPRGNDATA lpRgnData);	Заполняет структуру lpRgnData типа RGNDATA данными, описывающими регион hRgn
HRGN ExtCreateRegion(CONST XFORM *lpXform, DWORD nCount, CONST RGNDATA *lpRgnData);	Создает регион по структуре RGNDATA с возможным аффинным преобразованием, заданным структурой XFORM

Укажем на некоторые возможные применения этих функций.

Функция **PtInRegion** может оказаться полезной при реализации некоторых экзотических разновидностей кнопок или интерактивных областей, изменяющих цвет под курсором мыши. Приложение должно лишь создать объект региона, соответствующий интерактивной области, и вызвать функцию **PtInRegion** при обработке сообщения **WM_MOUSEMOVE** для изменения изображения.

Функция `OffsetRgn` может применяться для отсечения при многократном рисовании движущихся объектов. Кроме того, при помощи этой функции можно отслеживать положение движущегося объекта в игре, чтобы фиксировать столкновения с другими объектами, описываемыми своими регионами.

Функция `CombineRgn` позволяет выполнять с объектами регионов некоторые полезные операции, заимствованные из теории множеств. Она комбинирует два исходных региона и строит третий, на который ссылается параметр `hrgnDest`. Все три дескриптора регионов должны быть действительными еще до вызова функции, однако регион, представленный дескриптором `hrgnDest`, заменяется новым, сформированным в результате вызова функции. Параметр `fnCombineMode` определяет операцию, выполняемую с исходными регионами, в соответствии с табл. 2.16.

Таблица 2.16. Операции над регионами, выполняемые функцией `CombineRgn`

Значение <code>fnCombineMode</code>	Новый регион
<code>RGN_AND</code>	Область пересечения двух исходных регионов
<code>RGN_OR</code>	Объединение двух исходных регионов
<code>RGN_XOR</code>	Объединение двух исходных регионов, за исключением области пересечения
<code>RGN_DIFF</code>	Часть региона <code>hrgnSrc1</code> , не входящая в регион <code>hrgnSrc2</code>
<code>RGN_COPY</code>	Регион <code>hrgnSrc1</code>

Стоит отметить, что на месте параметров `hrgnSrc1` и `hrgnDest` может быть один и тот же объект, например `hrgnSrc1`.

Функция `CombineRgn` возвращает целочисленный код сложности генерированного региона или код ошибки. Возможные значения кода возврата приведены в табл. 2.17.

Таблица 2.17. Коды, возвращаемые функцией `CombineRgn`

Значение	Описание
<code>NULREGION</code>	Пустой регион
<code>SIMPLEREGION</code>	Регион определяется одним прямоугольником
<code>COMPLEXREGION</code>	Регион определяется несколькими прямоугольниками
<code>ERROR</code>	Регион не создан в результате ошибки (недопустимые значения параметров или нехватка памяти)

Прорисовка регионов

Для прорисовки регионов, заданных дескриптором `hrgn`, предусмотрены функции, перечисленные в табл. 2.18.

Во всех этих функциях координаты регионов задаются в логической системе координат.

Создание объектов регионов и операции с ними связаны со значительными затратами времени и памяти, особенно при усложнении формы региона. Поэтому, если рисуемую фигуру легко воспроизвести другими средствами GDI (функциями рисования прямоугольников, эллипсов и им подобных), следует отдавать предпочтение этому способу. Функции прорисовки регионов следует использовать для замены более дорогостоящих операций, например функций вывода отдельных пикселов. Также они могут быть полезны, когда приложение рисует на экране два перекрыва-

юющихся круга и нужно закрасить общую область некоторой кистью. В этом случае определение двух дуг, ограничивающих общую область, может оказаться не очень простой задачей. В то же время пересечение двух круговых регионов элементарно определяется функцией `CombineRgn`.

Таблица 2.18. Функции рисования регионов

Функция	Описание
<code>BOOL FillRgn(HDC hdc, HRGN hrgn, HBRUSH hbr);</code>	Закрашивает регион кистью <code>hbr</code>
<code>BOOL PaintRgn(HDC hdc, HRGN hrgn);</code>	Закрашивает регион кистью, выбранной в контекст устройства
<code>BOOL FrameRgn(HDC hdc, HRGN hrgn, HBRUSH hbr, int nWidth, int nHeight);</code>	Обводит контур региона кистью <code>hbr</code> . Ширина вертикальных «мазков» кисти равна <code>nWidth</code> , высота горизонтальных «мазков» кисти равна <code>nHeight</code>
<code>BOOL InvertRgn(HDC hdc, HRGN hrgn);</code>	Инвертирует цвет каждого пикселя внутри региона (работает аналогично функции <code>InvertRect</code>)

Отсечение

Отсечение — это ограничение вывода рисуемого изображения в пределах некоторой заданной области. Напомним, что Windows использует соответствующие понятия, связанные с данной проблемой:

- *Обновляемый регион (недействительный регион)* — та часть окна, которая требует обновления после тех или иных событий. Обновляемый регион формируется как системой, так и приложением в результате вызова функции `InvalidateRect` или `InvalidateRgn`.
- *Видимый регион* — та часть окна, которую в данный момент видит пользователь. В результате перекрытия другим окном видимая часть окна может измениться.
- *Регион отсечения* — область, внутри которой система разрешает рисование. Когда приложение получает контекст устройства, система устанавливает регион отсечения как результат пересечения видимого региона и обновляемого региона. Приложение может усилить ограничения, накладываемые на регион отсечения, при помощи функции `SetWindowRgn` или `SelectClipRgn`.

Обратите внимание на то, что при отсечении координаты всех регионов задаются в *системе координат устройства*.

Отсечение для всего окна

Функция `SetWindowRgn` предназначена для установки *региона окна*. Регион окна определяет область внутри окна, где Windows разрешает рисование. Функция имеет следующий прототип:

```
int SetWindowRgn(
    HWND hWnd,           // дескриптор окна
    HRGN hRgn,          // дескриптор региона
    BOOL bRedraw        // флаг перерисовки окна после установки региона
);
```

Координаты региона определяются относительно левого верхнего угла окна, а не его клиентской области.

Функция может использоваться для изменения формы главного окна приложения.

Отсечение для клиентской области окна

Регион с дескриптором `hRgn`, созданный при помощи рассмотренных выше функций, может быть использован для отсечения при выводе в клиентской области окна. Для этого он должен быть выбран в контекст устройства при помощи функции `SelectClipRgn`:

```
SelectClipRgn(hDC, hRgn);
```

В этом случае GDI создает копию данных региона и связывает ее с контекстом устройства. Поэтому после выполнения функции `SelectClipRgn` объект `hRgn` может быть удален.

Чтобы удалить из контекста устройства выбранный ранее регион отсечения, нужно вызвать функцию `SelectClipRgn` с параметром `NULL`:

```
SelectClipRgn(hDC, NULL);
```

Чаще всего регионы отсечения применяются для того, чтобы ограничить рисование только той областью экрана, где это действительно необходимо. Это особенно актуально для анимационных или игровых приложений, в которых скорость рисования может иметь критическое значение, а также для программных имитаторов аппаратуры, требующих рисования в реальном времени (см. главу 12).

Примеры использования регионов отсечения содержатся в листинге 2.2 и листингах главы 12.

Отображение текста

В главе 1 уже рассматривалась задача вывода строки текста в центре клиентской области окна (см. листинг 1.1). При этом текст выводился функцией `DrawText`. Ка-
жущаяся простота, с которой был достигнут нужный результат, объясняется тем, что в программе использовались значения по умолчанию для тех атрибутов контекста устройства, которые влияют на рисование текста (табл. 2.19).

Таблица 2.19. Атрибуты контекста устройства, влияющие на вывод текста

Атрибут	Значение по умолчанию	Функции для изменения значения атрибута
Шрифт	SYSTEM_FONT	SelectObject
Цвет текста	Черный	SetTextColor
Цвет фона графических элементов	Белый	SetBkColor
Режим смешивания фона	OPAQUE	SetBkMode
Режим отображения	MM_TEXT	SetMapMode

При необходимости можно изменить указанные значения, вызвав соответствующие функции GDI. Например, можно окрасить выводимый текст любым цветом, вызвав предварительно функцию `SetTextColor`. Или, если системный шрифт

SYSTEM_FONT вас не устраивает, вы можете создать свой логический шрифт и выбрать его в контекст устройства для последующего использования. Прежде чем рассматривать эти возможности, остановимся на вопросе кодировки символов, так как этот атрибут также влияет на создаваемый шрифт.

Наборы символов и кодировки

Представление символов в компьютерах базируется на кодовых таблицах, в которых каждому отображаемому на экране символу соответствует некоторый целочисленный код. Кодовую таблицу называют также *набором символов* (*character set*).

Одним из первых стандартов кодовых таблиц был стандарт представления символов ASCII¹. В наборе символов ASCII (рис. 2.22) каждому символу присвоен 7-битный двоичный код, поэтому общее количество символов равно $2^7 = 128$ символов.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0:	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
1:	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
2:	!	"	#	\$	%	&	'	()	*	,	-	.	/		
3:	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4:	@	A	B	C	D	E	F	G	H	I	Ј	Ќ	Љ	М	Њ	Ѡ
5:	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6:	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7:	p	q	r	s	t	u	v	w	x	y	z	{	}	~	□	

Рис. 2.22. Стандарт представления символов ASCII

Для обеспечения компактности кодовая таблица изображена на рис. 2.22 в двухмерном представлении. Номера строк и столбцов даны в шестнадцатеричной нотации. Шестнадцатеричный код символа образуется в результате сцепления номера строки и номера столбца. Например, код буквы N равен 0x4E, что соответствует двоичному коду 01001110 или десятичному коду 78.

Первые 32 символа в таблице (коды 0x00 – 0x1F) соответствуют неотображаемым управляющим кодам. К ним относятся «возврат каретки» (код 0x0D)², «перевод строки» (код 0x0A) и другие неотображаемые символы³. Затем следуют пробел (код 0x20), синтаксические и служебные знаки, знаки математических операций, цифры и буквы английского алфавита в верхнем и нижнем регистрах. Последний символ (код 0x7F) используется также как управляющий код.

Позже появились различные расширения таблицы ASCII с использованием 8-битной кодировки, позволяющей отображать 256 символов. Во всех расширениях первые 128 позиций повторяют стандарт ASCII.

Наиболее распространенной реализацией такой кодировки является *расширенный набор символов IBM*, предложенный производителями IBM PC в начале 80-х годов. В старшей половине кодовой таблицы эта кодировка содержит псевдографические символы, символы греческого алфавита и некоторые математические символы. Набор символов IBM растиражирован в миллионах микросхем

¹ American Standard Code for Information Interchange.

² Этот код вырабатывается, в частности, при нажатии клавиши Enter.

³ Windows выводит неотображаемые управляющие символы в виде маленьких квадратов.

ПЗУ, которые установлены в видеоадаптерах, принтерах и микросхемах BIOS. Для множества программ, работающих в текстовом режиме и написанных не для Microsoft Windows, используется эта кодировка, поскольку в них для вывода информации на экран используются символы псевдографики, имеющие коды от 0xB0 до 0xDF.

Известно несколько вариантов кодирования набора символов IBM, которые называются *кодовыми страницами* (*code pages*). Вариант, используемый в США и большинстве европейских стран, называется CP437 (code page 437). В России получила наибольшее распространение так называемая *альтернативная кодировка ГОСТа*. Она известна также под именем CP866 и отличается от CP437 тем, что некоторые символы во второй половине таблицы заменены на кириллицу, а псевдографические символы остались на своих местах. Кодовая страница CP866 показана на рис. 2.23.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0:	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	
1:	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	
2:	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
3:	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	
4:	@	А	В	С	Д	Е	Ф	Г	Х	І	Ј	К	Л	М	Н	
5:	Р	Q	Р	С	Т	U	V	W	X	Y	Z	[\	^	-	
6:	‘	а	б	с	д	е	ғ	һ	і	ј	ҝ	լ	մ	ո	՞	
7:	ր	զ	Ր	Ց	Տ	ւ	Վ	Խ	Յ	Զ	{		}	~	□	
8:	Ա	Բ	Վ	Շ	Գ	Ե	Ջ	Զ	Ի	Յ	Կ	Լ	Մ	Ն	Ո	
9:	Ր	Ծ	Ւ	Փ	Խ	Ը	Չ	Շ	Ռ	՚	Ե	Յ	Ւ	Յ	Յ	
A:	ա	բ	վ	գ	դ	ե	շ	զ	ն	յ	կ	լ	մ	ն	ո	
B:	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	
C:	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	
D:	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	
E:	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	
F:	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	ֿ	

Рис. 2.23. Кодовая страница CP866

С появлением операционной системы Microsoft Windows возникла необходимость разработки нового расширения таблицы ASCII, так как текстовый режим MS-DOS, а вместе с ним и символы псевдографики стали уже ненужной архаикой. Так появился *набор символов ANSI*¹, который, фактически, стал международным стандартом ISO².

Набор символов ANSI также имеет разные варианты кодировки, реализованные в виде кодовых страниц, учитывающих национальные особенности алфавитов разных стран. Некоторые из этих страниц показаны в табл. 2.20.

На рис. 2.24 показано содержимое кодировки Windows Latin1 (кодовая страница 1252), а на рис. 2.25 — содержимое кодировки Windows Cyrillic (кодовая страница 1251).

Хотя однобайтных кодировок хватает для представления символов большинства мировых языков, китайский, корейский и японский языки содержат слишком

¹ ANSI — American National Standards Institute.

² ISO — International Organization for Standardization.

много символов, для кодировки которых не хватит одного байта. Поэтому для этих языков используются многобайтные кодировки. Следует отметить, что в кодовых страницах 932 (японский язык), 949 (корейский язык) и 950 (китайский язык) используются также разные принципы кодировки.

Таблица 2.20. Варианты кодирования набора символов ANSI

Кодовая страница	Коды 0 .. 127	Коды 128 .. 255
1250 (Windows Latin2)	ASCII	Восточная Европа
1251 (Windows Cyrillic)	ASCII	Кириллица
1252 (Windows Latin1)	ASCII	США и Западная Европа
1253	ASCII	Греция
1254	ASCII	Турция
...

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0:	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
1:	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
2:	!	"	#	\$	%	&	'	()	*	,	-	.	/		
3:	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4:	@	A	B	C	D	E	F	G	H	I	Ј	Ќ	Љ	Њ	Ѡ	Ѽ
5:	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6:	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7:	p	q	r	s	t	u	v	w	x	y	z	{	}	~	□	
8:	€	□	,	ƒ	„	…	†	‡	^	‰	š	<	Œ	□	ž	□
9:	□	‘	”	•	—	—	~	™	š	>	œ	□	ž	Ӵ		
A:	i	¢	¤	¥	₩	₪	₪	₪	₪	₪	₪	₪	₪	₪	₪	₪
B:	°	±	2	3	‘	μ	¶	·	·	·	·	·	·	·	·	·
C:	À	Á	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Ó	Ý	Ù
D:	Ђ	Ѓ	Ќ	Ѝ	Ѝ	Ѝ	Ѝ	Ѝ	Ѝ	Ѝ	Ѝ	Ѝ	Ѝ	Ѝ	Ѝ	Ѝ
E:	à	á	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	ó	ý	ù
F:	đ	ѓ	ќ	ќ	ќ	ќ	ќ	ќ	ќ	ќ	ќ	ќ	ќ	ќ	ќ	ќ

Рис. 2.24. Кодировка Windows Latin 1 (1252)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0:	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
1:	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
2:	!	"	#	\$	%	&	'	()	*	,	-	.	/		
3:	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4:	@	A	B	C	D	E	F	G	H	I	Ј	Ќ	Љ	Њ	Ѡ	Ѽ
5:	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6:	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7:	p	q	r	s	t	u	v	w	x	y	z	{	}	~	□	
8:	Ђ	Ѓ	Ќ	Ѝ	Ѝ	Ѝ	Ѝ	Ѝ	Ѝ	Ѝ	Ѝ	Ѝ	Ѝ	Ѝ	Ѝ	Ѝ
9:	ѝ	‘	”	•	—	—	~	™	љ	>	њ	ќ	ќ	ќ	ќ	ќ
A:	ў	ў	ў	ў	ў	ў	ў	ў	ў	ў	ў	ў	ў	ў	ў	ў
B:	°	±	I	i	ѓ	μ	·	·	·	·	·	·	·	·	·	·
C:	ѧ	ѧ	ѧ	ѧ	ѧ	ѧ	ѧ	ѧ	ѧ	ѧ	ѧ	ѧ	ѧ	ѧ	ѧ	ѧ
D:	Ր	Ծ	Ը	Ծ	Ը	Ծ	Ը	Ծ	Ը	Ծ	Ը	Ծ	Ը	Ծ	Ը	Ծ
E:	ա	բ	գ	դ	ե	շ	չ	զ	յ	ա	բ	գ	դ	ե	շ	չ
F:	ր	տ	ս	փ	չ	շ	չ	շ	չ	շ	չ	շ	չ	շ	չ	շ

Рис. 2.25. Кодировка Windows Cyrillic (1251)

Если вы хотите узнать на уровне Win API, какие кодовые страницы установлены или поддерживаются в вашей системе, можно воспользоваться функцией `EnumSystemCodePages`. На уровне пользователя эту информацию можно получить при помощи приложения *Regional Settings* (Язык и стандарты) панели управления Windows.

Unicode

Работа с разными кодировками, особенно многобайтными, доставляет программисту немало сложностей, например, в случае преобразования символов из одной кодировки в другую. Для решения подобных проблем был предложен *стандарт Unicode*. В Консорциум Unicode, осуществляющий сопровождение данного стандарта, входят Apple, Hewlett-Packard, IBM, Microsoft, Sun и другие компании. В этом стандарте каждый символ кодируется двумя байтами, что обеспечивает возможность выражения до 65 536 символов. Стандарт Unicode позволяет представить большую часть символов практических всех языков мира.

Символы в стандарте Unicode группируются в логические зоны. Например, в первой зоне содержатся символы с кодами от 0x0000 до 0x007F, и младшие байты этих кодов повторяют таблицу ASCII. Самая большая зона содержит 29 902 иероглифов, используемых в Китае, Японии и Корее.

Windows поддерживает работу с символами как в традиционной ANSI-кодировке, так и в кодировке Unicode. Почти все функции Win32 API, получающие в качестве аргумента адрес строки, имеют ANSI- и Unicode-версии.

При соблюдении небольшого количества правил вы можете написать программу так, что она будет компилироваться как в ANSI-версии, так и в Unicode-версии.

Если в начале файла имеется директива `#define UNICODE`, то компилятор создает Unicode-версию, если такой директивы нет — ANSI-версию. Разумеется, в многофайловом проекте эта директива должна одновременно присутствовать или отсутствовать во всех файлах проекта.

Вот эти правила:

- ❑ Для определения символов и строк вместо типа `char` используйте тип `TCHAR`, вместо типа `char*` используйте тип `LPTSTR`, вместо типа `const char*` — тип `LPCTSTR`.
- ❑ Для определения строковых констант используйте макрос `TEXT`, то есть вместо нотации `"abcd"` употребляйте нотацию `TEXT("abcd")`.
- ❑ В случае использования функций из С-библиотеки, работающих со строками, предусмотрите вызов нужной версии функции. Для этого нужно добавить в начале файла, но после директивы `#define UNICODE`, директивы условной компиляции, подобные следующим инструкциям:

```
#ifdef UNICODE  
#define sprintf swprintf  
#define sscanf swscanf  
#define strcpy wcscpy  
#define strcat wcsconcat  
#define strlen wcslen  
#endif
```

В данной книге все программные примеры даны в ANSI-версии, чтобы не усложнять программный код.

Шрифты

Любой шрифт, с которым имеет дело Windows, характеризуется несколькими параметрами:

- *Гарнитура (typeface)* — это совокупность нескольких начертаний шрифта, объединенных стилевыми и другими признаками. Самыми известными гарнитурами являются Arial, Times New Roman, Courier New.
- *Размер шрифта* — это высота прямоугольника, в котором помещаются все символы шрифта.
- *Начертание* — это специфические характеристики шрифта. В Windows доступны нормальное (normal) начертание, курсивное (italic), полужирное (bold), с подчеркиванием символа (underline), с перечеркиванием символа (strike out). Кроме того, эти виды начертаний могут комбинироваться в любом сочетании.
- *Фиксированная или переменная ширина символов*. Шрифты первой группы называют *моноширинными*. В них все символы имеют одинаковую ширину. Шрифты второй группы называют *пропорциональными*. В них символ «i» занимает гораздо меньше места, чем символ «m». Пропорциональные шрифты более удобны для чтения и поэтому чаще используются в текстовых документах.
- *Глиф (glyph)* — это графическая форма отдельного символа при его изображении на бумаге или экране дисплея. Различные гарнитуры шрифтов различаются прежде всего глифами символов.

На рис. 2.26 показано, как выглядит текст, нарисованный с использованием разных гарнитур.

Шрифт Arial: ABCabciw123
 Шрифт Times New Roman: ABCabciw123
 Шрифт Courier New: ABCabciw123

Рис. 2.26. Вид текста при выводе с разными гарнитурами

В приведенном примере шрифты Arial и Times New Roman являются пропорциональными, а шрифт Courier New — моноширинным. Можно также отметить, что глифы шрифтов Times New Roman и Courier New имеют так называемые засечки, а глифы шрифта Arial таких засечек не имеют.

Шрифты, обладающие общими принципами начертания, принадлежат одному из *семейств (family)*. Характеристики семейств шрифтов приведены в табл. 2.21.

Таблица 2.21. Семейства шрифтов

Семейство шрифтов	Флаг (константа Windows)	Код	Описание
Don't care	FF_DONTCARE	0x00	Шрифт с произвольными атрибутами (используется шрифт по умолчанию)
Roman	FF_ROMAN	0x10	Пропорциональные шрифты с засечками (серифами), например Times New Roman
Swiss	FF_SWISS	0x20	Пропорциональные шрифты без засечек, например Arial
Modern	FF_MODERN	0x30	Моноширинные шрифты с засечками или без засечек, например Courier New

Семейство шрифтов	Флаг (константа Windows)	Код	Описание
Script	FF_SCRIPT	0x40	«Рукописные» шрифты, например Monotype Corsiva
Decorative	FF_DECORATIVE	0x50	Декоративные шрифты, например Old English

Типы шрифтов

Windows поддерживает две главные категории шрифтов: «шрифты GDI» и «шрифты устройства» (*device fonts*). Шрифты GDI хранятся в файлах, которые обычно расположены в одном из подкаталогов операционной системы. Шрифты устройства соответствуют конкретному устройству вывода. Например, большинство принтеров имеет набор встроенных шрифтов устройства.

Шрифты GDI подразделяются на три типа:

- растровые шрифты;
- векторные шрифты;
- шрифты типа TrueType.

В шрифтах *растрового типа* символы хранятся в виде растровых картинок — прямоугольных матриц из точек-пикселов. В этих шрифтах каждая точка символа описывается отдельно. Растровые шрифты удобны для вывода текста на экран, особенно при малой высоте букв. Однако для каждого размера шрифта и, возможно, для различных разрешений экрана необходимо хранить свой набор символов. Растровые шрифты плохо поддаются масштабированию, так как при увеличении символов просто дублируются строки или колонки пикселов, что, конечно, приводит к «зазубринам» в очертаниях глифов. Чаще всего растровые шрифты применяются для воспроизведения текстовых элементов интерфейса Windows.

В *векторных шрифтах* глиф описывается последовательностью линейных отрезков, которые затем рисуются с помощью пера. Векторные шрифты легко масштабируются в широких пределах. Однако они имеют более низкую скорость отображения, плохую четкость при маленьких размерах, а при большом увеличении символы выглядят очень бледными, потому что их контуры являются всего лишь тонкими линиями. В настоящее время они применяются в основном только для плоттеров. Векторные шрифты так же, как и растровые шрифты, хранятся в файлах с расширением .fon.

С появлением шрифтов *типа TrueType* в версии Windows 3.1 значительно повысились возможности и гибкость работы с текстами. TrueType — это технология контурных шрифтов, которая была разработана Apple Computer Inc. и Microsoft Corporation. Эта технология поддерживается многими производителями шрифтов. Отдельные символы шрифтов TrueType определяются контурами, состоящими из прямых линий и кривых. Таким образом, Windows может масштабировать символы, изменяя определяющие контуры координаты. При этом контуры глифов остаются плавными и при увеличении размеров символов. Шрифты TrueType могут быть использованы как для вывода на экран, так и для вывода на принтер, делая возможным режим отображения текста WYSIWYG (What-You-See-Is-What-You-Get)¹.

¹ Что видите — то и получаете.

Когда программе необходимо использовать шрифт TrueType определенного размера, Windows формирует растровое представление этого шрифта. Этот процесс называется *растеризацией шрифта*. Алгоритмы растеризации учитывают не только координаты точек соединения прямых и кривых каждого символа, но и некоторые дополнительные данные, содержащиеся в описании шрифта, чтобы скомпенсировать ошибки округления, которые могли бы привести к искажению формы символа. В результате достигается хорошее качество изображения глифов символов независимо от коэффициента масштабирования.

Шрифт TrueType обычно хранится в одном файле с расширением .ttf. В операционной системе Windows впоследствии появилась поддержка шрифтов OpenType. Эти шрифты аналогичны шрифтам PostScript, но они закодированы в формате шрифтов TrueType. Файлы шрифтов OpenType имеют расширение .otf. Технология OpenType также позволяет объединить несколько шрифтов OpenType в один файл. Для таких шрифтов, называемых коллекциями TrueType, используется расширение .ttc.

Установка и удаление шрифтов

Чтобы приложение смогло выводить текст, используя глифы некоего конкретного шрифта, он должен либо быть установлен в *системной таблице шрифтов* (*system font table*), либо присутствовать в используемом графическом устройстве в качестве встроенного шрифта. Имена шрифтов, установленных на графическом устройстве и хранящихся во внутренней системной таблице, можно получить при помощи функции `EnumFontFamilies` или `ChooseFont`.

Приложение может загрузить шрифт вызовом одной из функций: `AddFontResource` или `AddFontResourceEx`. Эти функции загружают шрифт из соответствующего ресурсного файла. Однако такая установка является временной, поскольку после рестарта операционной системы шрифт окажется недоступным. Чтобы установленный шрифт присутствовал в системе постоянно, информация о нем должна быть включена в реестр Windows.

Если установленный шрифт становится ненужным, то он может быть удален из системной таблицы с помощью функции `RemoveFontResource`.

Приложение, изменяющее системную таблицу шрифтов, должно оповестить об этом все окна верхнего уровня рассылкой сообщения `WM_FONTCHANGE`. Приложения, использующие список установленных шрифтов, должны обрабатывать это сообщение и обновлять содержимое списка шрифтов.

Более подробную информацию об установке шрифтов можно найти в книге [6].

Внедрение шрифтов

При передаче текстовых документов на другие компьютеры нередко возникают серьезные проблемы со шрифтами. Установив на своем компьютере нужные шрифты и подготовив документ с их использованием, вы можете столкнуться с ситуацией, когда на другом компьютере с другим набором установленных шрифтов ваш документ будет выглядеть совершенно иначе.

Технология *внедрения шрифтов* (*font embedding*) позволяет включить шрифты прямо в документ. При открытии документа внедренные шрифты автоматически устанавливаются на другом компьютере, благодаря чему документ сохраняет прежний вид.

Внедрение шрифтов должно соответствовать лицензионным правилам использования шрифтов. Для определения лицензионного статуса шрифтов TrueType/OpenType следует вызвать функцию `GetOutlineTextMetrics` и затем проверить значение поля `otmfsType` структуры `OUTLINETEXTMETRIC`.

Метрики шрифтов

При форматировании текста Win32 GDI использует *метрики шрифта*, которые поясняются на рис. 2.27.



Рис. 2.27. Метрики шрифта

Отсчет всех размеров выполняется от так называемой *базовой линии* шрифта. На ней находится нижняя граница глифов большинства прописных букв. Все символы шрифта размещаются в прямоугольных ячейках.

- *Высота шрифта* `tmHeight` складывается из надстрочного интервала и подстрочного интервала.
- *Надстрочный интервал* `tmAscent` – это расстояние от базовой линии до верхней границы ячейки символа.
- *Подстрочный интервал* `tmDescent` – это расстояние от базовой линии до нижней границы ячейки символа.
- *Внутренний зазор* `tmInternalLeading` определяет пространство для размещения диакритических знаков, таких как знак акцента или умляут.
- *Внешний зазор* `tmExternalLeading` определяет минимальный интервал между соседними строками для многострочного текста.

В полиграфии размер шрифта измеряется в пунктах. Один пункт равен 0,01389 дюйма ($1/71,99424$). В компьютерной верстке пункт округляется до $1/72$ дюйма. Еще один типографский термин «кегль» определяется как разность высоты шрифта и внутреннего зазора.

Для получения информации о шрифте, выбранном в контекст устройства, предназначена функция `GetTextMetrics`:

```
BOOL GetTextMetrics(HDC hdc, LPTEXTMETRIC lptm);
```

Возвращаемая функцией информация размещается в структуре типа `TEXTMETRIC` с адресом `lptm`. Структура `TEXTMETRIC` имеет следующее определение:

```
typedef struct tagTEXTMETRIC {
    LONG tmHeight;           // высота символов
    LONG tmAscent;           // надстрочный интервал
    LONG tmDescent;          // подстрочный интервал
    LONG tmInternalLeading;  // внутренний зазор
```

продолжение ↗

```

LONG tmExternalLeading; // внешний зазор
LONG tmAveCharWidth; // средняя ширина символов
LONG tmMaxCharWidth; // максимальная ширина символов
LONG tmWeight; // вес (насыщенность) шрифта
LONG tmOverhang; // дополнительный промежуток для синтеза
// некоторых шрифтов
LONG tmDigitizedAspectX; // горизонтальный аспект устройства
LONG tmDigitizedAspectY; // вертикальный аспект устройства
TCHAR tmFirstChar; // первый символ, для которого в шрифте имеется
// глиф
TCHAR tmLastChar; // последний символ, для которого имеется глиф
TCHAR tmDefaultChar; // символ для замены символов, не имеющих глифа
TCHAR tmBreakChar; // символ для вставки в промежутки слов при
// выравнивании текста
BYTE tmItalic; // признак курсива
BYTE tmUnderlined; // признак подчеркивания
BYTE tmStruckOut; // признак перечеркивания
BYTE tmPitchAndFamily; // шаг, технология и семейство шрифта
BYTE tmCharSet; // набор символов
} TEXTMETRIC;

```

Первые семь полей структуры определяют основные метрики шрифта (см. рис. 2.27) в логических единицах, которые зависят от режима отображения, выбранного в контексте устройства.

Поле `tmWeight` определяет вес (жирность) шрифта. Эта величина может изменяться от 0 до 1000, но чаще всего она равна или 400 для нормального начертания шрифта, или 700 для полужирного начертания.

В поле `tmOverhang` содержится величина, на которую увеличивается ширина символов для синтезированных шрифтов, например для наклонных или жирных шрифтов, получаемых из нормального шрифта. Шрифты TrueType обычно не используют это поле, так как в них для каждого начертания создается свой шрифт.

Поля `tmDigitizedAspectX` и `tmDigitizedAspectY` прокомментированы в MSDN терминами *горизонтальный аспект устройства* (*horizontal aspect of the device*) и *вертикальный аспект устройства* (*vertical aspect of the device*) соответственно, для которых был создан текущий шрифт. По существу, они содержат те же значения, которые возвращает функция `GetDeviceCaps` с аргументами `LOGPIXELSX` и `LOGPIXELSY`, то есть «число пикселов на 1 логический дюйм по горизонтали» и «число пикселов на 1 логический дюйм по вертикали». При синтезе шрифтов имеют значение не абсолютные значения этих величин, а их соотношение `tmDigitizedAspectY / tmDigitizedAspectX`, называемое *аспектным соотношением* (*aspect ratio*).

Поле `tmDefaultChar` имеет довольно простой смысл. Если приложение пытается вывести неотображаемый символ, код которого отсутствует в шрифте, то вместо него будет выведен символ с кодом, содержащимся в поле `tmDefaultChar`. На рис. 2.22–2.25 показано, что Windows выводит в таких случаях контурный прямоугольник.

Поле `tmBreakChar` обычно содержит код пробела (0x20).

В поле `tmPitchAndFamily` содержится информация о шаге, технологии и семействе шрифта. Четыре старших бита указывают на шаг и технологию. Соответствующие константы приведены в табл. 2.22.

Четыре старших бита обозначают семейство шрифта. Возможные значения в виде констант Windows были приведены в табл. 2.21.

Таблица 2.22. Шаг и технология шрифта

Константа	Код	Описание
TMPF_FIXED_PITCH	0x01	Если этот бит установлен, шрифт является пропорциональным, в противном случае — моноширинным. Обратите внимание на то, что трактовка значения бита противоположна имени константы
TMPF_VECTOR	0x02	Если этот бит установлен, шрифт — векторный
TMPF_TRUETYPE	0x04	Если этот бит установлен, используется шрифт True Type
TMPF_DEVICE	0x08	Если этот бит установлен, используется шрифт устройства

В поле `tmCharSet` находится код используемого набора символов. Возможным значениям кода сопоставлены константы, определенные в файле `wingdi.h`. Некоторые из них показаны в табл. 2.23.

Таблица 2.23. Коды наборов символов

Константа	Код	Примечание
ANSI_CHARSET	0	
DEFAULT_CHARSET	1	
SYMBOL_CHARSET	2	
SHIFTJIS_CHARSET	128	
HANGUL_CHARSET	129	
GB2312_CHARSET	134	
CHINESEBIG5_CHARSET	136	
GREEK_CHARSET	161	Только для Windows NT/2000
TURKISH_CHARSET	162	Только для Windows NT/2000
HEBREW_CHARSET	177	Только для Windows NT/2000
ARABIC_CHARSET	178	Только для Windows NT/2000
RUSSIAN_CHARSET	204	Только для Windows NT/2000
EASTEUROPE_CHARSET	238	Только для Windows NT/2000
OEM_CHARSET	255	

Чтобы использовать функцию `GetTextMetrics`, сначала нужно определить структурную переменную типа `TEXTMETRIC`, например:

```
TEXTMETRIC tm ;
```

Затем необходимо получить дескриптор контекста устройства и вызвать функцию `GetTextMetrics`:

```
hDC = GetDC(hWnd) ;
GetTextMetrics(hDC, &tm) ;
```

После этого метрики текущего шрифта будут находиться в соответствующих полях переменной `tm`. В завершение не забудьте освободить контекст устройства:

```
ReleaseDC(hWnd, hDC) ;
```

Приведем пример. Допустим, что текущее разрешение экрана составляет 96 dpi¹, а в приложении используются: а) шрифт с гарнитурой «Arial», нормальным начертанием и высотой 18 логических единиц; б) режим отображения `MM_TEXT`, в котором логические единицы совпадают с физическими единицами, то есть пикселями.

¹ dpi — dot per inch (число точек на дюйм).

Тогда после вызова функции `GetTextMetrics` поля переменной `tm` будут заполнены следующими значениями:

```
{18, 15, 3, 2, 1, 7, 43, 400, 0, 96, 96, 30, 255, 31, 32, 0, 0, 0, 39, 204}
```

Обратите внимание на то, что интерпретация поля `tmPitchAndFamily` связана с его побитовой кодировкой, поэтому десятичное значение 39 стоит преобразовать к шестнадцатеричному коду `0x27`.

Логические шрифты

К счастью, при выводе текста приложениям не приходится напрямую общаться с физическими шрифтами. С физическими шрифтами работают шрифтовые драйверы, находящиеся в системе на одном уровне с драйверами графических устройств. В программах же используются так называемые логические шрифты.

Логический шрифт представляет собой объект GDI, описывающий требования к шрифту со стороны приложения. GDI анализирует запрошенные параметры и подбирает наиболее подходящий шрифт из тех, которые зарегистрированы в системе.

Объект логического шрифта находится под управлением GDI вместе с другими логическими объектами. Приложения работают с логическими шрифтами только через их дескрипторы, имеющие тип `HFONT`.

Стандартные шрифты

По умолчанию в контексте устройства выбран системный шрифт `SYSTEM_FONT`, основным и почти единственным преимуществом которого является то, что он всегда доступен для использования. Системный шрифт является растровым, содержит буквы переменной ширины, не имеющие засечек, и для него используется кодировка ANSI.

Однако в некоторых случаях может понадобиться шрифт с фиксированной шириной букв или шрифт в кодировке OEM¹. Вы можете получить дескриптор одного из стандартных (встроенных) шрифтов при помощи функции `GetStockObject`, передав в качестве аргумента одно из значений, указанных в табл. 2.24.

Таблица 2.24. Идентификаторы встроенных шрифтов

Идентификатор	Код	Описание
<code>OEM_FIXED_FONT</code>	10	Шрифт в кодировке OEM с фиксированной шириной букв
<code>ANSI_FIXED_FONT</code>	11	Шрифт в кодировке ANSI с фиксированной шириной букв
<code>ANSI_VAR_FONT</code>	12	Шрифт в кодировке ANSI с переменной шириной букв
<code>SYSTEM_FONT</code>	13	Системный шрифт в кодировке ANSI с переменной шириной букв, используется для отображения текста в меню, в заголовках окон, в элементах управления диалоговых окон, а также в клиентской области окна
<code>DEVICE_DEFAULT_FONT</code>	14	Шрифт, который используется для данного устройства по умолчанию
<code>SYSTEM_FIXED_FONT</code>	16	Шрифт в кодировке ANSI с фиксированной шириной букв. Использовался как системный шрифт в старых версиях (до Windows 3.0)
<code>DEFAULT_GUI_FONT</code>	17	Шрифт по умолчанию для объектов пользовательского интерфейса, таких как меню и диалоговые окна (только для Windows NT/2000)

¹ OEM — Original Equipment Manufacturers

Получив дескриптор встроенного шрифта, необходимо выбрать его в контекст устройства при помощи функции `SelectObject`. После окончания работы с встроенным шрифтом его удаление не требуется.

Создание логических шрифтов

Область применения стандартных шрифтов обычно ограничивается простым выводом элементов пользовательского интерфейса. Для любых других целей приходится создавать собственные логические шрифты. В GDI для этого предусмотрены функции `CreateFont`, `CreateFontIndirect` и `CreateFontIndirectEx`.

В этом разделе не будет рассматриваться функция `CreateFontIndirectEx`, которая поддерживает использование шрифтов нового формата OpenType фирмы Microsoft. При необходимости найти нужную информацию можно в MSDN.

Функция `CreateFont` использует для описания логического шрифта 14 параметров, поэтому она не слишком удобна в использовании. Вместо нее лучше пользоваться функцией `CreateFontIndirect`. Эта функция получает указатель на структуру `LOGFONT`, в которой упакованы эти же 14 параметров. Функция имеет следующий прототип:

```
HFONT CreateFontIndirect(CONST LOGFONT* lpf);
```

Структура `LOGFONT` определена в файле `wingdi.h` следующим образом:

```
typedef struct tagLOGFONT
{
    int lfHeight;
    int lfWidth;
    int lfEscapement;
    int lfOrientation;
    int lfWeight;
    BYTE lfItalic;
    BYTE lfUnderline;
    BYTE lfStrikeOut;
    BYTE lfCharSet;
    BYTE lfOutPrecision;
    BYTE lfClipPrecision;
    BYTE lfQuality;
    BYTE lfPitchAndFamily;
    char lfFaceName[LF_FACESIZE];
} LOGFONT;
```

Рассмотрим назначение ее полей:

- **lfHeight** — желательная высота шрифта в логических единицах. Поле может иметь положительное, нулевое или отрицательное значение. Положительное значение определяет требуемую высоту ячеек, то есть величину `tmHeight` на рис. 2.27. Если указано нулевое значение, то выбирается шрифт размером 12 пунктов (значение по умолчанию). Абсолютная величина отрицательного значения определяет высоту символов, то есть расстояние `tmHeight - tmInternalLeading`.
- **lfWidth** — желательная средняя ширина символов в логических единицах. Если поле `lfWidth` имеет нулевое значение, то аспектное соотношение устройства вывода сопоставляется с аспектным соотношением доступных шрифтов в поиске наиболее подходящего шрифта по этому параметру.
- **lfEscapement** — угол между базовой линией текста и осью X устройства в десятых долях градуса. Отсчет ведется против часовой стрелки.

- ❑ **lfOrientation** — угол между базовой линией каждого символа и осью *X* устройства в десятых долях градуса. Отсчет ведется против часовой стрелки. Если устройство работает в расширенном графическом режиме (**GM_ADVANCED**), доступном лишь в Windows NT/2000, то наклон **lfEscapement** и ориентация **lfOrientation** задаются независимо друг от друга. В совместимом графическом режиме (**GM_COMPATIBLE**) поле **lfEscapement** задает одновременно и наклон, и ориентацию символов. MSDN рекомендует в последнем случае устанавливать одно и то же значение для полей **lfEscapement** и **lfOrientation**.
- ❑ **lfWeight** — желательный вес (жирность) шрифта в интервале от 0 до 1000. Например, значение 400 создает нормальный шрифт, а 700 — полужирный шрифт. В табл. 2.25 приведены константы, которые можно использовать для установки данного поля. Значение **FW_DONTCARE** предписывает выбрать вес шрифта по умолчанию.

Таблица 2.25. Константы для определения веса шрифта¹

Константа	Значение
FW_DONTCARE	0
FW_THIN	100
FW_EXTRALIGHT	200
FW_ULTRALIGHT	200
FW_LIGHT	300
FW_NORMAL	400
FW_REGULAR	400
FW_MEDIUM	500
FW_SEMIBOLD	600
FW_DEMIBOLD	600
FW_BOLD	700
FW_EXTRABOLD	800
FW_ULTRABOLD	800
FW_HEAVY	900
FW_BLACK	900

- ❑ **lfItalic** — запрашивается курсивный шрифт, если значение поля равно **TRUE**.
- ❑ **lfUnderline** — запрашивается шрифт с подчеркиванием букв.
- ❑ **lfStrikeOut** — запрашивается шрифт с перечеркнутыми буквами.
- ❑ **lfCharSet** — набор символов шрифта. Вы можете использовать одну из констант, приведенных в табл. 2.23.
- ❑ **lfOutPrecision** — требуемая степень соответствия параметров выбираемого шрифта запрашиваемым характеристикам. Это поле указывает GDI критерии выбора между имеющимися в системе шрифтами. Наиболее часто используются значения из табл. 2.26.

¹ Можно использовать любое из указанных значений, однако следует иметь в виду, что многие шрифты содержат описания символов только для веса **FW_NORMAL**, **FW_REGULAR** и **FW_BOLD**.

Таблица 2.26. Константы для поля lfOutPrecision

Константа	Код	Описание
OUT_DEFAULT_PRECIS	0	Стандартный способ подбора шрифтов
OUT_TT_PRECIS	4	Отдается предпочтение шрифтам True Type
OUT_DEVICE_PRECIS	5	Отдается предпочтение шрифтам устройства вывода
OUT_RASTER_PRECIS	6	Отдается предпочтение растровым шрифтам
OUT_TT_ONLY_PRECIS	7	Используются только шрифты True Type
OUT_OUTLINE_PRECIS	8	Отдается предпочтение шрифтам True Type и другим контурным шрифтам

- **lfClipPrecision** — способ отсечения символа, частично попавшего за пределы региона отсечения. Поле может иметь одно или несколько значений из табл. 2.27, объединенных побитовой операцией ИЛИ. Из всех флагов к отсечению, собственно, относится только константа CLIP_DEFAULT_PRECIS.

Таблица 2.27. Константы для поля lfClipPrecision

Константа	Код	Описание
CLIP_DEFAULT_PRECIS	0	Стандартная процедура отсечения
CLIP_LH_ANGLES	0x10	Если этот флаг установлен, то направление вращения для всех шрифтов зависит от ориентации координатной системы (левосторонняя или правосторонняя). Если флаг не установлен, то шрифты устройств вращаются всегда против часовой стрелки, но вращение других шрифтов зависит от ориентации координатной системы
CLIP_EMBEDDED	0x80	Запрашивается использование внедренных шрифтов, доступных только для чтения

- **lfQuality** — качество вывода глифов. Можно использовать одну из констант, приведенных в табл. 2.28. Для шрифтов TrueType константы DRAFT_QUALITY и PROOF_QUALITY не имеют значения, поскольку контуры глифов свободно масштабируются до нужной величины.

Таблица 2.28. Константы для поля lfQuality

Константа	Код	Описание
DEFAULT_QUALITY	0	Внешний вид символов несущественен
DRAFT_QUALITY	1	Внешний вид символов менее важен по сравнению с другими атрибутами шрифта. Для растровых шрифтов это позволяет GDI синтезировать разные размеры шрифта, но с потерей качества
PROOF_QUALITY	2	Внешний вид символов более важен по сравнению с другими атрибутами шрифта, поэтому масштабирование растровых шрифтов запрещено
NONANTIALIASED_QUALITY	3	Сглаживание (antialiasing) запрещено
ANTIALIASED_QUALITY	4	Выполнять сглаживание (antialiasing), если оно поддерживается шрифтом и если размер шрифта не слишком велик и не слишком мал

- **lfPitchAndFamily** — задает шаг и семейство шрифта. Возможные значения определяются как объединение (логическая операция ИЛИ) одного из значений, приведенных в табл. 2.29, и одного из значений, приведенных в табл. 2.21.

Таблица 2.29. Константы, указывающие шаг шрифта

Константа	Код	Описание
DEFAULT_PITCH	0x0	Шаг по умолчанию
FIXED_PITCH	0x1	Моноширинный шрифт
VARIABLE_PITCH	0x2	Пропорциональный шрифт

□ **fFaceName** — строка с завершающим нулевым байтом, задающая имя гарнитуры шрифта. Размер строки, включая завершающий нуль, не должен превышать 32 байта. Имена гарнитур шрифтов, установленных в системе в настоящий момент, можно получить при помощи функции `EnumFontFamiliesEx`. Если поле **fFaceName** содержит пустую строку, то GDI выбирает первый попавшийся шрифт, который удовлетворяет другим заданным атрибутам.

Перед вызовом функции `CreateFontIndirect` вы должны определить переменную типа `HFONT` для создаваемого шрифта:

```
HFONT hFont;
```

а также переменную типа `LOGFONT`:

```
LOGFONT lf;
```

После этого заполните ее поля нужными значениями, определяющими параметры шрифта. Неиспользованные поля следует установить в 0. Более эффективным вариантом является объявление переменной `lf` со спецификатором `static`:

```
static LOGFONT lf;
```

так как в этом случае компилятор обеспечит инициализацию всех полей структуры `lf` нулевыми значениями.

Если вы проектируете приложение, в котором пользователь должен иметь возможность выбора используемого шрифта, то для заполнения полей структурной переменной `lf` следует воспользоваться функцией `ChooseFont`. Она вызывает *стандартное диалоговое окно* для выбора шрифта. Эта возможность будет рассмотрена в главе 7 (раздел «Диалоговые окна общего пользования»).

После того как переменная `lf` подготовлена, вызовите функцию `CreateFontIndirect`:

```
hFont = CreateFontIndirect(&lf);
```

Полученный дескриптор `hFont` созданного логического шрифта выберите в контекст устройства:

```
HFONT hOldFont = (HFONT)SelectObject(hdc, hFont);
```

Далее вы можете работать с этим шрифтом в функциях вывода текста, а после того, как он перестанет быть нужным, верните в контекст устройства предшествующий шрифт с одновременным удалением шрифта `hFont`:

```
DeleteObject(SelectObject(hdc, hOldFont));
```

Пример работы с логическими шрифтами приведен в листинге 2.3.

Подстановка шрифта

Новый логический шрифт, созданный функцией `CreateFont` или `CreateFontIndirect`, не ассоциируется ни с каким физическим шрифтом, пока он не будет выбран в контекст устройства вызовом функции `SelectObject`. Процесс поиска и выбора нужного физического шрифта, реализуемый системой в ходе выполнения функции `SelectObject`, называется *подстановкой* или *отображением шрифта* (*font mapping*).

Процесс отображения логического шрифта довольно сложен. GDI сравнивает параметры, заданные в структуре `LOGFONT`, с параметрами различных шрифтов, доступных для графического устройства, выбирая наиболее подходящий шрифт. Для сравнения используются штрафные очки, которые имеют разные весовые коэффициенты. Выбирается тот шрифт, для которого штрафная сумма будет наименьшей.

Наиболее важным фактором при подборе физического шрифта является набор символов, который задается в поле `lfCharSet`. При несовпадении этого атрибута очень велика вероятность, что символы будут выводиться совершенно неверными глифами. Следующее по важности поле — это `lfOutPrecision`. Этот показатель ограничивает рассматриваемые наборы символов определенными типами шрифтов. Затем оценивается поле `lfFaceName`, а после него — поле `lfPitchAndFamily`. Моноширинные шрифты сильно отличаются по внешнему виду от пропорциональных, поэтому этот атрибут также является важным при подстановке. После сравнения указанных полей GDI сравнивает высоту символов, заданную в поле `lfHeight`, а затем поля `lfWidth`, `lfItalic`, `lfUnderline`, `lfStrikeOut`.

Получение информации о выбранном физическом шрифте

После того как логический шрифт выбран в контекст устройства, приложение может получить дополнительную информацию о подобранном физическом шрифте и его метриках.

Функция `GetTextMetrics`, позволяющая выполнить эту задачу, уже рассматривалась ранее.

Но можно использовать и другую функцию, прототип которой приведен ниже:

```
int GetTextFace(HDC hDC, int nCount, LPTSTR lpFaceName);
```

Эта функция получает имя гарнитуры физического шрифта. Оно может отличаться от имени гарнитуры, которое было задано в структуре `LOGFONT`. Искомое имя в виде строки с завершающим нулевым символом записывается в символьный буфер с адресом `lpFaceName`. Длина буфера (количество символов) определяется параметром `nCount`. В случае успешного завершения функция возвращает количество символов, скопированных в буфер, включая и завершающий нулевой символ. В случае ошибки возвращается нулевое значение. Функцию `GetTextFace` можно вызвать, указав на месте третьего параметра значение `NULL`. В этом случае она просто вернет количество символов в имени гарнитуры.

Функция `GetTextCharSet(HDC hDC)` возвращает идентификатор набора символов для текущего шрифта, выбранного в контекст устройства.

Есть и другие функции для получения дополнительной информации о текущей реализации физического шрифта, к которым относятся `GetFontLanguageInfo`, `GetTextCharSetInfo` и `GetOutlineTextMetrics`. Они используются при нетривиальном форматировании текста, например при непосредственной работе с глифами. Описание этих функций вы можете найти в MSDN или в книге [6].

Вывод текста

Win32 GDI обеспечивает полный набор функций для форматирования и рисования текста в клиентской области окна или на бумажной странице принтера. Эти функции могут быть разделены на те, которые форматируют текст, подготавливая его для вывода, и те, которые действительно отображают текст. *Форматиру-*

ющие функции выравнивают текст, устанавливают межсимвольные промежутки, изменяют протяженность разделительных символов, устанавливают цвет текста и цвет фона графических элементов. *Рисующие функции* выводят отдельные символы или целые строки текста.

Простой вывод текста

Простейшая функция вывода текстовой строки `TextOut` имеет следующий прототип:

```
BOOL TextOut (
    HDC hdc,           // дескриптор контекста устройства
    int nXStart,       // х-координата стартовой позиции
    int nYStart,       // у-координата стартовой позиции
    LPCTSTR lpString, // указатель на символьную строку
    int cbString       // число символов в строке
);
```

Функция обеспечивает вывод строки с адресом `lpString`, размещая текст в заданной позиции с учетом текущего режима выравнивания. При выводе используются текущие значения атрибутов контекста устройства — шрифт, цвет текста и цвет фона графических элементов, режим смешивания фона и многие другие. Функция не распознает конец строки `lpString` по завершающему нулевому символу, поэтому количество выводимых символов задается параметром `cbString`. Символы строки должны входить в набор символов текущего шрифта. Позиционирование текста зависит от текущего режима выравнивания.

Текущий режим выравнивания текста реализован как атрибут контекста устройства, определяющий правила позиционирования текста. Он указывает, что считать *опорной точкой* (*reference point*): точку (`nXStart`, `nYStart`) или текущую позицию пера в контексте устройства. Также режим выравнивания определяет, как позиционировать строку текста (обрамляющий прямоугольник) относительно опорной точки и как выводить текст: слева направо или справа налево.

Значение этого атрибута можно изменить при помощи функции `SetTextAlign`:

```
UINT SetTextAlign(HDC hdc, UINT fMode);
```

передавая второму параметру битовую маску, образованную объединением¹ флагов, перечисленных в табл. 2.30. При этом из каждой группы флагов можно использовать только один флаг. Флаги с кодом 0x00 являются значениями по умолчанию.

Таблица 2.30. Флаги выравнивания текста

Группа	Флаг	Код	Описание
Что считать опорной точкой. Обновлять или не обновлять текущую позицию пера	TA_NOUPDATECP	0x00	В качестве опорной точки использовать точку (<code>nXStart</code> , <code>nYStart</code>). Текущая позиция пера не обновляется после вывода текста
	TA_UPDATECP	0x01	В качестве опорной точки использовать текущую позицию пера. Текущая позиция пера обновляется после вывода текста

¹ Поразрядная логическая операция ИЛИ.

Группа	Флаг	Код	Описание
Выравнивание по горизонтали	TA_LEFT	0x00	Опорная точка задает левую границу обрамляющего прямоугольника
	TA_RIGHT	0x02	Опорная точка задает правую границу обрамляющего прямоугольника
	TA_CENTER	0x06	Горизонтальный центр обрамляющего прямоугольника совмещается с опорной точкой
Выравнивание по вертикали	TA_TOP	0x00	Опорная точка задает верхнюю границу обрамляющего прямоугольника
	TA_BOTTOM	0x08	Опорная точка задает нижнюю границу обрамляющего прямоугольника
	TA_BASELINE	0x18	Базовая линия текста совмещается с опорной точкой
Направление вывода текста	TA_RTLREADING	0x100	Текст выводится справа налево (применяется только для арабского языка или иврита)

Пример использования функции `TextOut` можно найти в листинге 2.1, а также в некоторых других фрагментах программного кода.

Вывод текста с табуляцией

К сожалению, функция `TextOut` не обрабатывает управляющие символы, в том числе символы перевода строки `\n` и табуляции `\t`. Встретив любой такой символ, она просто выведет глиф • (для системного шрифта `SYSTEM_FONT`) или глиф □ для несистемного шрифта.

В то же время табуляция широко используется в простейших текстовых редакторах для выравнивания текста по столбцам, облегчающего восприятие информации. Для обеспечения работы с табулированным текстом Windows содержит функции `TabbedTextOut` и `GetTabbedTextExtent`.

Функция `TabbedTextOut` имеет следующий прототип:

```
LONG TabbedTextOut(HDC hDC, int X, int Y, LPCTSTR lpString, int nCount,
                    int iNumTabs, CONST LPINT lpnTabStops, int xTabOrigin);
```

Первые пять параметров имеют то же значение, что и у функции `TextOut`. Шестой параметр, `iNumTabs`, определяет количество позиций табуляции. Седьмой параметр, `lpnTabStops`, содержит указатель на массив позиций табуляции, заданных в логических единицах. Позиции табуляции должны быть отсортированы в возрастающем порядке.

Если шестой параметр равен нулю и одновременно седьмой параметр равен `NULL`, то позиции табуляции устанавливаются через одинаковые промежутки, равные восьмикратной средней ширине символов. Если шестой параметр равен единице, то первый элемент массива `lpnTabStops` содержит число символьных позиций, которое каждый раз прибавляется для определения следующей позиции табуляции.

Последний параметр, `xTabOrigin`, задает логическую координату по горизонтали *точки отсчета* позиций табуляции. Часто бывает удобно определить эту точку так, чтобы она совпадала с начальной позицией вывода строки, поскольку в этом случае массив `lpnTabStops` перестает быть зависимым от конкретной позиции вывода.

Если выводимая строка содержит символы табуляции `\t`, то GDI отображает начало строки, пока не обнаружит символ табуляции. После этого GDI просматривает массив позиций табуляций, и если первая позиция подходит для продол-

жения вывода (то есть она находится правее границы последнего выведенного символа строки), то вывод продолжается с этой позиции. В противном случае GDI берет следующую позицию из массива и проверяет ее и т. д. То же самое происходит при обнаружении следующего символа табуляции. Таким образом, функция не гарантирует, что символы после *n*-го символа табуляции будут выводиться в *n*-й позиции табуляции.

Позиции табуляции в массиве *lpnTabStops* могут быть отрицательными. В этом случае GDI использует абсолютное значение указанной величины, но выравнивает текст по правому краю перед заданной позицией, вместо выравнивания по левому краю после нее.

В случае успешного завершения функция *TabbedTextOut* возвращает 32-разрядное число, старшее слово которого содержит высоту, а младшее — ширину выведенной строки. В случае неудачного завершения работы возвращается нулевое значение.

Функция *GetTabbedTextExtent* возвращает размеры табулированного текста, не выводя его.

Следующий фрагмент кода показывает пример использования функции *TabbedTextOut*. Оконная процедура, код которой приводится ниже, обеспечивает вывод на экран небольшой таблицы:

```
HRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    int tabstop[] = { -130, 150, 250 };
    int i;
    const char* lines[] = {
        "Group"\t"Result" "\t" "Function" "\t" "Parameters",
        "Font" "\t" "HFONT" "\t" "CreateFont" "\t" "(int nHeight, ...)",
        "Text" "\t" "COLORREF""\t" "SetTextColor""\t" "(HDC hdc, ...)",
        "Text" "\t" "BOOL" "\t" "TextOut" "\t" "(HDC hdc, ...)"
    };
    int x = 50, y = 50;

    switch (uMsg)
    {
        case WM_PAINT:
            hDC = BeginPaint(hWnd, &ps);

            for (i = 0; i < 4; i++) {
                if (i == 1) y += 10;
                y += HIWORD(TabbedTextOut(hDC, x, y, lines[i], strlen(lines[i]),
                    sizeof(tabstop)/sizeof(int), tabstop, x));
            }

            EndPaint(hWnd, &ps);
            break;

        case WM_DESTROY:
            PostQuitMessage(0);
            break;

        default:
            return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }
    return 0;
}
```

Таблица формируется в цикле `for`. Обратите внимание на то, что значение, возвращаемое функцией `TabbedTextOut`, мы используем для вычисления смещения, чтобы получить координату y следующей строки выводимого текста. Последнему параметру функции передается то же значение x , которое задает абсциссу начала вывода строки. Поэтому, определяя абсолютные значения позиций табуляции, GDI прибавляет к величинам, извлекаемым из массива `tabstop`, величину x . Результат работы этого кода показан на рис. 2.28.

Заметим, что горизонтальная линейка, показывающая физические координаты по горизонтали, нарисована соответствующими вызовами функций `MoveToEx`, `LineTo` и `TextOut`. Чтобы не загромождать пример, эти инструкции в программе не показаны. Обратите внимание на выравнивание по правому краю второго столбца таблицы относительно позиции 180, вычисленной в результате сложения $50 + 130$.

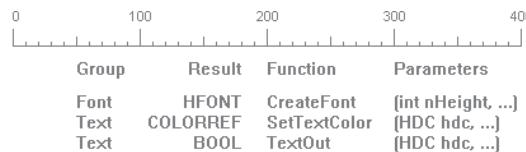


Рис. 2.28. Вывод табулированного текста

Межсимвольные интервалы

В контексте устройства имеется специальный атрибут, управляющий расстоянием между символами, — *межсимвольный интервал (extra space)*. Межсимвольный интервал добавляется к каждому символу, включая символы пробела, когда GDI выводит строку текста. По умолчанию этот атрибут равен нулю. Функция `SetTextCharacterExtra` присваивает ему новое целочисленное значение в логических единицах, возвращая предыдущее значение. Функция `GetTextCharacterExtra` возвращает текущее значение межсимвольного интервала.

Межсимвольные интервалы могут использоваться как для разрядки, так и для уплотнения текста. Чтобы почувствовать, какие возможности дает функция `SetTextCharacterExtra`, модифицируйте предыдущую программу следующим образом. Фрагмент с циклом `for` надо заменить таким кодом:

```
SetTextCharacterExtra(hdc, 4);
for (i = 0; i < 4; i++) {
    if (i == 1) {
        y += 10;
        SetTextCharacterExtra(hdc, 0);
    }
    y += HIWORD(TabbedTextOut(hdc, x, y, lines[i], strlen(lines[i]),
        sizeof(tabstop)/sizeof(int), tabstop, x));
}
```

После этого запустите программу и посмотрите, как изменится отображение текста на экране.

Выравнивание по ширине (выключка¹)

Функция `TextOut` обеспечивает выравнивание текста по левому или правому краю, а также по центру. Вид выравнивания зависит от текущего режима выравнивания

¹ Типографский термин, означающий увеличение межсловных интервалов в строке для выравнивания по ширине.

ния. Но при этом функция не поддерживает выравнивание по ширине (выключку). Выравнивание по ширине означает, что левый край текста должен быть выровнен по левой границе области, а правый край — по правой границе. Это достигается равномерным увеличением протяженности пробелов, разделяющих слова в строке.

Для решения указанной проблемы GDI позволяет использовать функции `GetTextExtentPoint32` и `SetTextJustification`.

Допустим, что надо реализовать выключку при выводе строк текста в область с левой границей `x` и правой границей `x + width`. Предполагается, что протяженность строк не превышает `width` логических единиц. План решения обычно содержит следующие шаги:

1. С помощью инструкций

```
SIZE size;  
GetTextExtentPoint32(hdc, lpString, nCount, &size);
```

получить размеры обрамляющего прямоугольника для строки `lpString`, содержащей `nCount` символов. Эти размеры возвращаются в виде значений полей параметра `size`. Поле `size.cx` будет содержать ширину обрамляющего прямоугольника, а поле `size.cy` — высоту обрамляющего прямоугольника.

2. Подсчитать количество разделительных символов `nBreak` в строке `lpString`. Разделительным символом обычно является пробел, но он может быть переопределен в каком-либо шрифте на другой символ. Если нужно уточнить, какой символ на самом деле является разделительным, используйте функцию `GetTextMetrics` с последующим анализом поля `tmBreakChar` структуры `TEXTMETRIC`.
3. Вычислить размер пространства, которое необходимо распределить между `nBreak` разделительными символами:

```
int breakExtra = width - size.cx;
```

4. Вызвать функцию

```
SetTextJustification(hdc, breakExtra, nBreak);
```

передав ей определенные выше величины `breakExtra` и `nBreak`. Функция `SetTextJustification` присвоит атрибуту контекста устройства, отвечающему за выключку текста, величину дополнительного интервала, используемую после этого функциями `TextOut` и `ExtTextOut`.

5. Позаботиться о том, чтобы текущий режим выравнивания текста содержал флаг `TA_LEFT` (значение по умолчанию). После этого уже можно вызвать функцию `TextOut` для отображения строки текста.

Следует отметить, что при каждом вызове функции `SetTextJustification` накапливается погрешность, если величина `breakExtra` не делится нацело на `nBreak`. Чтобы эта погрешность не повлияла на последующую работу функции `GetTextExtentPoint32` при выводе многострочного текста, рекомендуется перед обращением к `GetTextExtentPoint32` сбросить накопленную погрешность вызовом функции

```
SetTextJustification(hdc, 0, 0);
```

В листинге 2.1 приводится программа, в которой демонстрируется описанная технология выравнивания текста по ширине.

Листинг 2.1. Проект TextJust

```
//////////  
// TextJust.cpp  
#include <windows.h>  
#include "KWnd.h"  
  
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);  
BOOL TextJustOut(HDC hdc, int x, int y, LPCTSTR lpStr, int width,  
    char breakChar = ' ');  
  
//////////  
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,  
    LPSTR lpCmdLine, int nCmdShow)  
{  
    MSG msg;  
    KWnd mainWnd("Text Justification", hInstance, nCmdShow, WndProc);  
  
    while (GetMessage(&msg, NULL, 0, 0)) {  
        TranslateMessage(&msg);  
        DispatchMessage(&msg);  
    }  
    return msg.wParam;  
}  
  
//////////  
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)  
{  
    HDC hDC;  
    PAINTSTRUCT ps;  
  
    char* text[6] =  
    { "Рассуждай",  
        "Рассуждай ТОКМО",  
        "Рассуждай ТОКМО О ТОМ",  
        "Рассуждай ТОКМО О ТОМ, о чем понятия",  
        "Рассуждай ТОКМО О ТОМ, о чем понятия твои тебе",  
        "Рассуждай ТОКМО О ТОМ, о чем понятия твои тебе сие дозволяют."  
    };  
  
    int x = 20, y = 20;  
    SIZE size;  
    int nWidth, i;  
  
    switch (uMsg)  
    {  
        case WM_PAINT:  
            hDC = BeginPaint(hWnd, &ps);  
  
            GetTextExtentPoint32(hDC, text[5], strlen(text[5]), &size);  
            nWidth = size.cx;  
            for (i = 0; i < 6; i++) {  
                TextJustOut(hDC, x, y, text[i], nWidth);  
                y += size.cy + 2;  
            }  
  
            EndPaint(hWnd, &ps);  
    }  
}
```

продолжение ↗

Листинг 2.1 (продолжение)

```

break;

case WM_DESTROY:
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}

return 0;
}

///////////////////////////////
BOOL TextJustOut(HDC hdc, int x, int y, LPCTSTR lpStr, int width,
    TCHAR breakChar)
{
    SIZE size;
    int nCount = strlen(lpStr);

    SetTextJustification(hdc, 0, 0);
    GetTextExtentPoint32(hdc, lpStr, nCount, &size);

    int nBreak = 0;
    for (int i = 0; i < nCount; ++i)
        if (lpStr[i] == breakChar)
            nBreak++;
    int breakExtra = width - size.cx;
    if (breakExtra < 0) breakExtra = 0;
    SetTextJustification(hdc, breakExtra, nBreak);
    return TextOut(hdc, x, y, lpStr, nCount);
}
/////////////////////////////

```

В проекте используется класс `KWnd`, поэтому не забудьте скопировать в каталог проекта файлы `KWnd.h` и `KWnd.cpp`, код которых приведен в листинге 1.2, и подключить их в состав проекта при помощи окна `Project Workspace` интегрированной среды.

Код, отвечающий за выравнивание по ширине, инкапсулирован в функцию `TextJustOut`. В оконной процедуре функция `TextJustOut` вызывается, чтобы вывести шесть строк из массива `text`. Ширина области вывода `nWidth` определяется по самой длинной строке с адресом `text[5]`. Результат работы программы показан на рис. 2.29.

Рассуждай
Рассуждай ТОКМО ТОКМО
Рассуждай ТОКМО 0 ТОМ. 0 ЧЕМ ПОНЯТИЯ
Рассуждай ТОКМО 0 ТОМ, 0 ЧЕМ ПОНЯТИЯ ТВОИ ТЕБЕ
Рассуждай ТОКМО 0 ТОМ, О ЧЕМ ПОНЯТИЯ ТВОИ ТЕБЕ СИЕ ДОЗВОЛЯЮТ.

Рис. 2.29. Выравнивание текста по ширине

Абзацное форматирование в прямоугольнике

В пользовательском интерфейсе Windows часто требуется вывести длинный текст в прямоугольнике, способном вместить несколько строк. Или, напротив, короткий текст поместить в прямоугольнике с учетом заданного позиционирования. Для решения подобных проблем Win32 GDI предоставляет функции `DrawText` и `DrawTextEx`.

Функция `DrawText` имеет следующий прототип:

```
int DrawText(
    HDC hdc,           // дескриптор контекста устройства
    LPCTSTR lpString, // указатель на текстовую строку
    int nCount,        // длина текста
    LPRECT lpRect,     // прямоугольник, в котором размещается текст
    UINT uFormat       // флаги форматирования
);
```

Так же как и другие функции вывода текста, функция `DrawText` принимает в качестве параметров указатель на символьную строку и длину строки. Однако функция ведет себя более интеллектуально по отношению к строкам с завершающим нулевым символом. Вы можете передать значение `-1` на месте параметра `nCount`, и функция сама определит длину строки. Если же строка не имеет завершающего нулевого символа, то благороднее будет указать ее длину в аргументе `nCount`.

Функция `DrawText` выводит текстовую строку в прямоугольной области, заданной параметром `lpRect`. Последний параметр, `uFormat`, определяет метод форматирования текста. Его значение представляет собой битовую маску, образованную объединением флагов форматирования. Наиболее часто употребляемые флаги приведены в табл. 2.31 (полный перечень флагов см. в MSDN).

Таблица 2.31. Флаги форматирования для функции `DrawText`

Флаг	Код	Описание
<code>DT_LEFT</code>	<code>0x0000</code>	Выравнивание текста влево
<code>DT_CENTER</code>	<code>0x0001</code>	Центрирование по горизонтали
<code>DT_RIGHT</code>	<code>0x0002</code>	Выравнивание текста вправо
<code>DT_TOP</code>	<code>0x0000</code>	Начало размещения — в верхней части прямоугольника
<code>DT_VCENTER</code>	<code>0x0004</code>	Центрирование по вертикали (используется только вместе с <code>DT_SINGLELINE</code>)
<code>DT_BOTTOM</code>	<code>0x0008</code>	Размещение внизу прямоугольника (используется только вместе с <code>DT_SINGLELINE</code>)
<code>DT_WORDBREAK</code>	<code>0x0010</code>	Перенос текста на следующую строку после окончания очередного слова, если оставшаяся часть текста не помещается по ширине прямоугольника вывода. Флаг работает, только если не указан флаг <code>DT_SINGLELINE</code>
<code>DT_SINGLELINE</code>	<code>0x0020</code>	Вывод текста только в одну строку. Символы возврата каретки и перевода строки не воспринимаются как управляющие символы (выводятся как • или □)
<code>DT_EXPANDTABS</code>	<code>0x0040</code>	Интерпретация символов <code>\t</code> как управляющих символов, задающих табуляцию. По умолчанию шаг табуляции равен восьмикратной средней ширине символов
<code>DT_NOPREFIX</code>	<code>0x0800</code>	Отменяет обработку префиксов &. Обычно <code>DrawText</code> интерпретирует префикс & как директиву «подчеркнуть следующий за префиксом символ», а последовательность && — как директиву вывести одиничный символ &. С флагом <code>DT_NOPREFIX</code> символ & воспринимается наравне с другими символами, а не как префикс
<code>DT_PATH_ELLIPSIS</code>	<code>0x4000</code>	Если строка не помещается по ширине прямоугольника, часть символов в середине строки замещается многоточием (используется только вместе с <code>DT_SINGLELINE</code>)
<code>DT_END_ELLIPSIS</code>	<code>0x8000</code>	Если строка не помещается по ширине прямоугольника, часть символов в конце строки замещается многоточием (используется только вместе с <code>DT_SINGLELINE</code>)

Если параметр *uFormat* имеет нулевое значение, то Windows интерпретирует текст как ряд строк, разделенных символами возврата каретки ('\r' или 0x0D) или символами конца строки ('\n' или 0x0A). Вывод текста производится, начиная с верхнего левого угла прямоугольника. Возврат каретки или конец строки воспринимаются как команда перейти на новую строку. Вывод новой строки начинается под предыдущей строкой с интервалом, равным высоте символа в строке.

Любой текст, в том числе и части букв, попадающие при выводе правее или ниже границ прямогоугольника, отсекаются. Можно предотвратить потерю текста от отсечения по правой границе, задав флаг **DT_WORDBREAK**.

Если выводится короткий текст, то рекомендуется указывать флаг **DT_SINGLELINE**. Это позволяет использовать все флаги позиционирования текста в прямомугольнике.

В случае успешного завершения функция **DrawText** обычно возвращает высоту выведенного текста. Для многострочного текста возвращается суммарная высота всех строк. Исключение составляют случаи использования функции с флагом **DT_VCENTER** или **DT_BOTTOM**. В этих двух случаях функция возвращает смещение нижней границы текста относительно верхней границы прямогоугольника.

Если функция по каким-либо причинам не смогла нормально выполнить свою работу, то возвращается нулевое значение.

Покажем примеры использования функции **DrawText**. На рис. 2.30 приведен результат трехкратного вызова функции

```
DrawText(hdc, text1, -1, &rect, uFormat);
```

для вывода строки

```
char text1[] = "&Open C:\\WINNT\\system32\\gdi32.dll";
```

со следующими значениями параметра *uFormat*:

- 0;
- DT_SINGLELINE | DT_CENTER | DT_VCENTER | DT_NOPREFIX**;
- DT_SINGLELINE | DT_RIGHT | DT_BOTTOM**.

Прямоугольник *rect* отображен на рисунке дополнительным вызовом функции **Rectangle**.

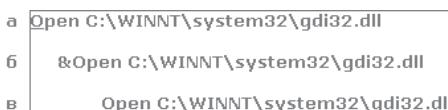


Рис. 2.30. Вывод строки *text1* с разным позиционированием в прямомугольнике *rect*

На рис. 2.31 демонстрируется вывод этого же текста в более узком прямомугольнике, который не может уместить весь текст. Трем вызовам функции соответствуют значения параметра *uFormat*:

- 0;
- DT_SINGLELINE | DT_VCENTER | DT_END_ELLIPSIS**;
- DT_SINGLELINE | DT_BOTTOM | DT_PATH_ELLIPSIS**.

Обратите внимание на то, что при первом вызове функции **DrawText** конец строки отсекается, при втором — хвостовой фрагмент строки заменяется многоточием, при третьем — замена многоточием осуществляется в средней части, чтобы показать завершающую часть строки.

На рис. 2.32 показан результат вызова функции
`DrawText(hDC, text2, -1, &rect, DT_SINGLELINE | DT_VCENTER | DT_EXPANDTABS);`
для вывода строки
`char text2[] = "&Open \tC:\\WINNT\\system32\\gdi32.dll";`
содержащей символ табуляции '\t'.

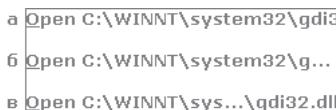


Рис. 2.31. Вывод строки text1 в прямоугольнике, имеющем недостаточную ширину для размещения текста



Рис. 2.32. Вывод строки text2, содержащей символ табуляции

На рис. 2.33 приведен вывод текста
`char text3[] = "Warning: this computer program \n is protected by copyright law \n and international treaties";`
в результате вызова функции
`DrawText(hDC, text3, -1, &rect, DT_CENTER);`



Рис. 2.33. Вывод текста, содержащего символы конца строки (\n), с флагом DT_CENTER

Наконец, аналогичный текст, но не содержащий символов конца строки
`char text4[] = "Warning: this computer program is protected by copyright law and international treaties";`
выводится с помощью вызова
`DrawText(hDC, text4, -1, &rect, DT_WORDBREAK);`

так, как это показано на рис. 2.34.



Рис. 2.34. Преобразование односторочного текста в многострочный при помощи флага DT_WORDBREAK

Функция `DrawTextEx` имеет следующий прототип:

```
int DrawTextEx(HDC hDC, LPTSTR lpcchText, int cchText, LPRECT lpRect,
    UINT dwDTFormat, LPDRAWTEXTPARAMS lpDTPParams);
```

Она отличается от `DrawText` наличием дополнительного параметра, который содержит указатель на структуру `DRAWTEXTPARAMS`. Эта структура определяет расстояния между позициями табуляции, левые и правые поля для вывода текста,

а также используется для возвращения длины выведенной строки. Более подробная информация об этой функции приведена в MSDN.

Функция ExtTextOut

Функция `ExtTextOut`, представляющая собой расширенную версию функции `TextOut`, имеет следующий прототип:

```
BOOL ExtTextOut(HDC hdc, int X, int Y, UINT fuOptions, CONST RECT* lprc,
    LPCTSTR lpString, UINT cbCount, CONST INT* lpDx);
```

Параметры `hdc`, `X`, `Y`, `lpString`, `cbCount` имеют тот же смысл, что и аналогичные параметры функции `TextOut`. Поэтому следующий вызов:

```
ExtTextOut(hdc, x, y, 0, NULL, lpString, cbCount, NULL);
```

эквивалентен вызову функции

```
TextOut(hdc, x, y, lpString, cbCount);
```

Остается рассмотреть три новых параметра. Параметр `fuOptions` содержит набор флагов, которые управляют интерпретацией других параметров. Необязательный параметр¹ `lprc` указывает на прямоугольник, который исполняет роль прямоугольника отсечения, или прямоугольника фона для выводимого текста, или того и другого вместе. Необязательный параметр `lpDx` содержит указатель на массив расстояний между ячейками символов. В табл. 2.32 перечислены допустимые значения флагов, которые могут объединяться в битовую маску, используемую в качестве значения параметра `fuOptions`.

Таблица 2.32. Флаги функции ExtTextOut

Флаг	Код	Описание
ETO_OPAQUE	0x0002	Перед выводом текста прямоугольник <code>lprc</code> закрашивается цветом фона графических элементов
ETO_CLIPPED	0x0004	Текст отсекается по прямоугольнику <code>lprc</code>
ETO_GLYPH_INDEX	0x0010	Параметр <code>lpString</code> указывает на массив индексов глифов (вместо кодов символов), который можно получить вызовом функции <code>GetCharacterPlacement</code> . Индексы глифов всегда являются 16-разрядными величинами
ETO_RTLREADING	0x0080	Для шрифтов арабского языка и иврита текст выводится справа налево
ETO_NUMERICSLOCAL	0x0400	Числа выводятся символами национальных алфавитов
ETO_NUMERICSLATIN	0x0800	Числа выводятся арабскими цифрами
ETO_IGNORELANGUAGE	0x1000	Не выполнять дополнительную языковую обработку
ETO_PDY	0x2000	Параметр <code>lpDx</code> указывает на массив пар, в которых первое число определяет шаг по горизонтали между двумя смежными ячейками символов, а второе — относительное смещение этих ячеек по вертикали

Одним из возможных применений функции `ExtTextOut` является вывод текста в ячейки некоторой таблицы. Если вы хотите заполнить фон ячеек таблицы цветом фона графических элементов, который используется при выводе текста, то можно передать прямоугольник ячейки функции `ExtTextOut` и установить флаги `ETO_OPAQUE` и `ETO_CLIPPED`.

¹ Может иметь значение `NULL`.

Также функция позволяет более точно размещать глифы символов при использовании флага `ETO_GLYPH_INDEX`. Такое размещение бывает необходимо для графических приложений, имеющих режим отображения текста `WYSIWYG`. В этом режиме обеспечивается печать документа на принтере точно в таком виде, в каком он представлен на экране. Более подробную информацию об использовании функции `ExtTextOut` для решения таких проблем можно найти в книге [6].

Нетривиальный вывод текста

В современных текстовых редакторах и других графических пакетах решаются различные нетривиальные задачи вывода текста. Неполный перечень таких задач приведен в следующем списке:

- ❑ Управление кернингом (контекстная регулировка межсимвольных расстояний).
- ❑ Закраска текста кистью.
- ❑ Работа с текстом в растровом формате.
- ❑ Применение траекторий GDI при выводе текста.

Эти вопросы не рассматриваются в книге, так как это привело бы к неоправданному увеличению объема издания. Достаточно подробно они освещаются в книге [6].

Полосы прокрутки и вывод текста

При выводе текста может возникнуть ситуация, когда в клиентской области окна недостаточно места для его размещения. Например, длина строк может оказаться больше ширины окна, а количество строк слишком большим, чтобы поместиться в окне с данной высотой. *Полосы прокрутки (scroll bars)* являются самым удобным решением этой проблемы. Они просты в использовании и обеспечивают удобный просмотр информации.

Следует различать два вида полос прокрутки:

- ❑ *полоса прокрутки окна*;
- ❑ *полоса прокрутки – элемент управления типа Scroll bar*.

Второй вид полос прокрутки рассматривается в главе 7.

Полосы прокрутки окна, называемые также *стандартными полосами прокрутки*, размещаются вдоль правой и нижней рамок окна, как показано на рис. 2.35. Чтобы они появились, достаточно при вызове функции `CreateWindow` указать флаги `WS_VSCROLL` (вертикальная прокрутка) и `WS_HSCROLL` (горизонтальная прокрутка) в параметре `dwStyle`. Вертикальная полоса прокрутки позволяет пользователю просматривать информацию в окне, прокручивая ее вниз или вверх. Горизонтальная полоса прокрутки позволяет перемещать информацию вправо или влево. Терминология полос прокрутки, связанная с направлениями прокрутки, ориентирована на пользователя. Фактически программа перемещает документ в окне в противоположном направлении.

На рисунке поясняется также рекомендуемое использование вертикальной полосы прокрутки для просмотра текста. Текущее положение информации в окне относительно документа в целом отображается положением *движка (thumb)*. Кроме щелчков мышью в указанных местах пользователь может также при помощи мыши переместить движок в любое необходимое положение. Использование горизонтальной полосы прокрутки осуществляется аналогичным образом.

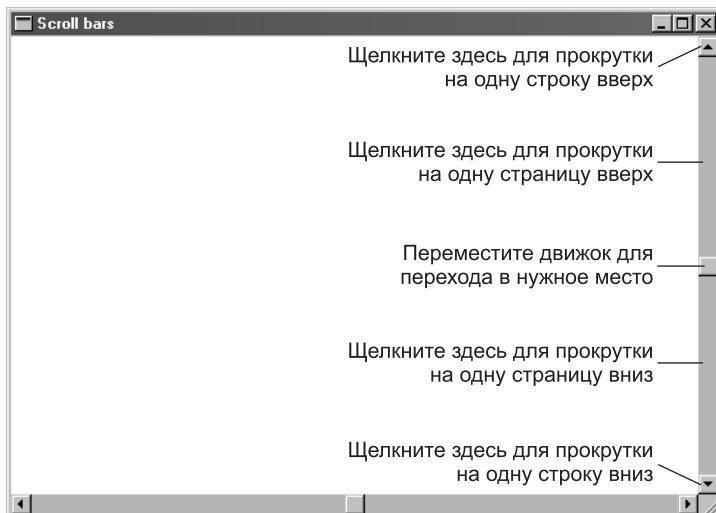


Рис. 2.35. Полосы прокрутки окна

Следует помнить, что, когда у окна появляются полосы прокрутки, клиентская область окна не включает в себя пространство, занятое полосами прокрутки.

Windows обеспечивает всю логику работы мыши с полосами прокрутки. Однако у полос прокрутки нет интерфейса клавиатуры. Если необходимо дублировать некоторые функции полос прокрутки клавишами управления курсором, то следует реализовать эту логику самостоятельно (этот вопрос будет рассмотрен в главе 4).

Параметры полосы прокрутки

Каждая полоса прокрутки характеризуется несколькими параметрами:

- *Диапазон (range)*, задаваемый двумя целыми числами, отражающими минимальное (*nMin*) и максимальное (*nMax*) значения в «единицах данных» проблемной области. При выводе текста «единицами данных» для вертикального измерения обычно считают строки текста, а в горизонтальном измерении¹ — отдельные символы. Величина *range* определяется по формуле $range = nMax - nMin + 1$.
- *Положение (position)* — целое число внутри диапазона, отражающее положение движка. Когда движок находится в крайней верхней или крайней левой позиции на полосе прокрутки, то его положение соответствует минимальному значению диапазона. Крайнее нижнее или крайнее правое положение движка соответствует максимальному значению диапазона.
- *Размер страницы (page)* — целое число, отображающее количество «единиц данных», которые могут разместиться в клиентской области окна при его текущих размерах. Windows использует размер страницы и диапазон полосы прокрутки для управления *длиной движка*. При этом длина движка выглядит пропорционально той части документа, которую пользователь видит в окне. Кроме того, если указан размер страницы, то система берет на себя управление *види-*

¹ При выводе растровых изображений «единицей данных» является 1 пиксель.

мостью полосы прокрутки при изменении размеров окна. Как только размер страницы становится больше диапазона, полоса прокрутки становится невидимой и недоступной.

- По умолчанию для полосы прокрутки установлен диапазон от 0 до 100. Но вы можете переопределить этот диапазон. Так, если отображаемый документ содержит `nLine` строк, то можно использовать два варианта определения диапазона. В первом случае для границ диапазона задаются значения `nMin = 0` и `nMax = nLine - 1`. Во втором случае задаются `nMin = 1` и `nMax = nLine`. Первый вариант считается более предпочтительным, так как он согласуется с индексацией элементов в массиве, а отображаемый текст часто хранится в виде массива строк.

Для установки диапазона полосы прокрутки, размера страницы и текущего положения движка предназначена функция `SetScrollInfo`:

```
int SetScrollInfo(HWND hwnd, int fnBar, LPCSCROLLINFO lpsi, BOOL fRedraw);
```

Параметр `hwnd` содержит либо дескриптор окна, если функция применяется для стандартных полос прокрутки, либо дескриптор элемента управления Scroll Bar. Интерпретация зависит от значения второго параметра — `fnBar`.

Параметр `fnBar` определяет тип полосы прокрутки. Он может принимать значения `SB_VERT` (стандартная вертикальная полоса), `SB_HORZ` (стандартная горизонтальная полоса) и `SB_CTL` (элемент управления Scroll Bar). Параметр `lpsi` содержит указатель на структуру `SCROLLINFO`, которая будет рассматриваться ниже.

Параметр `bRedraw` определяет, должна ли система перерисовать полосу прокрутки сразу после выполнения функции `SetScrollInfo`.

Структура `SCROLLINFO` определена в файле `winuser.h` следующим образом:

```
typedef struct tagSCROLLINFO
{
    UINT cbSize;           // размер структуры в байтах
    UINT fMask;            // маска устанавливаемых/возвращаемых параметров
    int nMin;              // минимальное значение диапазона
    int nMax;              // максимальное значение диапазона
    UINT nPage;            // размер страницы
    int nPos;              // положение движка
    int nTrackPos;         // текущее положение движка, перемещаемого пользователем
} SCROLLINFO;
```

Поле `fMask` задает параметры, которые будут устанавливаться функцией `SetScrollInfo` или возвращаться функцией `GetScrollInfo`. Значением этого поля может быть комбинация флагов, перечисленных в табл. 2.33.

Для получения информации о текущих значениях параметров полосы прокрутки приложение может воспользоваться функцией `GetScrollInfo`.

Важно понимать, что если в программе используются полосы прокрутки, то вместе с Windows вы берете на себя ответственность за их поддержку и обновление положения движка. Так, Windows отвечает за следующие аспекты:

- управление логикой работы мыши с полосой прокрутки;
- обеспечение временной инверсии цвета при щелчке кнопкой мыши в зоне прокрутки на одну страницу;
- перемещение движка, когда пользователь захватывает его мышью и передвигает по полосе;
- отправка сообщений полосы прокрутки в оконную процедуру;

- ❑ переключение видимости полосы прокрутки в зависимости от того, помещается ли документ по своим размерам полностью в окне.

Таблица 2.33. Значения флагов для параметра fMask

Флаг	Описание
SIF_PAGE	Поле nPage содержит размер страницы для пропорциональной полосы прокрутки
SIF_POS	Поле nPos содержит позицию движка, которая не обновляется при перемещении движка пользователем
SIF_RANGE	Поля nMin и nMax содержат минимальное и максимальное значения для диапазона прокрутки
SIF_TRACKPOS	Поле nTrackPos содержит текущую позицию движка, когда пользователь перетаскивает его (поле доступно только для функции GetScrollInfo)
SIF_ALL	Объединение SIF_PAGE SIF_POS SIF_RANGE SIF_TRACKPOS
SIF_DISABLENOSCROLL	Это значение используется только при установке параметров полосы прокрутки. Если новые параметры делают полосу прокрутки ненужной, то вместо ее удаления система делает ее недоступной

Код вашей программы отвечает за следующие аспекты:

- ❑ инициализацию диапазона полосы прокрутки;
- ❑ задание размера страницы для «пропорциональной» полосы прокрутки;
- ❑ обработку сообщения полосы прокрутки;
- ❑ обновление положения движка.

Сообщения полос прокрутки

Windows посыпает оконной процедуре синхронные сообщения WM_VSCROLL и WM_HSCROLL, когда пользователь щелкает мышью на различных зонах вертикальной или горизонтальной полосы прокрутки либо перемещает движок.

Когда оконная процедура получает эти сообщения, параметр wParam содержит в своем младшем слове некоторый код, по которому можно узнать, какое событие произошло. Возможным значениям кода соответствуют определенные идентификаторы, начинающиеся с префикса SB_.

На рис. 2.36 показано, какие коды сообщений вырабатывает Windows при тех или иных действиях пользователя. С каждым действием мыши связаны как минимум два сообщения. Одно сообщение создается при нажатии кнопки мыши, а второе — когда пользователь отпускает ее. Оконная процедура приложения может получить множество сообщений с кодами SB_LINEUP и SB_PAGEUP, если кнопка мыши остается нажатой в соответствующей позиции полосы прокрутки. Сообщение с кодом SB_ENDSCROLL показывает, что кнопка мыши отпущена. Как правило, сообщения SB_ENDSCROLL можно игнорировать.

При перемещении движка (*thumb*) мышью система вырабатывает серию сообщений с кодом SB_THUMBTRACK. Если младшее слово параметра wParam имеет значение SB_THUMBTRACK или SB_THUMBUPOSITION, то старшее слово параметра wParam определяет текущее положение полосы прокрутки. Во всех остальных случаях старшее слово параметра wParam можно не учитывать. Также можно игнорировать параметр lParam, который обычно используется для полос прокрутки, создаваемых в окнах диалога.

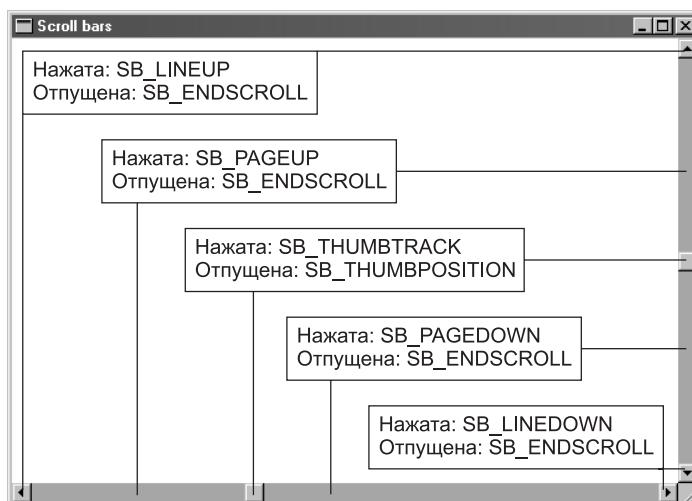


Рис. 2.36. Значения младшего слова параметра wParam для сообщений полосы прокрутки

Получив сообщение от полосы прокрутки, оконная процедура должна его обработать. Для этого сначала определяется приращение текущей позиции: yInc для вертикальной полосы или xInc для горизонтальной полосы. Приращение выражено в «единицах данных». Оно может быть как положительным, так и отрицательным. Затем для прокрутки содержимого окна вызывается функция ScrollWindow:

```
BOOL ScrollWindow(
    HWND hWnd,           // дескриптор окна
    int XAmount,         // горизонтальный скроллинг
    int YAmount,         // вертикальный скроллинг
    CONST RECT* lpRect, // указатель на прямоугольник клиентской области
    CONST RECT* lpClipRect // указатель на прямоугольник отсечения
);
```

Параметры XAmount и YAmount задают величину прокрутки в пикселях. Параметры lpRect и lpClipRect чаще всего получают значение NULL, что означает прокрутку для всей клиентской области.

Таким образом, обработка сообщения WM_VSCROLL сопровождается следующим вызовом:

```
ScrollWindow(hWnd, 0, -yStep * yInc, NULL, NULL);
```

Обработка сообщения WM_HSCROLL реализуется следующим вызовом:

```
ScrollWindow(hWnd, -cxChar * xInc, 0, NULL, NULL);
```

В указанных вызовах величина yStep задает шаг между строками в пикселях, а величина cxChar равна средней ширине символов в пикселях. Знак «минус» перед приращением отражает тот факт, что фактическое перемещение документа в окне противоположно направлению прокрутки с точки зрения пользователя.

После прокрутки содержимого окна вы должны позаботиться об обновлении позиции движка на полосе прокрутки. Например, для вертикальной полосы могут быть использованы следующие инструкции:

```
si.nPos += yInc;
SetScrollInfo(hWnd, SB_VERT, &si, TRUE);
```

где si — переменная типа SCROLLINFO.

Заметим, что в результате выполнения функции `ScrollWindow` часть клиентской области освобождается от информации. Для того чтобы обеспечить вывод в освободившуюся зону новой порции текстового документа, приложение обычно вызывает инструкции

```
InvalidateRect(hWnd, NULL, TRUE);
UpdateWindow(hWnd);
```

Напомним, что функция `UpdateWindow` вызывает передачу сообщения `WM_PAINT` непосредственно оконной процедуре. Это важно, если вы хотите обеспечить немедленную реакцию приложения на действия пользователя, когда тот быстро перемещает движок полосы прокрутки (сообщение `SB_THUMBTRACK`). Дело в том, что при обычном порядке обработки функция `InvalidateRect` вызывает постановку сообщения `WM_PAINT` в очередь приложения, а там это сообщение обрабатывается с самым низким приоритетом.

Наконец, в блоке обработки сообщения `WM_PAINT` вы должны предусмотреть код для вывода той части текстового документа, которая определяется текущим положением вертикального и горизонтального движков.

Пример работы с полосами прокрутки приводится в листинге 2.2.

Примеры программ

Просмотрщик текстовых файлов

Приложение `TextViewer` предназначено для чтения и вывода на экран текстового файла. В приложении демонстрируются: использование полос прокрутки, вывод текста при помощи функций `TextOut` и `TabbedTextOut`, а также использование регистра отсечения.

В этом примере также показано, что использование идей ООП вряд ли ухудшает качество программного кода, скорее — наоборот. С этой целью разработан класс `KDocument` для решения тех подзадач, которые связаны с загрузкой, хранением и выводом документа в клиентскую область окна с учетом его прокрутки. Интерфейс класса содержится в файле `KDocument.h`, а его реализация — в файле `KDocument.cpp`.

Программа представляет собой многофайловый проект, приведенный в листинге 2.2¹.

Листинг 2.2. Проект `TextViewer`

```
///////////////////////////////
// KDocument.h
#include <windows.h>
#include <vector>
#include <string>
using namespace std;

class KDocument {
```

¹ Как всегда, не забудьте скопировать в каталог проекта файлы `KWnd.h` и `KWnd.cpp` (из листинга 1.2 в главе 1) и включить их в состав проекта при помощи окна `Project Workspace` интегрированной среды.

```
public:
    BOOL Open(const char* file);
    void Initialize(LPTEXTMETRIC tm);
    void ScrollSettings(HWND hwnd, int width, int height);
    void UpdateHscroll(HWND hwnd, int xInc);
    void UpdateVscroll(HWND hwnd, int yInc);
    void PutText(HWND hwnd, HDC hdc);

    int cxChar;           // средняя ширина символа
    int yStep;            // высота (шаг) строки
    int lineLenMax;       // максимальная длина строки
    SCROLLINFO vsi;       // вертикальный скроллинг
    int vertRange;        // диапазон вертикальной полосы прокрутки
    SCROLLINFO hsi;       // горизонтальный скроллинг
    int horzRange;        // диапазон горизонтальной полосы прокрутки

private:
    vector<string> lines; // вектор для хранения строк документа
};

////////////////////////////////////////////////////////////////
// KDocument.cpp
#include <windows.h>
#include <fstream>
#include "KDocument.h"

BOOL KDocument::Open(const char* file) {
    ifstream finp(file);
    char buf[200];

    if(!finp.good()) {
        MessageBox(NULL, "Не найден входной файл", "Error", MB_OK);
        return FALSE;
    }
    // Прочитать файл, сохранив информацию в векторе строк lines
    while(!finp.eof()) {
        finp.getline(buf, 199);
        buf[199] = 0;
        lines.push_back(string(buf));
    }
    // Вычислить максимальную длину строки
    lineLenMax = 0;
    for (int i = 0; i < lines.size(); ++i) {
        int lineLen = lines[i].size();
        // Корректировка, если строка содержит символы табуляции
        int iTabPos = 0;
        while (1) {
            iTabPos = lines[i].find('\t', iTabPos);
            if (iTabPos != -1) {
                lineLen += 8;
                iTabPos++;
            }
            else break;
        }
        if (lineLen > lineLenMax) lineLenMax = lineLen;
    }
    return TRUE;
}

void KDocument::Initialize(LPTEXTMETRIC tm) {
```

продолжение ↗

Листинг 2.2 (продолжение)

```

cxChar = tm->tmAveCharWidth;
yStep = tm->tmHeight + tm->tmExternalLeading;
vsi.nMin = vsi.nPos = 0;
hsi.nMin = hsi.nPos = 0;
}

void KDocument::ScrollSettings(HWND hwnd, int width, int height) {
    // Вертикальный скроллинг
    vsi.cbSize = sizeof(vsi);
    vsi.fMask = SIF_RANGE | SIF_PAGE | SIF_POS;
    vsi.nPage = height / yStep - 1;
    vsi.nMax = lines.size() - 1;
    if (vsi.nPage > vsi.nMax)
        vsi.nPos = vsi.nMin;
    vertRange = vsi.nMax - vsi.nMin + 1;
    SetScrollInfo(hwnd, SB_VERT, &vsi, TRUE);

    // Горизонтальный скроллинг
    hsi.cbSize = sizeof(SCROLLINFO);
    hsi.fMask = SIF_RANGE | SIF_PAGE | SIF_POS;
    hsi.nPage = width/cxChar - 2;
    hsi.nMax = lineLenMax;
    if (hsi.nPage > hsi.nMax)
        hsi.nPos = hsi.nMin;
    horzRange = hsi.nMax - hsi.nMin + 1;
    SetScrollInfo(hwnd, SB_HORZ, &hsi, TRUE);
}

void KDocument::UpdateVscroll(HWND hwnd, int yInc) {
    // ограничение на положительное приращение
    yInc = min(yInc, vertRange - (int)vsi.nPage - vsi.nPos);
    // ограничение на отрицательное приращение
    yInc = max(yInc, vsi.nMin - vsi.nPos);

    if (yInc) {
        ScrollWindow(hwnd, 0, -yStep * yInc, NULL, NULL);
        vsi.nPos += yInc;
        SetScrollInfo(hwnd, SB_VERT, &vsi, TRUE);
        InvalidateRect(hwnd, NULL, TRUE);
        UpdateWindow (hwnd);
    }
}

void KDocument::UpdateHscroll(HWND hwnd, int xInc) {
    // ограничение на положительное приращение
    xInc = min(xInc, horzRange - (int)hsi.nPage - hsi.nPos);
    // ограничение на отрицательное приращение
    xInc = max(xInc, hsi.nMin - hsi.nPos);

    if (xInc) {
        ScrollWindow(hwnd, -cxChar * xInc, 0, NULL, NULL);
        hsi.nPos += xInc;
        SetScrollInfo(hwnd, SB_HORZ, &hsi, TRUE);
        InvalidateRect(hwnd, NULL, TRUE);
        UpdateWindow (hwnd);
    }
}

```

```
}

void KDocument::PutText(HWND hwnd, HDC hdc) {
    RECT rect;
    GetClientRect(hwnd, &rect);
    rect.left += cxChar;
    rect.right -= cxChar;
    HRGN hRgn = CreateRectRgnIndirect(&rect);
    SelectClipRgn(hdc, hRgn);

    int x = cxChar * (hsi.nMin - hsi.nPos + 1);
    int y = yStep;
    int amountLines = lines.size();
    int iBeg = vsi.nPos;
    int iEnd = (vsi.nPos+vsi.nPage < amountLines)? vsi.nPos+vsi.nPage :
        amountLines;

    for (int i = iBeg; i < iEnd; ++i) {
        int iTabPos = lines[i].find('\t');
        if (-1 == iTabPos)
            TextOut(hdc, x, y, lines[i].c_str(), lines[i].size());
        else
            TabbedTextOut(hdc, x, y, lines[i].c_str(), lines[i].size(), 0, 0, x);
        y += yStep;
    }
    SelectClipRgn(hdc, NULL);
}
////////////////////////////////////////////////////////////////
// TextViewer.cpp
#include <windows.h>
#include "KWnd.h"
#include "KDocument.h"

#define FILE_NAME "D:\\Program files\\Microsoft Visual
Studio\\VC98\\MFC\\SRC\\README.TXT"
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

KDocument doc;
//=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    if (!doc.Open(FILE_NAME)) return 0;

    KWnd mainWnd("Text Viewer", hInstance, nCmdShow, WndProc, NULL,
        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, CS_HREDRAW | CS_VREDRAW,
        WS_OVERLAPPEDWINDOW | WS_VSCROLL | WS_HSCROLL);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}
//=====
```

Листинг 2.2 (продолжение)

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    TEXTMETRIC tm;
    int cxClient=0, cyClient=0;
    static int xInc, yInc;

    switch (uMsg)
    {
        case WM_CREATE:
            hDC = GetDC(hWnd);
            GetTextMetrics(hDC, &tm);
            doc.Initialize(&tm);
            ReleaseDC(hWnd, hDC);
            break;

        case WM_SIZE:
            hDC = GetDC(hWnd);
            cxClient = LOWORD(lParam);
            cyClient = HIWORD(lParam);
            if (cxClient > 0)
                doc.ScrollSettings(hWnd, cxClient, cyClient);
            ReleaseDC(hWnd, hDC);
            break;

        case WM_VSCROLL:
            switch(LOWORD(wParam)) {
                case SB_LINEUP:
                    yInc = -1; break;
                case SB_LINEDOWN:
                    yInc = 1; break;
                case SB_PAGEUP:
                    yInc = -(int)doc.vsi.nPage; break;
                case SB_PAGEDOWN:
                    yInc = (int)doc.vsi.nPage; break;
                case SB_THUMBTRACK:
                    yInc = HIWORD(wParam) - doc.vsi.nPos; break;
                default: yInc = 0;
            }
            doc.UpdateVscroll(hWnd, yInc);
            break;

        case WM_HSCROLL:
            switch(LOWORD(wParam)) {
                case SB_LINELEFT:
                    xInc = -1; break;
                case SB_LINERIGHT:
                    xInc = 1; break;
                case SB_PAGELEFT:
                    xInc = -0.8 * (int)doc.hsi.nPage; break;
                case SB_PAGERIGHT:
                    xInc = 0.8 * (int)doc.hsi.nPage; break;
                case SB_THUMBTRACK:
                    xInc = HIWORD(wParam) - doc.hsi.nPos; break;
                default: xInc = 0;
            }
    }
}
```

```
doc.UpdateHscroll(hWnd, xInc);
break;

case WM_PAINT:
hDC = BeginPaint(hWnd, &ps);

doc.PutText(hWnd, hDC);

EndPaint(hWnd, &ps);
break;

case WM_DESTROY:
PostQuitMessage(0);
break;

default:
return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
////////////////////////////////////////////////////////////////
```

Обратите внимание на следующие особенности реализации класса KDocument:

- Для хранения текстового документа используется член класса `lines`, который объявлен как вектор строк типа `string`.
- В методе `Open` заданный входной файл читается по строкам. Предполагается, что строки разделяются символом `\n`. Каждая прочитанная строка добавляется в конец вектора `lines`. Затем в цикле `for` вычисляется максимальная длина строки `lineLenMax` с учетом возможного присутствия символов табуляции. Присутствие символа табуляции в строке `lines[i]` проверяется методом `find` класса `string`, который возвращает позицию обнаруженного символа `\t` или значение `-1`, если символ табуляции не обнаружен. В методе `ScrollSettings` величина `lineLenMax` используется для установки максимального значения диапазона горизонтальной полосы прокрутки.
- В методе `Initialize` вычисляются размеры «единиц данных» для скроллинга. Величина `cxChar` задает шаг изменения в горизонтальном измерении, а величина `yStep` — шаг изменения в вертикальном измерении. Для вычислений используются метрики текста, полученные в параметре `tm`.
- Метод `ScrollSettings` предназначен для установки параметров вертикальной и горизонтальной полос прокрутки при заданных ширине `width` и высоте `height` клиентской области окна.
- Метод `UpdateVscroll` выполняет итоговую обработку всех сообщений от вертикальной полосы прокрутки. В нем же производятся прокрутка содержимого окна при помощи функции `ScrollWindow`, обновление положения движка с помощью `SetScrollInfo` и обновление клиентской области при помощи функции `UpdateWindow`. Более подробные комментарии по этому этапу можно найти в разделе «Полосы прокрутки и вывод текста».

Обратите особое внимание на первые две инструкции в теле функции. Они предваряются комментариями «ограничение на положительное приращение» и «ограничение на отрицательное приращение». Первая инструкция обеспече-

чивает прекращение прокрутки при продвижении документа к его концу, как только позиция движка достигнет значения `vertRange - (int)vsi.nPage`. В этом положении движка последняя страница документа полностью видна в окне и дальнейшая прокрутка теряет смысл. Заметим, что преобразование к типу `int` для поля `vsi.nPage` необходимо для корректного выполнения макроса `min`.

Вторая инструкция обеспечивает прекращение прокрутки при продвижении документа к его началу, как только позиция движка достигнет значения `vsi.nMin`.

- Метод `UpdateHscroll` производит итоговую обработку всех сообщений от горизонтальной полосы прокрутки. Реализация метода аналогична реализации метода `UpdateVscroll`.
- Метод `PutText` предназначен для вывода фрагмента текста из документа в клиентскую область окна.

Для создания полей слева и справа от видимого в окне текста (в данном примере шириной в один символ) используется регион отсечения `hRgn`. Он создается вызовом функции `CreateRectRgnIndirect`, причем в качестве аргумента передается адрес прямоугольника `rect`, который в вертикальном измерении совпадает с прямоугольником клиентской области, а в горизонтальном измерении уменьшен слева и справа на величину `cxChar`. Созданный регион выбирается в контекст устройства функцией `SelectClipRgn`.

Фрагмент текста выводится в цикле `for` в соответствии с текущим положением движка `vsi.nPos` на вертикальной полосе прокрутки и текущим положением движка `hs1.nPos` на горизонтальной полосе прокрутки. Если в текущей строке `lines[i]` нет символов табуляции, то вызывается функция `TextOut`. В противном случае используется функция `TabbedTextOut`.

По умолчанию в контексте устройства установлен режим отображения `MM_TEXT`, поэтому начало координат совмещено с левым верхним углом клиентской области. Координата `y` для вывода каждой следующей строки вычисляется с учетом смещения `yStep`.

Если значение `hs1.nPos` совпадает со значением `hs1.nMin`, то координата `x`, вычисляемая по формуле `cxChar × (hs1.nMin - hs1.nPos + 1)`, имеет значение `cxChar`. В этом случае строка выводится в окно, начиная с первого символа. А поскольку в контексте устройства выбран регион отсечения с дескриптором `hRgn`, то первый видимый символ в окне нельзя выводить левее, чем `x = cxChar`.

В случае `hs1.nPos > 0` координата `x` будет принимать сначала нулевое, а потом отрицательные значения, абсолютная величина которых возрастает по мере прокрутки документа по горизонтали вправо. То есть начало вывода строки приходится на точку экрана, лежащую слева за пределами окна приложения. Разумеется, за счет отсечения, за которым следует Windows, пользователь будет видеть только ту часть строки, которая попадает в клиентскую область окна. А еще точнее, только ту часть строки, которая попадает в регион отсечения.

Перед выходом из функции инструкция `SelectClipRgn(hdc, NULL)` удаляет из контекста устройства выбранный ранее регион отсечения.

Теперь следует рассмотреть особенности кода, приведенного в файле `TextViewer.cpp`:

- Для работы с документом объявлен глобальный объект `doc` класса `KDocument`.
- В функции `WinMain` вызывается метод `Open` объекта `doc`, выполняющий загрузку документа из файла, имя которого задается макросом `FILE_NAME`. В данном случае используется файл `README.TXT`, входящий в состав Microsoft Visual C++ 6.0¹. Конечно, в этом макросе вы можете определить путь к любому другому текстовому файлу.
- Обратите внимание на флаги `WS_VSCROLL | WS_HSCROLL`, переданные конструктору класса `KWnd` в составе параметра `windowStyle`. Именно они обеспечивают появление у главного окна приложения вертикальной и горизонтальной полос прокрутки.
- В оконной процедуре `WndProc` метод `Initialize` объекта `doc` вызывается при обработке сообщения `WM_CREATE`, а метод `ScrollSettings` — при обработке сообщения `WM_SIZE`.
- Обработка сообщения `WM_VSCROLL` завершается вызовом метода `doc.UpdateVscroll(hWnd, yInc)`. А обработка сообщения `WM_HSCROLL` завершается вызовом метода `doc.UpdateHscroll(hWnd, xInc)`.
- Сам вывод текста происходит в блоке обработки сообщения `WM_PAINT` при помощи метода `doc.PutText(hWnd, hDC)`.

Выполните компиляцию этого проекта и поэкспериментируйте с изменениями размеров окна приложения и прокруткой текста в окне.

Вывод временной диаграммы напряжения переменного электрического тока

Приложение `U_T_Diagram` демонстрирует использование страничной системы координат, создание логических перьев, рисование различных линий, создание логических шрифтов, вывод текста с применением разных шрифтов.

В программе решается задача формирования и вывода на экран временной диаграммы напряжения переменного электрического тока. Действующее значение напряжения — 220 В, частота — 50 Гц. Диаграмма строится для промежутка времени, в котором происходят ровно два периода колебаний.

На рис. 2.37 показано окно приложения `U_T_Diagram` после его запуска. К сожалению, черно-белый рисунок трансформирует различные цвета в оттенки серого и поэтому не передает всей красоты созданной диаграммы. Например, линии координатной сетки на самом деле имеют зеленоватый цвет, а синусоида графика — малиновый цвет. Если вы хотите все-таки насладиться этой красотой, то придется ввести в компьютер прилагаемый ниже текст программы, откомпилировать и запустить ее.

Период синусоидальных колебаний T связан с их частотой f соотношением $T = 1/f$, поэтому для частоты 50 Гц период составляет 0,02 с, или 20 мс. Также полезно вспомнить, что действующее значение напряжения переменного тока U связано с его амплитудным значением U_m соотношением $U = 0,707 \cdot U_m$. Из этого следует, что амплитудное значение напряжения равно 311,17 В.

¹ На вашем компьютере путь к этому файлу может быть другим.

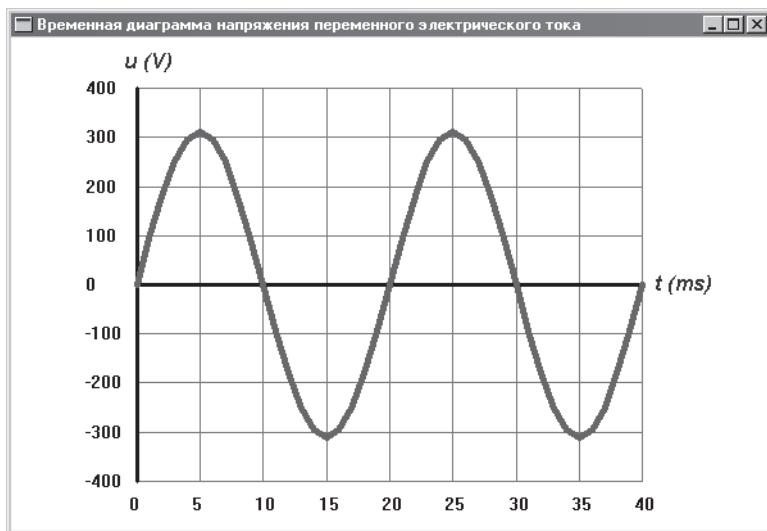


Рис. 2.37. Окно приложения U_T_Diagram

Для улучшения внешнего вида графика используются не только разные цвета линий, но и разная толщина. Линии координатной сетки рисуются толщиной 1 пиксел, оси X и Y — толщиной 3 пикселя, а сама синусоида — толщиной 5 пикселов.

Прежде чем привести листинг программы, сделаем некоторые пояснения по поводу выбора и настройки координатной системы. Размышления над сутью решаемой задачи приводят нас к убеждению, что здесь можно обойтись без аффинных преобразований и, следовательно, нас вполне устроит *страничная система координат*.

Физическую область вывода (viewport) стоит привязать к клиентской области окна, размеры которой можно получить при помощи функции `GetClientRect`. Благодаря этому рисунок будет автоматически масштабироваться при изменении размеров основного окна приложения.

Теперь подумаем, как нам организовать *логическую область вывода (window)*. Казалось бы, здесь удобно использовать логические единицы, присущие данной проблемной области, то есть по оси абсцисс — миллисекунды, а по оси ординат — вольты. Но если выбрать эти логические единицы, то возникнет очень неприятная проблема. Фактически, мы потеряем контроль над толщиной пера, используемого для прорисовки линий. Это раз. Кроме того, толщина пера окажется разной при рисовании по горизонтали и по вертикали, так как она будет определяться масштабными коэффициентами mx и my ¹. Это два.

Причиной перечисленных неприятностей являются особенности реализации в Win32 API объекта логического пера. Дело в том, что в вызове функции `CreatePen(penStyle, nWidth, crColor)` параметр `nWidth` задает толщину пера именно в логических единицах. Исключением является нулевое значение `nWidth`,

¹ $mx = xVE / xWE$, $my = yVE / yWE$, где xVE , yVE — экстенты физической области вывода, а xWE , yWE — экстенты логической области вывода.

определяющее перо толщиной 1 пиксел независимо от установленных координатных преобразований.

Например, если для физической области вывода размеры экстентов равны $xVE = 952$, $yVE = 702^1$, а для логической области будут выбраны значения экстентов $xWE = 60$ (мс), $yWE = 1000$ (В), то логическая единица по горизонтали будет соответствовать $mx = 952 / 60 = 15,86$ или, округленно, 16 пикселям. Логическая единица по вертикали будет соответствовать $my = 702 / 1000 = 0,72$ или, округленно, одному пикселу.

Чтобы все-таки нарисовать график так, как это показано на рис. 2.37, будем придерживаться следующего плана действий. Установим режим отображения **MM_ISOTROPIC**, а значения экстентов для логической области вывода сделаем совпадающими со значениями экстентов для физической области вывода. В этом случае будет получен полный контроль над толщиной пера. Проблему же масштабирования придется решать «вручную», используя соответствующие масштабные коэффициенты:

```
nPixPerVolt = yVE / 1000.0;  
nPixPerMs = xVE / 60.0;
```

Код проекта приведен в листинге 2.3.

Листинг 2.3. Проект U_T_Diagram²

```
////////////////////////////////////////////////////////////////////////  
// U_T_Diagram.cpp  
#include <windows.h>  
#include <math.h>  
#include "KWnd.h"  
  
HRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);  
void DrawDiagram(HWND, HDC);  
  
#define Pi 3.14159265  
  
=====  
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,  
    LPSTR lpCmdLine, int nCmdShow)  
{  
    MSG msg;  
    KWnd mainWnd("Временная диаграмма напряжения переменного электрического тока",  
        hInstance, nCmdShow, WndProc);  
  
    while (GetMessage(&msg, NULL, 0, 0)) {  
        TranslateMessage(&msg);  
        DispatchMessage(&msg);  
    }  
    return msg.wParam;  
}  
  
=====
```

продолжение ↗

¹ Эти значения являются шириной и высотой клиентской области окна, созданного с размерами по умолчанию.

² Не забудьте добавить к проекту файлы **KWnd.h** и **KWnd.cpp** (см. листинг 1.2 в главе 1).

Листинг 2.3 (продолжение)

```

LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;

    switch (uMsg)
    {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &ps);

        DrawDiagram(hWnd, hDC);

        EndPaint(hWnd, &ps);
        break;

    case WM_DESTROY:
        PostQuitMessage(0);
        break;

    default:
        return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }
    return 0;
}

//=====
void DrawDiagram(HWND hwnd, HDC hdc)
{
    RECT rect;
    GetClientRect(hwnd, &rect);
    const int xVE = rect.right - rect.left;
    const int yVE = rect.bottom - rect.top;
    const int xWE = xVE;
    const int yWE = yVE;
    double nPixPerVolt = yVE / 1000.0;
    double nPixPerMs = xVE / 60.0;      // n Pixels Per Millisec

    SetMapMode (hdc, MM_ISOTROPIC);
    SetWindowExtEx (hdc, xWE, yWE, NULL);
    SetViewportExtEx (hdc, xVE, -yVE, NULL);
    SetViewportOrgEx (hdc, 10*nPixPerMs, yVE/2, NULL);

    const int tMin = 0;          // ms
    const int tMax = 40;         // ms
    const int uMin = -400;       // V
    const int uMax = 400;        // V
    const int tGridStep = 5;
    const int uGridStep = 100;
    int x, y;

    char* xMark[] = { "0", "5", "10", "15", "20", "25", "30", "35", "40" };
    char* yMark[] = { "-400", "-300", "-200", "-100", "0", "100", "200", "300",
                     "400" };

    // Сетка
    HPEN hPen0 = CreatePen(PS_SOLID, 1, RGB(0, 160, 0));
    HPEN hOldPen = (HPEN)SelectObject(hdc, hPen0);
    int u = uMin;

```

```
int xMin = tMin * nPixPerMs;
int xMax = tMax * nPixPerMs;
for (int i = 0; i < 9; ++i) {
    y = u * nPixPerVolt;
    MoveToEx(hdc, xMin, y, NULL);
    LineTo(hdc, xMax, y);
    TextOut(hdc, xMin-40, y+8, yMark[i], strlen(yMark[i]));
    u += uGridStep;
}

int t = tMin;
int yMin = uMin * nPixPerVolt;
int yMax = uMax * nPixPerVolt;
for (i = 0; i < 9; ++i) {
    x = t * nPixPerMs;
    MoveToEx(hdc, x, yMin, NULL);
    LineTo(hdc, x, yMax);
    TextOut(hdc, x-6, yMin-10, xMark[i], strlen(xMark[i]));
    t += tGridStep;
}

// Ось "x"
HPEN hPen1 = CreatePen(PS_SOLID, 3, RGB(0, 0, 0));
SelectObject(hdc, hPen1);
MoveToEx(hdc, 0, 0, NULL); LineTo(hdc, xMax, 0);

static LOGFONT lf;
lf.lfPitchAndFamily = FIXED_PITCH | FF_MODERN;
lf.lfItalic = TRUE;
lf.lfWeight = FW_BOLD;
lf.lfHeight = 20;
lf.lfCharSet = DEFAULT_CHARSET;
lstrcpy( (LPSTR)&lf.lfFaceName, "Arial" );

HFONT hFont0 = CreateFontIndirect(&lf);
HFONT hOldFont = (HFONT)SelectObject(hdc, hFont0);
SetTextColor(hdc, RGB(0, 0, 200));

TextOut(hdc, xMax+10, 10, "t (ms)", 6);
// Ось "y"
MoveToEx(hdc, 0, yMin, NULL); LineTo(hdc, 0, yMax);
TextOut(hdc, -10, yMax+30, "u (V)", 5);

// График
HPEN hPen2 = CreatePen(PS_SOLID, 5, RGB(200, 0, 100));
SelectObject(hdc, hPen2);
int tStep = 1;
double radianPerMs = 2 * Pi / 20;
const double uAmplit = 311.17;      // volt
t = tMin;
MoveToEx(hdc, 0, 0, NULL);
while (t <= tMax) {
    u = uAmplit * sin( t * radianPerMs);
    LineTo(hdc, t * nPixPerMs, u * nPixPerVolt);
    t += tStep;
}

// Заголовок
char* title = "Диаграмма напряжения переменного электр. тока";
```

продолжение ↗

Листинг 2.3 (продолжение)

```

lf.lfItalic = FALSE;
lf.lfWeight = FW_BOLD;
lf.lfHeight = 30;
HFONT hFont1 = CreateFontIndirect(&lf);
SelectObject(hdc, hFont1);
SetTextColor(hdc, RGB(0, 200, 0));
TextOut(hdc, 0, yMax + 70, title, strlen(title));

SelectObject(hdc, hOldPen);
SelectObject(hdc, hOldFont);
}
/////////////////////////////////////////////////////////////////

```

Основная работа в приведенной программе сосредоточена в функции `DrawDiagram`, которая вызывается в блоке обработки сообщения `WM_PAINT`.

Обратите внимание на то, как реализована настройка страничной системы координат. В выбранной нами стратегии решения логические единицы совпадают с физическими единицами, поэтому введем новый термин — *металогические единицы*, которые отражают проблемную область. Одна металогическая единица по горизонтали соответствует 1 мс, а одна металогическая единица по вертикали соответствует 1 В.

Заметим, что вызов функции

```
SetViewportExtEx (hdc, xVE, -yVE, NULL);
```

реализует преобразование *отражения*, благодаря которому значения по оси ординат возрастают снизу вверх, как это принято в проблемной области.

Другой вызов:

```
SetViewportOrgEx (hdc, 10*nPixPerMs, yVE/2, NULL);
```

обеспечивает установку начала координат строго посередине по вертикали и с отступом на 10 металогических единиц слева по горизонтали.

В программе используются металогические величины `tMin`, `tMax`, `uMin`, `uMax`, `tGridStep`, `uGridStep`, являющиеся константами. Также используются и металогические переменные `t`, `u`. Переход к соответствующим логическим (и совпадающим с ними физическим) величинам осуществляется через коэффициенты `nPixPerMs`, `nPixPerVolt`.

Для построения графика в цикле `while` применяется кусочно-линейная аппроксимация функции `sin(x)`. После очередного приращения аргумента вычисляется значение функции, по которому вычисляется текущее значение `u`. Затем при помощи функции `LineTo` проводится отрезок линии, соединяющий предыдущее и текущее значения `u`.

Так как функция `sin` требует в качестве аргумента значение угла в радианах, то используется понятие угловой частоты $\omega = 2 \cdot \pi \cdot f = 2 \cdot \pi / T$ для вычисления соответствующего масштабного коэффициента:

```
radianPerMs = 2 * Pi / 20;
```

После компиляции и запуска на выполнение можно поэкспериментировать с окном программы, изменяя его размеры, чтобы увидеть, как будет масштабироваться изображение.

3

GDI. Палитры, растры, метафайлы

В третьей главе будет продолжено рассмотрение графического интерфейса устройства — GDI. На очереди рассмотрение таких графических объектов, как палитры, битовые образы, или растры, и метафайлы. Даже если вы не планируете использовать палитры в создаваемых приложениях, беглое знакомство с ними может оказаться полезным, так как понятие цветовых таблиц используется в форматах DIB-растров и DIB-секций.

Палитры

Если устройство вывода поддерживает полный диапазон цветов, определенных 24-битным RGB-значением¹, то палитра вам не нужна. К сожалению, еще не вышли из употребления старые модели дисплеев, которые поддерживают только 256 цветов. Другие графические устройства, такие как принтеры и плоттеры, тоже имеют ограничения по количеству воспроизводимых цветов.

Графические объекты, именуемые в Windows *палитрами*, обеспечивают надлежащее согласование цветов при работе приложений на разных аппаратных платформах. Чтобы узнать, поддерживает ли конкретное устройство работу с палитрой, нужно вызвать функцию `GetDeviceCaps`, передав значение индекса `RASTERCAPS`, и убедиться, что возвращаемое значение содержит флагок `RC_PALETTE`.

Для упрощения изложения в следующих разделах, посвященных палитре, под устройством вывода будет подразумеваться дисплей.

Основные принципы управления палитрами

Цветовая палитра представляет собой таблицу (массив) цветов, которые способно воспроизвести устройство. *Вход в таблицу* для доступа к каждому цвету осуществляется по индексу.

В Win32 GDI используется несколько типов палитр.

Системная палитра (*system palette*) определяет все цвета, которые могут быть одновременно отображены устройством. Диапазон цветов зависит от аппаратной

¹ 24-битное значение позволяет поддерживать 16 777 216 цветов.

палитры, реализуемой видеоадаптером, и от настроек экрана. В любом случае системная палитра является неким «программным представителем» аппаратной палитры. Приложение не может непосредственно модифицировать системную палитру. Системная палитра обычно содержит 256 элементов (входов), из которых 20 зарезервировано для так называемых *статических цветов*, которые никогда не изменяются. Эти цвета используются всеми приложениями для вывода меню, кнопок, фона окна, текстовых надписей и других элементов стандартного интерфейса.

Логическая палитра (logical palette) — это палитра уровня приложения. Она создается приложением и закрепляется за контекстом устройства. Логическая палитра позволяет определять и использовать цвета, которые соответствуют потребностям приложения. После создания логической палитры с помощью функции `CreatePalette` или `CreateHalftonePalette` она должна быть *реализована* вызовом функции `RealizePalette`. В процессе реализации Windows заполняет неиспользуемые элементы в системной палитре цветами из логической палитры. Если неиспользованных элементов меньше, чем нужно, то Windows распределяет оставшиеся цвета из логической палитры, либо подыскивая ближайший цвет в аппаратной палитре, либо определяя подходящее *смещение (dithering)* цветов, если применяются сплошные кисти.

Для успешного применения логической палитры нужно учесть еще один нюанс, о котором в MSDN практически ничего не говорится. Во всех функциях, требующих передачи аргумента типа `COLORREF`, соответствующее значение должно быть определено с помощью макроса `PALETTERGB`, а не при помощи макроса `RGB`.

Если приложение явно не создает логическую палитру, то используется *палитра по умолчанию (default palette)*, содержащая только те 20 цветов, которые имеются в системной палитре. В этом случае для аппроксимации цвета, отсутствующего в палитре, будут использоваться указанные 20 цветов.

Чтобы обеспечить одновременную работу нескольких приложений, использующих различные логические палитры, Windows предоставляет активному окну¹ приоритет в установлении цветов для логической палитры. Неактивным окнам приходится пользоваться оставшимися цветами. Неактивные окна используют все незадействованные элементы в логической палитре и применяют ближайшие согласованные цвета для всех невыполненных запросов к палитре. Однако эта проблема обычно не очень существенна, поскольку в большинстве приложений используются только системные цвета.

Для координации функционирования активных и неактивных окон при работе с палитрами, прежде чем предоставить фокус ввода приложению, которое использует логическую палитру, Windows отправляет ему сообщение `WM_QUERYNEWPALETTE`. Это сообщение дает возможность приложению еще раз реализовать палитру и восстановить цвета, которые могли быть изменены другими приложениями, пока окно было неактивным.

Сообщение `WM_PALETTECHANGED` отправляется всем окнам, когда одно из приложений реализует свою логическую палитру. Для неактивных окон это сообщение несет информацию о том, что некоторые цвета в палитре могли быть изменены другим окном, имеющим фокус ввода.

¹ Окну, имеющему фокус ввода.

Системная палитра

Системная палитра является графическим объектом уровня операционной системы. Ранее уже говорилось о том, что диапазон цветов, которые одновременно воспроизводятся устройством, зависит как от аппаратной палитры, реализуемой видеoadаптером, так и от настроек экрана.

Чтобы посмотреть или изменить настройки экрана, нужно щелкнуть правой кнопкой мыши на свободной части рабочего стола экрана, после чего появится всплывающее меню. В этом меню следует выполнить команду **Свойства**, что приведет к отображению диалогового окна **Свойства: Экран**. В окне нужно выбрать вкладку **Настройка** и найти на ней групповое поле **Цветовая палитра**. Элемент управления типа **COMBOBOX** внутри этого поля позволяет выбирать количество одновременно отображаемых цветов на экране. Если у вас достаточно современный дисплей, то доступными, скорее всего, окажутся следующие установки:

- 256 цветов;
- Hi Color (16 бит);
- True Color (24 или 32 бит).

В 256-цветном режиме каждый пиксель представлен в кадровом буфере видеoadаптера одним байтом. Один байт позволяет закодировать до 256 разных цветов, одновременно отображаемых на экране. Точный состав цветов определяется палитрой видеoadаптера, которая предоставляется пользовательским приложениям в виде системной палитры.

Системная палитра представляет собой таблицу из 256 структур типа **PALETTEENTRY**:

```
typedef struct {
    BYTE peRed;
    BYTE peGreen;
    BYTE peBlue;
    BYTE peFlags;
} PALETTEENTRY;
```

Структура **PALETTEENTRY** определяет цвет конкретного элемента в массиве по его компонентам **RGB**. Поле **peFlags** используется при создании логических палитр.

Приложение не имеет непосредственного доступа к системной палитре. Однако оно может получить текущее состояние системной палитры при помощи функции **GetSystemPaletteEntries**, имеющей следующий прототип:

```
UINT GetSystemPaletteEntries(
    HDC hdc,           // дескриптор контекста устройства
    UINT iStartIndex,  // начальный вход в таблицу
    UINT nEntries,     // количество входов
    LPPALETTEENTRY lpre // указатель на массив структур PALETTEENTRY
);
```

Параметр **lpre** задает адрес массива, в который помещается информация о палитре. Если этот параметр равен **NULL**, то функция возвращает общее количество цветов в системной палитре.

Системную палитру можно условно разделить на две секции. Первая секция содержит фиксированные, или *статические*, цвета, а вторая — цвета, которые могут изменяться приложением при помощи логической палитры.

По умолчанию в системной палитре определена группа из 20 статических цветов. Размещение статических цветов в системной палитре показано в табл. 3.1.

Таблица 3.1. Статические цвета в системной палитре

Индекс	Значение RGB	Цвет	Примечание
0	0x00, 0x00, 0x00	Черный	В любом режиме
1	0x80, 0x00, 0x00	Темно-красный	В любом режиме
2	0x00, 0x80, 0x00	Темно-зеленый	В любом режиме
3	0x80, 0x80, 0x00	Темно-желтый	В любом режиме
4	0x00, 0x00, 0x80	Темно-синий	В любом режиме
5	0x80, 0x00, 0x80	Темно-малиновый	В любом режиме
6	0x00, 0x80, 0x80	Темно-голубой	В любом режиме
7	0xC0, 0xC0, 0xC0	Светло-серый	В любом режиме
8	0xC0, 0xDC, 0xC0	Денежный зеленый	В режиме True Color
9	0xA6, 0xCA, 0xF0	Небесный	В режиме True Color
246	0xFF, 0xFB, 0xF0	Очень светло-серый	В режиме True Color
247	0xA0, 0xA0, 0xA4	Средне-серый	В режиме True Color
248	0x80, 0x80, 0x80	Темно-серый	В любом режиме
249	0xFF, 0x00, 0x00	Красный	В любом режиме
250	0x00, 0xFF, 0x00	Зеленый	В любом режиме
251	0xFF, 0xFF, 0x00	Желтый	В любом режиме
252	0x00, 0x00, 0xFF	Синий	В любом режиме
253	0xFF, 0x00, 0xFF	Малиновый	В любом режиме
254	0x00, 0xFF, 0xFF	Голубой	В любом режиме
255	0xFF, 0xFF, 0xFF	Белый	В любом режиме

Стоит обратить внимание на необычное размещение статических цветов. Так, первая группа из десяти цветов располагается в начале таблицы, а вторая группа из десяти цветов — в конце таблицы. Это связано с тем, что наиболее употребительная растровая операция **XOR** часто используется для инвертирования цветов и последующего их восстановления. Поэтому важно обеспечить инвертирование цвета при помощи инвертирования значения индекса в цветовой таблице. Например, инверсия нулевого значения индекса дает значение 255 (0xFF), при этом черный цвет трансформируется в белый. Позицию 1 занимает темно-красный цвет RGB(0x80, 0x00, 0x00). Если инвертировать индекс, то он переходит в 254 (0xFE), что соответствует голубому цвету. Он не совсем совпадает с цветом, дополняющим темно-красный цвет в цветовом пространстве RGB (RGB(0x7F, 0xFF, 0xFF)), но все же достаточно близок к нему.

Первые 8 цветов и последние 8 цветов никогда не изменяются, независимо от цветового режима. Реализация цветов с индексами 8, 9, 246, 247 зависит от режима. В таблице указаны RGB-значения этих цветов для режима True Color. В 256-цветном режиме RGB-значения этих цветов будут другими.

Чтобы увидеть, как выглядит системная палитра на экране, можно создать маленькую экспериментальную программу, код которой приведен в листинге 3.1.

Листинг 3.1. Проект SystemPalette

```
//////////  
// SystemPalette.cpp  
#include <windows.h>  
#include "KWnd.h"  
  
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
```

```
//=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("System palette", hInstance, nCmdShow, WndProc, NULL, 0, 0,
        524, 543);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    RECT r0;
    RECT rect;

    HBRUSH brush;
    int dW, dH, i, j;
    int index = 0;
    PALETTEENTRY pal[256];

    switch (uMsg)
    {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &ps);

        GetSystemPaletteEntries(hDC, 0, 256, pal);

        GetClientRect(hWnd, &r0);
        dW = (r0.right - 4) / 16;
        dH = (r0.bottom - 4) / 16;
        SetViewportOrgEx(hDC, 2, 2, NULL);

        for (i = 0; i < 16; ++i)
            for (j = 0; j < 16; ++j) {
                SetRect(&rect, j*dW, i*dH, (j+1)*dW - 1, (i+1)*dH - 1);
                brush = CreateSolidBrush(RGB(pal[index].peRed,
                    pal[index].peGreen, pal[index].peBlue));
                FillRect(hDC, &rect, brush);
                DeleteObject(brush);
                index++;
            }

        EndPaint(hWnd, &ps);
        break;

    case WM_PALETTECHANGED:
        InvalidateRect(hWnd, NULL, TRUE);
        break;

    case WM_DESTROY:
```

продолжение ↗

Листинг 3.1. (продолжение)

```

PostQuitMessage(0);
break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
/////////////////////////////////////////////////////////////////

```

Основная работа происходит в блоке обработки сообщения WM_PAINT.

Здесь вызовом функции GetSystemPaletteEntries содержимое системной палитры копируется в массив pal.

Затем при помощи функции GetClientRect координаты клиентской области окна программы запоминаются в переменной r0, которая является структурой типа RECT. Поскольку для клиентской области всегда r0.left и r0.top равны нулю, то величины r0.right и r0.bottom представляют собой ширину и высоту клиентской области соответственно. При заданных в WinMain размерах основного окна клиентская область представляет собой квадрат 516×516 пикселов. Поэтому переменные dW и dH, определяющие размеры и положение очередного маленького квадрата rect, заполняемого очередным цветом из системной палитры при помощи кисти brush, получают значение 32.

В блоке обработки сообщения WM_PALETTECHANGED вызывается функция InvalidateRect, которая, в свою очередь, заставляет систему послать оконной процедуре сообщение WM_PAINT для обновления содержимого клиентской области.

Создайте соответствующий проект и откомпилируйте программу¹. Если в настройках экрана задан 256-цветный режим, то будет отображена картинка, показанная на рис. 3.1. И снова приходится сожалеть, что на черно-белом рисунке потеряно много важной информации, характеризующей палитру, поэтому мы надеемся, что вы видите эту картинку на экране монитора². Обратите внимание на размещение статических цветов. Десять из них находятся в начале таблицы, и еще десять — в конце. Остальные 236 нестатических цветов более или менее равномерно распределены в цветовом пространстве RGB, образуя несколько радужных последовательностей с разной интенсивностью цветов. Теперь, не закрывая данную программу, запустите на выполнение какое-нибудь графическое приложение, например MS Paint. Вы увидите, что среди нестатических цветов произошли изменения. Это говорит о том, что приложение MS Paint перестроило логическую палитру по своим внутренним критериям.

Проведем другой эксперимент. Измените настройки экрана, переключившись в режим True Color³. Окно программы System palette примет вид, показанный на рис. 3.2. Легко заметить, что статические цвета остались на своих местах, а нестатические исчезли. Точнее, нестатические цвета оказались заполненными значениями RGB(0x00, 0x00, 0x00). Дело в том, что в режиме True Color цветовая палитра приложениям не нужна, поэтому система не заботится о ее заполнении.

¹ Не забудьте добавить к проекту файлы KWnd.h и KWnd.cpp, код которых приведен в листинге 1.2.

² Данный проект можно создать самостоятельно или загрузить его исходный код из файлов к книге, которые можно найти на сайте издательства «Питер» (www.piter.com).

³ Предполагается, что ваш дисплей поддерживает этот режим.

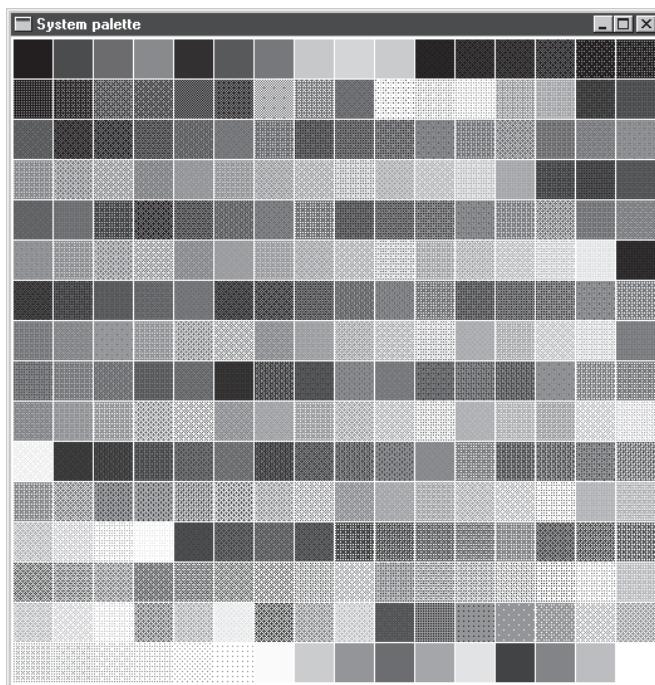


Рис. 3.1. Системная палитра в 256-цветном режиме экрана

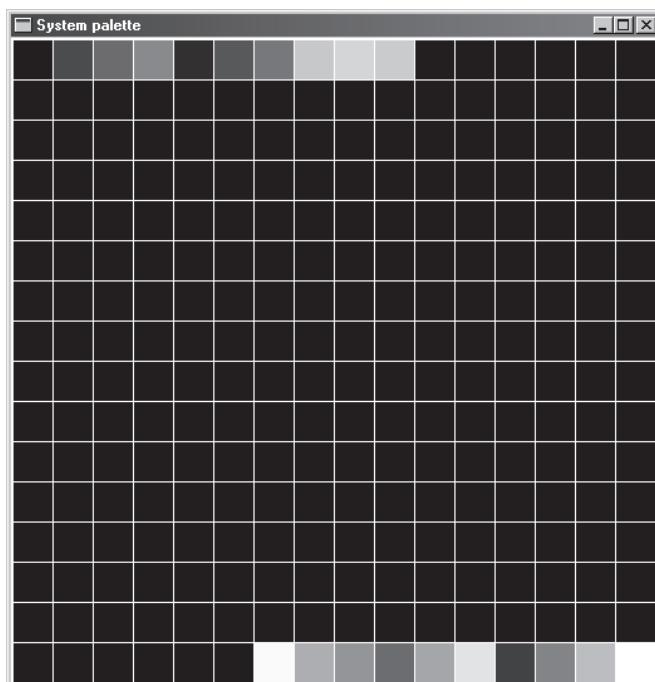


Рис. 3.2. Системная палитра в режиме True Color

Логическая палитра

При работе с логической палитрой следует помнить, что все цвета в приложении, которые реализованы как значения типа `COLORREF`, должны быть определены с помощью одного из следующих макросов:

```
COLORREF PALETTEINDEX(WORD wPaletteIndex);
COLORREF PALETERGB(BYTE bRed, BYTE bGreen, BYTE bBlue);
```

Макрос `PALETTEINDEX` получает индекс и возвращает 32-разрядное описание элемента палитры. В контексте устройства оно используется как элемент логической палитры контекста.

Макросу `PALETERGB`, как и макросу `RGB`, передаются значения красной, зеленой и синей составляющих искомого цвета. При использовании этих макросов в контексте устройства без логической палитры их возвращаемые значения интерпретируются одинаково. Но если макрос `PALETERGB` используется в контексте устройства с логической палитрой, то для указанных в нем RGB-составляющих система ищет ближайшее совпадение в логической палитре, словно приложение указало индекс в палитре. Этот макрос используется чаще, так как он является очевидной заменой макроса `RGB` во всех функциях, требующих передачи значения типа `COLORREF`.

Логическая палитра по умолчанию

Ранее уже говорилось, что если приложение не создает явным образом логическую палитру, то в нем используется *логическая палитра по умолчанию*, содержащая только 20 статических цветов из системной палитры.

Для экспериментов с логической палитрой по умолчанию создадим приложение, выводящее в свое окно 8 разных цветов (листинг 3.2).

Листинг 3.2. Проект ColorsWithDefaultPal

```
///////////
// ColorsWithDefaultPal.cpp
#include <windows.h>
#include "KWnd.h"

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("Colors with default palette", hInstance, nCmdShow, WndProc,
        NULL, 0, 0, 800, 200);
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
```

```
RECT r0;
RECT rect[8];
int dW, dH, i;
static HBRUSH brush[8];
COLORREF rgbcColor[8] = { RGB(128,0,0), RGB(192,0,0),RGB(255,0,0),
    RGB(255,192,0), RGB(176,250,133), RGB(245,197,137),
    RGB(255,128,128), RGB(255,128,255)};
switch (uMsg)
{
case WM_CREATE:
    for (i = 0; i < 8; ++i)
        brush[i] = CreateSolidBrush(rgbcColor[i]);
    break;

case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);
    GetClientRect(hWnd, &r0);
    dW = r0.right / 8;
    dH = r0.bottom;
    for (i = 0; i < 8; ++i) {
        SetRect(&rect[i], i*dW, 0, (i+1)*dW, dH);
        FillRect(hDC, &rect[i], brush[i]);
    }
    EndPaint(hWnd, &ps);
    break;

case WM_DESTROY:
    for (i = 0; i < 8; ++i)
        DeleteObject(brush[i]);
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
////////////////////////////////////////////////////////////////
```

Если ваш экран остался в режиме True Color, то программа выведет на экран окно, показанное на рис. 3.3.

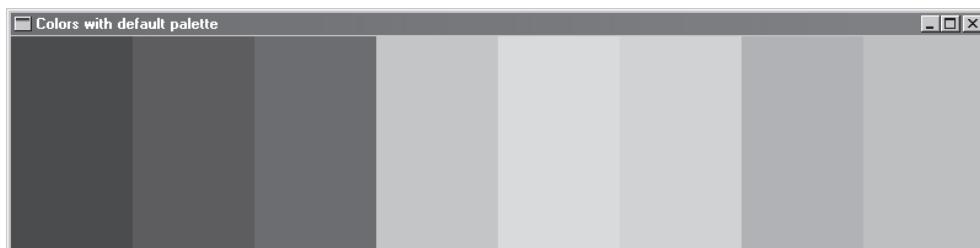


Рис. 3.3. Использование палитры по умолчанию (режим True Color)

В окне слева направо представлены 8 цветов, закодированные в массиве rgbcColor: темно-красный, темновато-красный, красный, оранжевый, салатный, бежевый, кремовый, сиреневый. Первые три цвета являются вариациями чистого красного

цвета с разной интенсивностью, остальные цвета образованы подходящими комбинациями RGB-составляющих цвета.

Теперь переключитесь в 256-цветный режим экрана и снова запустите программу. Вывод программы показан на рис. 3.4.

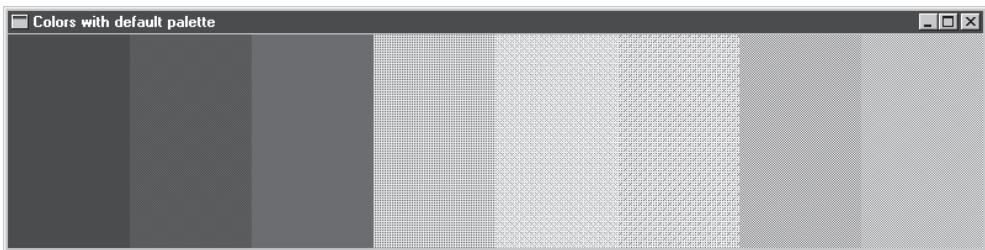


Рис. 3.4. Использование палитры по умолчанию (256-цветный режим)

Особенно заметно искажились оранжевый, салатный и бежевый цвета. Причиной этого является использование только двадцати статических цветов из системной палитры. Поскольку для указанных трех цветов не нашлось ближайшего подходящего цвета, то система попыталась создать запрошенный цвет при помощи смешения статических цветов.

Полутоновая палитра

Чтобы увеличить количество имеющихся цветов в палитре, можно воспользоваться полутоновой палитрой. Для этого необходимо выполнить следующие шаги:

1. Создать полутоновую палитру (функция `CreateHalftonePalette`).
2. Выбрать созданную палитру в контекст устройства (функция `SelectPalette`).
3. Реализовать палитру при помощи функции `RealizePalette`.

Функция `SelectPalette` имеет следующий прототип:

```
HPALETTE SelectPalette(
    HDC hdc,           // дескриптор контекста устройства
    HPALETTE hpal,     // дескриптор логической палитры
    BOOL bForceBackground // режим (фоновый или основной)
);
```

Последний параметр функции определяет, в каком режиме — фоновом (`TRUE`) или основном (`FALSE`) — будет реализована палитра при вызове функции `RealizePalette`. Процесс реализации палитры в общих чертах был описан выше.

Следует уточнить, что в фоновом режиме функция `RealizePalette` пытается разместить логическую палитру привязкой к уже существующим цветам в системной палитре, подыскивая ближайшее соответствие. Это размещение делается всегда, независимо от того, активно ли данное приложение.

В основном режиме функция `RealizePalette` копирует логическую палитру в системную, за исключением статических цветов, только тогда, когда приложение активно.

В листинге 3.3 приведен код приложения, которое выполняет те же действия, что и предыдущее приложение, но работает с полутоновой палитрой. В нем показана также обработка сообщений `WM_QUERYNEWPALETTE` и `WM_PALETTECHANGED`.

Листинг 3.3. Проект ColorsWithHalftonePal

```
////////////////////////////////////////////////////////////////
// ColorsWithHalftonePal.cpp
#include <windows.h>
#include "KWnd.h"

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("Colors with halftone palette", hInstance, nCmdShow,
        WndProc, NULL, 0, 0, 800, 200);
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    RECT r0;
    RECT rect[8];
    int dW, dH, i;

    COLORREF paletteColor[8] = { PALETERGB(128,0,0), PALETERGB(192,0,0),
        PALETERGB(255,0,0), PALETERGB(255,192,0), PALETERGB(176,250,133),
        PALETERGB(245,197,137), PALETERGB(255,128,128),
        PALETERGB(255,128,255)};

    static HBRUSH brush[8];
    static HPALETTE hPal;
    HPALETTE hOldPal;

    switch (uMsg)
    {
    case WM_CREATE:
        for (i = 0; i < 8; ++i)
            brush[i] = CreateSolidBrush(paletteColor[i]);
        hDC = GetDC(hWnd);
        hPal = CreateHalftonePalette(hDC);
        ReleaseDC(hWnd, hDC);
        break;

    case WM_PAINT:
        hDC = BeginPaint(hWnd, &ps);
        hOldPal = SelectPalette(hDC, hPal, FALSE);
        GetClientRect(hWnd, &r0);
        dW = r0.right / 8;
        dH = r0.bottom;
        for (i = 0; i < 8; ++i) {
            SetRect(&rect[i], i*dW, 0, (i+1)*dW, dH);
            FillRect(hDC, &rect[i], brush[i]);
        }
    }
```

продолжение ↗

Листинг 3.3 (продолжение)

```

SelectPalette(hDC, hOldPal, TRUE);
EndPaint(hWnd, &ps);
break;

case WM_QUERYNEWPALETTE:
    hDC = GetDC(hWnd);
    SelectPalette(hDC, hPal, FALSE);
    if (RealizePalette(hDC))
        InvalidateRect(hWnd, NULL, TRUE);
    ReleaseDC(hWnd, hDC);
break;

case WM_PALETTECHANGED:
    hDC = GetDC(hWnd);
    SelectPalette(hDC, hPal, FALSE);
    if (RealizePalette(hDC))
        InvalidateRect(hWnd, NULL, TRUE);
    ReleaseDC(hWnd, hDC);
break;

case WM_DESTROY:
    DeleteObject(hPal);
    for (i = 0; i < 8; ++i)
        DeleteObject(brush[i]);
    PostQuitMessage(0);
break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
/////////////////////////////////////////////////////////////////

```

Обратите внимание на следующие моменты:

- ❑ Логическая палитра `hPal` создается в блоке обработки сообщения `WM_CREATE`, а реализуется как в блоке обработки сообщения `WM_QUERYNEWPALETTE`, так и в блоке обработки сообщения `WM_PALETTECHANGED`. Функция `RealizePalette` возвращает количество цветов логической палитры, отображенных на системную палитру. Если это значение больше нуля, то вызывается функция `InvalidateRect`.
- ❑ В блоке обработки сообщения `WM_PAINT` логическая палитра выбирается в контекст устройства, после чего вызываются функции рисования.
- ❑ После запуска этой программы в 256-цветном режиме экрана вы должны увидеть картинку, практически совпадающую по качеству передачи цветов с той, которая была в режиме экрана True Color (см. рис. 3.3).

Использование специализированной палитры

Бывают ситуации, когда применение полутоновой палитры не обеспечивает удовлетворительной передачи всех требуемых цветов. Это характерно для приложений, которым необходимо отображать множество оттенков одного и того же цвета.

Нетрудно показать на примере предыдущего приложения, что оно не справится с выводом даже восьми оттенков одного, например зеленого, цвета. Для этого замените определение массива `paletteColor` следующим определением:

```
COLORREF paletteColor[8] = { PALETTERRGB(0,31,0), PALETTERRGB(0,63,0),
    PALETTERRGB(0,95,0), PALETTERRGB(0,127,0), PALETTERRGB(0,159,0),
    PALETTERRGB(0,191,0), PALETTERRGB(0,223,0), PALETTERRGB(0,255,0)};
```

Модифицированная программа выведет картинку, в которой первый, второй, шестой и седьмой прямоугольники *неразличимы по цвету*.

Это означает, что пришло время применить специализированную логическую палитру. Для ее создания используется функция `CreatePalette`:

```
HPALETTE CreatePalette(CONST LOGPALETTE* lplgpl);
```

Параметр `lplgpl` содержит указатель на структуру `LOGPALETTE`, определение которой имеет следующий вид:

```
typedef struct tagLOGPALETTE {
    WORD     palVersion; // номер версии палитры
    WORD     palNumEntries; // число входов в логической палитре
    PALETTEENTRY palPalEntry[1];
} LOGPALETTE;
```

Номер версии палитры не изменился со времен ее появления в Windows 3.0 и по-прежнему равен 0x300.

Поле `palPalEntry` содержит адрес массива структур `PALETTEENTRY`, определяющих цвет каждого элемента в логической палитре.

Структура `PALETTEENTRY` уже рассматривалась в разделе «Системная палитра». Три ее первых поля обычно описывают интенсивность компонентов RGB-модели, а поле `peFlags` указывает, как данный элемент должен интерпретироваться при реализации палитры. Возможные значения поля `peFlags` перечислены в табл. 3.2.

Таблица 3.2. Возможные значения поля `peFlags`

Значение	Описание
0	Стандартная процедура, предписывающая искать цвет RGB в системной палитре. Если цвет отсутствует, то он включается в палитру
PC_NOCOLLAPSE	Искать соответствие в системной палитре лишь при отсутствии свободных (неиспользуемых) элементов. В противном случае нужно использовать первый свободный элемент
PC_EXPLICIT	Не изменять системную палитру. В первые два байта структуры <code>PALETTEENTRY</code> поместить индекс элемента в системной палитре. Этот флаг позволяет приложению показать содержимое аппаратной палитры
PC_RESERVED	Указывает, что данный элемент палитры будет использоваться для анимации. Этот флаг запрещает другим окнам согласовывать какой-либо цвет с данным элементом палитры. Если в системной палитре есть свободные элементы, то первый из них используется для размещения данного элемента

Отметим, что в определении структуры `LOGPALETTE` массив `palPalEntry` содержит всего один элемент, так как нельзя заранее сказать, сколько элементов может потребоваться для создания логической палитры. Поэтому в коде приложения необходимо либо расширить определение типа `LOGPALETTE`, либо выделить в куче блок памяти необходимого размера и адресовать к нему при помощи указателя на `LOGPALETTE`. В рассматриваемой ниже программе демонстрируется первый подход.

После создания объекта палитры он должен быть реализован вызовом функции `RealizePalette` так же, как и в случае с полутоновой палитрой.

В листинге 3.4 приводится код программы, в которой создается логическая палитра, содержащая 128 градаций зеленого цвета, и демонстрируется вывод 16 прямоугольников, равномерно заполненных различными оттенками зеленого цвета.

Листинг 3.4. Проект LogicPalette

```

//=====
// LogicPalette.cpp
#include <windows.h>
#include "KWnd.h"

#define PAL_NUM_ENTRIES 128
#define NUM_RECT 16

typedef struct {
    LOGPALETTE lp;
    PALETTEENTRY ape[PAL_NUM_ENTRIES - 1];
} LogPal;

LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    MSG msg;
    hWnd = mainWnd("Colors with logic palette", hInstance, nCmdShow, WndProc,
        NULL, 0, 0, 808, 200);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    PAINTSTRUCT ps;
    RECT r0;
    RECT rect[NUM_RECT];
    int dw, dh, i;

    static LogPal pal;
    LOGPALETTE* pLP = (LOGPALETTE*) &pal;

    static COLORREF paletteColor[NUM_RECT];
    static HBRUSH brush[NUM_RECT];
    static HPALETTE hPal;
    HPALETTE hOldPal;

    BYTE green;
    BYTE stepPal, stepBrush;

    switch (uMsg)
    {
    case WM_CREATE:
        // Создаем логическую палитру
        pLP->palVersion = 0x300; // номер версии Windows
        pLP->palNumEntries = PAL_NUM_ENTRIES; // число входов палитры
        stepPal = 256 / PAL_NUM_ENTRIES;

```

```

green = 255;
for (i = 0; i < PAL_NUM_ENTRIES; i++) {
    pLP->palPalEntry[i].peRed = 0;
    pLP->palPalEntry[i].peGreen = green;
    pLP->palPalEntry[i].peBlue = 0;
    pLP->palPalEntry[i].peFlags = 0;
    green -= stepPal;
}
hPal = CreatePalette (pLP);
// Готовим кисти для рисования
stepBrush = 256 / NUM_RECT;
for (i = 0; i < NUM_RECT; ++i)
    brush[i] = CreateSolidBrush(PALETTERGB(0, stepBrush * i, 0));
break;

case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);
    hOldPal = SelectPalette(hDC, hPal, FALSE);
    GetClientRect(hWnd, &r0);
    dW = r0.right / NUM_RECT;
    dH = r0.bottom;
    for (i = 0; i < NUM_RECT; ++i) {
        SetRect(&rect[i], i*dW, 0, (i+1)*dW, dH);
        FillRect(hDC, &rect[i], brush[i]);
    }
    SelectPalette(hDC, hOldPal, TRUE);
    EndPaint(hWnd, &ps);
break;

case WM_QUERYNEWPALETTE:
    hDC = GetDC(hWnd);
    SelectPalette(hDC, hPal, FALSE);
    if (RealizePalette(hDC))
        InvalidateRect(hWnd, NULL, TRUE);
    ReleaseDC(hWnd, hDC);
break;

case WM_PALETTECHANGED:
    hDC = GetDC(hWnd);
    SelectPalette(hDC, hPal, FALSE);
    if (RealizePalette(hDC))
        InvalidateRect(hWnd, NULL, TRUE);
    ReleaseDC(hWnd, hDC);
break;

case WM_DESTROY:
    DeleteObject(hPal);
    for (i = 0; i < NUM_RECT; ++i)
        DeleteObject(brush[i]);
    PostQuitMessage(0);
break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}

```

Следует обратить внимание на определение типа LogPal в глобальной области видимости программы, которое расширяет стандартный тип LOGPALETTE. Наш пользовательский тип включает структуру типа LOGPALETTE, после которой помещен массив структур PALETTEENTRY, содержащий (`PAL_NUM_ENTRIES - 1`) элементов. Объект типа LogPal объявляется в теле функции WndProc:

```
static LogPal pal;
```

Чтобы работать с этим объектом, интерпретируя его как переменную типа LOGPALETTE, объявляется указатель на тип LOGPALETTE, после чего ему присваивается адрес переменной `pal`:

```
LOGPALETTE* pLP = (LOGPALETTE*) &pal;
```

Количество входов в логической палитре не может превышать 236, так как 20 элементов системной палитры занято статическими цветами. В нашей программе это количество задано константой `PAL_NUM_ENTRIES`, равной 128, поскольку это даже перекрывает потребности приложения, выводящего 16 градаций зеленого цвета.

Создание логической палитры осуществляется в блоке обработки сообщения `WM_CREATE`. Для этого сначала инициализируются поля объекта `pal`, адресуемые через указатель `pLP`, причем массив `palPalEntry` равномерно заполняется распределенными градациями зеленого цвета. После этого вызывается функция `CreatePalette`. Реализация палитры происходит при обработке сообщений `WM_QUERYNEWPALETTE` и `WM_PALETTECHANGED`. И наконец, в блоке обработки сообщения `WM_PAINT` логическая палитра используется после выбора ее в контекст устройства при помощи функции `SelectPalette`.

После запуска программы вы должны получить картинку, показанную на рис. 3.5.



Рис. 3.5. Использование специализированной палитры (256-цветный режим)

Растры

Графические объекты, рассмотренные в главе 2, такие как пиксели, линии и замкнутые фигуры, могут использоваться для построения инженерных и финансовых диаграмм, чертежей, геометрических узоров и иных графических композиций. Геометрические фигуры в этих изображениях описываются точными математическими формулами. Соответствующая область программирования компьютерной графики называется *векторной графикой* (*vector graphics*).

В другой, не менее важной области компьютерной графики используются оцифрованные изображения, полученные из окружающего мира. Эта область называется *растровой графикой* (*bitmap graphics*). Обычно растровые изображения получают от таких устройств, как цифровой фотоаппарат, сканер, видеокамера, либо

создают в каком-нибудь графическом редакторе, к которым относится, например, Microsoft Paint или Adobe Photoshop.

Растровый, или битовый, образ (bitmap) – это оцифрованное представление изображения. Каждый пиксель изображения представлен в растровом образе одним или несколькими битами, в которых закодирован его цвет. В монохромных битовых образах для хранения информации о каждом пикселе достаточно одного бита. Но для кодирования цветных изображений требуется более одного бита на пиксель. Число цветов, которые могут быть представлены в битовом образе, равно 2^n , где n – количество битов на пиксель. Например, для старой модели дисплея VGA с поддержкой 256 цветов битовые образы содержали по 8 битов на пиксель. Для современных мониторов, поддерживающих до 2^{24} цветов, битовый образ должен содержать 24 бита на пиксель, когда для каждого RGB-компоненты цвета используется 8 битов.

В ранних версиях Windows использовался только один формат битового образа, получивший позже наименование *аппаратно-зависимого растра*, или *DDB (device dependent bitmap)*. Особенность формата DDB состоит в том, что система всегда создает битовый образ в памяти с учетом характеристик конкретного графического оборудования. Например, если компьютер оснащен 256-цветным дисплеем, то созданный битовый образ будет содержать по 8 битов на пиксель. Формат DDB не подходил для переноса растров на другие компьютеры, так как характеристики графических устройств изменяются в очень широких пределах. Более того, даже на одном компьютере дисплей может работать в разных режимах, в зависимости от текущих настроек экрана.

Начиная с Windows 3.0, был введен новый формат битовых образов, названный *аппаратно-независимым растром*, или *DIB (device independent bitmap)*. Битовый образ DIB кроме самого массива пикселов содержит справочную информацию о растре и цветовую таблицу. В цветовой таблице отражается соответствие двоичного представления пикселов цветам RGB, поэтому битовые образы этого формата после сохранения в файле могут быть воспроизведены на любом растровом графическом устройстве. Если устройство, на которое переносится DIB, имеет более узкий диапазон поддерживаемых цветов, то цвета из таблицы преобразуются к ближайшим цветам, которые устройство может действительно воспроизвести.

В настоящее время аппаратно-зависимые растры (DDB) используются во внутренней работе графической системы Windows, а также для представления тех битовых образов в программе, которые не предназначены для передачи во внешний мир. Аппаратно-независимые растры (DIB), напротив, применяются для обмена битовыми образами между программами как в форме файлов, так и через буфер обмена Windows.

Мы начнем наше путешествие по растровой тематике с рассмотрения DIB, потом перейдем к DDB и в завершение познакомимся с новым типом растров, поддерживаемым в Win32 GDI, – DIB-секциями.

Аппаратно-независимые растры

Аппаратно-независимые растры (DIB) содержат полную информацию об изображении, что позволяет правильно отображать изображение на самых различных устройствах. Обычно файлы, в которых сохраняется DIB, имеют расширение .bmp. Поэтому формат DIB-растров имеет альтернативное наименование «файловый формат BMP».

Файловый формат BMP

Компоненты BMP-формата, размещаемые в памяти последовательно один за другим, перечислены в табл. 3.3.

Таблица 3.3. Формат DIB-растра (Файловый формат BMP)

Компонент	Размер	Примечание
Заголовок растрового файла BITMAPFILEHEADER bmfHeader;	14 байт	
Заголовок информационного блока BITMAPINFOHEADER bmiHeader;	40 байт	Возможно использование структуры BITMAPV4HEADER (108 байт) или BITMAPV5HEADER (124 байта)
Цветовая таблица RGBQUAD bmiColors[];	Переменный	Может отсутствовать
Массив пикселов BYTE aBitmapBits[][];	Переменный	

Заголовок растрового файла

Структура BITMAPFILEHEADER, используемая для объявления заголовка растрового файла bmfHeader, имеет следующее определение:

```
typedef struct tagBITMAPFILEHEADER {
    WORD bfType;      // тип файла
    DWORD bfSize;     // размер файла в байтах
    WORD bfReserved1; // резервное поле (должно быть равно нулю)
    WORD bfReserved2; // резервное поле (должно быть равно нулю)
    DWORD bfOffbits;  // байтовое смещение до начала массива пикселов
} BITMAPFILEHEADER;
```

Поле *bfType* должно содержать ASCII-символы В и М, которые, разумеется, означают *bitmap*. Соответствующие шестнадцатеричные значения равны 0x42 и 0x4D. При любых других значениях этого поля файл не будет опознаваться Windows как растровое изображение. Учтите, что слово типа WORD размещается в памяти, начиная с младшего байта, поэтому, если вы формируете заголовок, используйте инструкцию вида *bfType* = 0x4d42.

Поле *bfSize* содержит размер всего файла в байтах.

Поле *bfOffbits* задает байтовое смещение до начала растрового изображения. Если BMP-файл прочитан в память, то поле *bfOffbits* позволяет вычислить адрес начала массива *aBitmapBits*.

Заголовок информационного блока

После заголовка растрового файла следует заголовок информационного блока bmiHeader, в котором содержится информация о размерах и цветовом формате раstra. Соответствующая структура BITMAPINFOHEADER определена в файле wingdi.h:

```
typedef struct tagBITMAPINFOHEADER{
    DWORD biSize;          // размер структуры BITMAPINFOHEADER
    LONG biWidth;          // ширина раstra в пикселях
    LONG biHeight;          // высота раstra в пикселях + ориентация
    WORD biPlanes;          // количество плоскостей (должно быть равно 1)
    WORD biBitCount;        // количество битов на пикセル
    DWORD biCompression;    // алгоритм сжатия
    DWORD biSizeImage;       // размер массива пикселов в байтах
} BITMAPINFOHEADER;
```

```

    LONG      biXPelsPerMeter; // горизонтальное разрешение устройства вывода
    LONG      biYPelsPerMeter; // вертикальное разрешение устройства вывода
    DWORD     biClrUsed;      // количество элементов в цветовой таблице
    DWORD     biClrImportant; // количество элементов, реально используемых для вывода
                           // растра
} BITMAPINFOHEADER;

```

Поля *biWidth* и *biHeight* определяют ширину и высоту битового изображения. Высота *biHeight* обычно является положительной величиной, но она может быть и отрицательной. Знак этого поля определяет порядок следования строк развертки в массиве пикселов. Положительное значение соответствует обратному порядку строк (снизу вверх), при котором первый пиксель массива является первым пикселом *последней строки* развертки изображения. Такие DIB-растры называют *перевернутыми (bottom-up)*. Отрицательное значение соответствует прямому порядку строк (сверху вниз), и такие DIB-растры называют неперевернутыми (*top-down*). В большинстве BMP-файлов используется обратный порядок следования строк развертки.

Поле *biPlanes* задает количество цветовых плоскостей для целевого графического устройства. В разных устройствах может использоваться различная структура строк развертки (с одной или несколькими цветовыми плоскостями). Формат DIB поддерживает изображения только с одной плоскостью, поэтому поле *biPlanes* должно иметь единичное значение.

Поле *biBitCount* содержит количество битов, используемых для кодирования цвета одного пикселя. Иногда эту характеристику называют *цветовой глубиной*. Возможные значения поля приведены в табл. 3.4.

Таблица 3.4. Значения поля *biBitCount*

Значение	Максимальное количество цветов	Размер пикселя, байт	Описание
0	Зависит от внедренного изображения		Число битов на пиксель специфицируется форматом JPEG или PNG (только для Windows 98/2000)
1	2^1 (2)	1 / 8	Монохромное изображение. Цветовая таблица <i>bmiColors</i> содержит два элемента, соответствующие двум значениям типа RGBQUAD. Каждый бит из массива <i>aBitmapBits</i> определяет индекс элемента массива <i>bmiColors</i> . Если бит равен 0, то он окрашивается в соответствии с содержимым <i>bmiColors[0]</i> , если 1 — в соответствии с содержимым <i>bmiColors[1]</i>
4	2^4 (16)	1 / 2	Изображение содержит до 16 цветов, соответственно, цветовая таблица <i>bmiColors</i> содержит до 16 значений типа RGBQUAD. Каждый байт в массиве <i>aBitmapBits</i> представляет два пикселя. Цвет пикселя определяется его 4-битным индексом в цветовой таблице
8	2^8 (256)	1	Изображение содержит до 256 цветов, и <i>bmiColors</i> содержит до 256 элементов. В этом случае каждый байт в массиве <i>aBitmapBits</i> представляет отдельный пиксель. Значение байта используется как индекс в цветовой таблице

Таблица 3.4 (продолжение)

Значение	Максимальное количество цветов	Размер пикселя, байт	Описание
16	2^{16} (65 536)	2	Изображение содержит до 2^{16} цветов (пиксели типа High Color). Если поле biCompression равно BI_RGB, то цветовая таблица bmiColors обычно отсутствует. Каждое слово (WORD) в массиве aBitmapBits представляет отдельный пиксель. Относительная интенсивность синего, зеленого и красного цветов представлена пятью битами для каждого цветового компонента. Старший знаковый разряд слова не используется Если поле biCompression равно BI_BITFIELDS, то после заголовка информационного блока и перед цветовой таблицей добавляются три двойных слова — маски для извлечения синей, зеленой и красной составляющих из слова в массиве aBitmapBits
24 битные	2^{24} (16 777 216)	3	Изображение содержит до 2^{24} цветов (24-пиксели True Color). Цветовая таблица bmiColors обычно отсутствует. Каждый трехбайтный триплет в массиве aBitmapBits характеризует относительную интенсивность синей, зеленой и красной составляющих для цвета очередного пикселя. Иногда таблица bmiColors может использоваться для оптимизации вывода на устройствах, работающих с палитрой. В этом случае цветовая таблица содержит число входов, специфицированное в поле biClrUsed
32	2^{24} (16 777 216)	4	Изображение содержит до 2^{24} цветов (32-битные пиксели True Color). Если поле biCompression равно BI_RGB, то цветовая таблица bmiColors обычно отсутствует. Каждое двойное слово (DWORD) в массиве aBitmapBits характеризует отдельный пиксель. Относительная интенсивность синего, зеленого и красного цветов задана тремя байтами в двойном слове. Старший байт не используется. Если поле biCompression равно BI_BITFIELDS, то после заголовка информационного блока и перед цветовой таблицей добавляются три двойных слова — маски для извлечения синей, зеленой и красной составляющих из двойного слова в массиве aBitmapBits5

Иногда старший байт в 32-битных пикселях True Color используется для хранения альфа-канала, то есть данных о прозрачности пикселя.

Поле biCompression определяет алгоритм сжатия, применяемый к массиву пикселов. Допустимые значения поля приведены в табл. 3.5.

Если поле biHeight имеет отрицательное значение, то поле biCompression может принимать только значения BI_RGB или BI_BITFIELDS, поскольку неперевернутые раstry не допускают сжатие.

Таблица 3.5. Значения поля biCompression

Константа	Код	Описание
BI_RGB	0x0	Несжатое изображение
BI_RLE8	0x1	Изображение с кодировкой 8 бит/пиксел, сжатое с использованием алгоритма RLE
BI_RLE4	0x2	Изображение с кодировкой 4 бит/пиксел, сжатое с использованием алгоритма RLE
BI_BITFIELDS	0x3	Несжатые изображения с кодировкой 16 и 32 бит/пиксел. В изображение включаются три битовые маски, определяющие способ хранения компонентов RGB

Чаще всего поле *biCompression* содержит значение BI_RGB. Графические редакторы от Microsoft, а также среда разработки Visual Studio генерируют BMP-файлы только в несжатом формате. При отсутствии сжатия каждая строка развертки изображения представляет собой упакованный массив пикселов. Размер пикселя в байтах указан в табл. 3.4. Стока развертки всегда выравнивается по ближайшей границе двойного слова, а при необходимости строка дополняется нулями.

В растре DIB с кодировкой 4 и 8 бит/пиксел для уменьшения размеров растра может применяться необязательное сжатие по алгоритму RLE (Run-Length Encoding). Для 8-битных изображений этот алгоритм ищет последовательность смежных байтов с одинаковым значением, после чего заменяет их двумя байтами, в которых указывается счетчик повторений и код повторяющегося байта. Предусмотрены особые служебные последовательности для неповторяющихся байтов, для конца строки и конца изображения. Более подробное описание алгоритма RLE можно найти в книге [6].

Следующим полем в структуре BITMAPINFOHEADER является поле *biSizeImage*, содержащее размер массива пикселов в байтах. Если значением *biCompression* является BI_RGB, то поле *biSizeImage* может быть равно нулю. В этом случае GDI вычислит размер изображения по ширине, высоте и количеству битов на пиксель. Но при использовании сжатия RLE, а также для файлов формата JPEG или PNG это поле должно содержать фактический размер данных изображения.

Поля *biXPelsPerMeter* и *biYPelsPerMeter* содержат, соответственно, горизонтальное и вертикальное разрешение в пикселях на метр для целевого графического устройства. Приложение может использовать это значение, чтобы выбрать растр из группы доступных растров, который наиболее близок к характеристикам текущего устройства вывода.

Поле *biClrUsed* задает количество элементов в цветовой таблице. Для DIB-растров с количеством цветов, не превышающим 256, нулевое значение поля *biClrUsed* означает максимально возможное количество цветов, указанное в табл. 3.4. Обычный BMP-файл, хранящийся на диске, должен содержать полную цветовую таблицу с максимальным количеством элементов. Неполная цветовая таблица может использоваться только в DIB-растрах, которые хранятся в памяти.

Поле *biClrImportant* определяет количество элементов, реально необходимых для отображения растра. Нулевое значение поля означает, что используются все цвета из цветовой таблицы.

Следует отметить, что структура BITMAPINFOHEADER обычно называется «версией 3» описания растра. Этот номер версии принято относить к тем аспектам Win32

API, которые появились в Windows 3.1. Соответственно, новые возможности, добавленные в Windows 95 и Windows NT 4.0, именуются «версией 4», а новые возможности Windows 98 и Windows 2000 относятся к «версии 5».

В Windows 95 и Windows NT 4.0 появилась новая структура **BITMAPV4HEADER**, а в Windows 98 и Windows 2000 была добавлена структура **BITMAPV5HEADER**. Начало этих новых структур совпадает с типом **BITMAPINFOHEADER**. В структуре версии 4 появились новые поля для цветовых масок RGBA¹, цветовых пространств, конечных точек и гамма-коррекции (для поддержки ICM 1.0²). В структуре версии 5 добавились новые типы цветовых пространств, рекомендации по воспроизведению и данные цветового профиля, ориентированные на поддержку ICM 2.0. Подробные описания этих структур приведены в MSDN.

Цветовая таблица

Цветовая таблица является массивом структур типа **RGBQUAD**. Этот тип определен следующим образом:

```
typedef struct tagRGBQUAD {
    BYTE    rgbBlue;
    BYTE    rgbGreen;
    BYTE    rgbRed;
    BYTE    rgbReserved;
} RGBQUAD;
```

Поля структуры задают относительную интенсивность для синей, зеленоЙ и красной составляющих цвета пикселя. Последнее поле в структуре является резервным.

Обычно цветовая таблица используется в DIB-растрах, содержащих не более 256 цветов. В этом случае каждый пиксель массива **aBitmapBits** содержит индекс в цветовой таблице. Иногда цветовую таблицу включают и в DIB-растры формата True Color (Hi Color), что, на первый взгляд, кажется избыточным. Но это позволяет отображать указанные раstry с требуемым качеством передачи цветов на устройствах, не поддерживающих формат True Color, а работающих с цветовой палитрой.

Количество элементов в цветовой таблице задается в поле **biClrUsed** заголовка информационного блока. Если это поле имеет нулевое значение, то используется максимальное количество элементов для заданной цветовой глубины.

Массив пикселов

Пиксели изображения хранятся в массиве пикселов **aBitmapBits**. Если поле **biHeight** заголовка информационного блока содержит положительное значение, то строки развертки хранятся в обратном порядке, если же значение отрицательное — то в прямом порядке.

Пиксели упаковываются внутри строки развертки для экономии места. Строки дополняются битами до границы двойного слова.

Количество байтов на строку вычисляется по следующей формуле:

```
bytesPerLine = ((width * bitCount + 31) / 32) * 4;
```

где **width** — ширина изображения в пикселях, а **bitCount** — количество бит на пикセル.

¹ В режиме RGBA цвета сохраняются как совокупность Red-, Green-, Blue- и Alpha-компонентов.

² ICM — Integrated Color Management — набор функций уровня API и встроенных программных модулей, предназначенный для сквозного управления цветопередачей при воспроизведении изображений на разном периферийном оборудовании.

Для несжатых растров DIB, у которых поле `biCompression` имеет значение `BI_RGB`, обращение к отдельным пикселям массива является простой операцией, которая реализуется достаточно эффективно. Массив пикселов `aBitmapBits` можно объявить как двумерный, что отражало бы семантику хранения в нем строк развертки изображения. Однако основные функции отображения DIB в контексте устройства получают в качестве аргумента адрес одномерного массива. Поэтому удобней объявлять `aBitmapBits` как одномерный массив, а прямой доступ к пикселям осуществлять посредством преобразования пары индексов «логического» двумерного массива в эквивалентный индекс «физического» одномерного массива. Пример прямого доступа к пикселям приведен в листинге 3.5.

Упакованный аппаратно-независимый растр

Если есть файл DIB, который следует загрузить в приложение, то можно считать его непосредственно в выделенный блок памяти. Такой блок называют *упакованным (packed)* DIB-растром. Он содержит все компоненты файла DIB, кроме заголовка `bmfHeader`. Таким образом, упакованный DIB-растр начинается с заголовка информационного блока `bmiHeader`, за которым следуют массив масок, если он нужен, цветовая таблица (если она существует) и массив пикселов. В качестве указателя на упакованный DIB-растр в GDI обычно используется указатель на структуру `BITMAPINFO`. Следует отметить, что термин «упакованный» в данном случае не имеет никакого отношения к упаковке пикселов в строке развертки. Он просто означает, что компоненты DIB следуют друг за другом в смежных ячейках памяти.

Довольно большое количество функций, входящих в состав Win32 API, получает и возвращает упакованные DIB-растры. Кроме того, упакованные DIB-растры используются в работе буфера обмена Windows.

Отображение DIB в контексте устройства

Для вывода DIB-растра на поверхность графического устройства предназначены функции `StretchDIBits` и `SetDIBitsToDevice`.

Функция `StretchDIBits` имеет следующий прототип:

```
int StretchDIBits(
    HDC hdc,           // дескриптор контекста устройства
    int XDest,         // x-координата приемного прямоугольника
    int YDest,         // y-координата приемного прямоугольника
    int nDestWidth,   // ширина приемного прямоугольника
    int nDestHeight,  // высота приемного прямоугольника
    int XSrc,          // x-координата исходного прямоугольника
    int YSrc,          // y-координата исходного прямоугольника
    int nSrcWidth,    // ширина исходного прямоугольника
    int nSrcHeight,   // высота исходного прямоугольника
    CONST VOID* lpBits, // массив пикселов
    CONST BITMAPINFO* lpBitsInfo, // заголовок информационного блока
    UINT iUsage,       // интерпретация цветовой таблицы
    DWORD dwRop        // код растровой операции
);
```

Координаты и размеры исходного прямоугольника¹, относящегося к DIB-растру, задаются в пикселях. Координаты и размеры приемного прямоугольника,

¹ Термины «x-координата» и «y-координата» относятся к левому верхнему углу прямоугольника.

находящегося на поверхности графического устройства, задаются в логических единицах.

Манипулируя этими параметрами, можно получать различные преобразования при выводе изображения. Введем следующие обозначения:

- ❑ wBmp — ширина раstra в пикселях;
- ❑ hBmp — высота раstra в пикселях;
- ❑ xs — значение аргумента для XSrc;
- ❑ ys — значение аргумента для YSrc;
- ❑ ws — значение аргумента для nSrcWidth;
- ❑ hs — значение аргумента для nSrcHeight;
- ❑ xd — значение аргумента для XDest;
- ❑ yd — значение аргумента для YDest;
- ❑ wd — значение аргумента для nDestWidth;
- ❑ hd — значение аргумента для nDestHeight.

Исходный прямоугольник определяет, какая часть раstra выводится и осуществляются ли при этом зеркальные отражения по горизонтали и по вертикали. Возможные значения для соответствующих параметров приведены в табл. 3.6.

Таблица 3.6. Интерпретация параметров исходного прямоугольника

Значения для [xs, ys, ws, hs]	Исходный прямоугольник
0, 0, wBmp, hBmp	Все изображение, исходная ориентация
wBmp, 0, -wBmp, hBmp	Все изображение, зеркальное отражение по горизонтали
0, -hBmp, wBmp, hBmp	Все изображение, зеркальное отражение по вертикали
wBmp, hBmp, -wBmp, -hBmp	Все изображение, зеркальное отражение по горизонтали и по вертикали
0, 0, wBmp, 1	Первая строка развертки
0, 0, 1, hBmp	Первый столбец развертки

Приемный прямоугольник, задаваемый четверкой [xd, yd, wd, hd], определяет место на поверхности графического устройства, в котором размещается выводимая часть раstra, а также указывает, нужно ли масштабировать изображение. Для приемного прямоугольника также могут быть определены зеркальные отражения по горизонтали и по вертикали, которые осуществляются по правилам, аналогичным приведенным в табл. 3.6. Таким образом, окончательная ориентация зависит как от исходного, так и от приемного прямоугольника.

Если приемный прямоугольник совпадает по размерам с исходным прямоугольником, то вывод раstra осуществляется без масштабирования. В противном случае изображение либо увеличивается, либо уменьшается. Увеличение реализуется простым повторением пикселов, что обычно ведет к заметному ухудшению качества изображения. При уменьшении изображения несколько исходных пикселов преобразуются в один пикセル приемного прямоугольника. Алгоритм преобразования зависит от текущего режима масштабирования раstra (*bitmap stretching mode*), являющегося атрибутом контекста устройства. Для изменения режима масштабирования раstra предназначена функция SetStretchBltMode:

```
int SetStretchBltMode(HDC hdc, int iStretchMode);
```

Параметр iStretchMode может принимать одно из значений, указанных в табл. 3.7.

Таблица 3.7. Режимы масштабирования растра

Константа (прежнее имя)	Код	Описание
STRETCH_ANDSCANS (BLACKONWHITE)	0x1	Значение по умолчанию. Пиксели комбинируются поразрядной логической операцией И
STRETCH_ORSCANS (WHITEONBLACK)	0x2	Пиксели комбинируются поразрядной логической операцией ИЛИ
STRETCH_DELETESCANS (COLORONCOLOR)	0x3	Сохраняется один пиксел, а остальные удаляются
STRETCH_HALFTONE (HALFTONE)	0x4	Вычисляется средний цвет по нескольким пикселям

Продолжим рассмотрение параметров функции StretchDIBits.

Параметр lpBits задает адрес массива пикселов. Обратите внимание на то, что он определен как адрес одномерного массива.

Параметру lpBitsInfo передается адрес заголовка информационного блока. Здесь он имеет тип указателя на структуру BITMAPINFO, которая определена в Win32 GDI следующим образом:

```
typedef struct tagBITMAPINFO {
    BITMAPINFOHEADER bmiHeader;
    RGBQUAD          bmiColors[1];
} BITMAPINFO;
```

Обратите внимание на то, что в этой структуре резервируется место лишь для одного элемента цветовой таблицы. Поэтому будьте осторожны при ее использовании в каком-то другом контексте.

Параметр iUsage определяет интерпретацию цветовой таблицы и обычно принимает значение DIB_RGB_COLORS, когда таблица содержит RGB-составляющие цветов, либо DIB_PAL_COLORS, когда в таблице хранятся индексы логической палитры.

Последний параметр, dwRop, содержит код тернарной растровой операции. Эти операции будут рассматриваться позже, а пока мы будем использовать простейшую растровую операцию SRCCOPY, которая просто копирует пиксели источника в приемник.

Есть и еще одна функция для вывода аппаратно-независимых растров — SetDIBitsToDevice. Она обеспечивает вывод DIB-растра полностью или частично, с сохранением исходной ориентации и масштаба. Можно было бы сказать, что возможности рассмотренной выше функции StretchDIBits полностью перекрывают возможности SetDIBitsToDevice. Однако интерфейс последней функции позволяет копировать исходный растр частями, строка за строкой, что позволяет значительно экономить оперативную память, если это необходимо. Правила использования этой функции приведены в MSDN.

Примеры работы с растром DIB

В этом разделе приводятся два приложения, демонстрирующие технику работы с аппаратно-независимыми растрами. В первом приложении растр формируется программно, после чего сохраняется в BMP-файле. Во втором приложении растр загружается из BMP-файла, а затем воспроизводится в окне программы.

Подробности работы с DIB инкапсулированы в класс C++, который мы назвали KDib. Этот класс предназначен для работы только с несжатыми растрами.

В него включен тот минимум средств, которого хватает для решения рассматриваемых двух задач. Конечно, при желании можно легко расширить класс **KDib**, добавив поля и методы, необходимые для решения других подзадач.

Генератор BMP-растра для ANSI-символа

Программа, приведенная в листинге 3.5, осуществляет генерацию битового образа ANSI-символа, заданного десятичным кодом, после чего сохраняет сгенерированный образ символа в BMP-файле. Битовый образ ANSI-символа формируется в формате True Color с кодировкой 24 бит/пиксел. Цветовая таблица в растре отсутствует.

В программе используется шрифт, содержащийся в контексте устройства по умолчанию, то есть **SYSTEM_FONT**. Конечно, в будущем вы можете усложнить код программы, чтобы обеспечить выбор любого шрифта и любого размера символов, а также формировать не отдельный символ, а полный набор из 256 символов¹.

Код класса **KDib**, как обычно, размещается в двух файлах. Интерфейс содержится в файле **KDib.h**, а его реализация — в файле **KDib.cpp**.

Листинг 3.5. Проект CreateCharBmp

```
//////////  
// KDib.h  
#include <windows.h>  
#include <fstream>  
#include <string>  
using namespace std;  
  
class KDib {  
public:  
    KDib();  
    ~KDib();  
    BOOL CreateDib24(int w, int h, const char* fileName);  
    void StoreDib24();  
    BOOL LoadFromFile(const char* fileName);  
    void SetPixel(int x, int y, COLORREF color);  
    int Draw(HDC hdc, int xDst, int yDst, int wDst, int hDst,  
            int xSrc, int ySrc, int wSrc, int hSrc, DWORD rop);  
    int GetWidth() { return width; }  
    int GetHeight() { return height; }  
    const char* GetError() { return error.c_str(); }  
  
private:  
    int width;  
    int height;  
    int bytesPerLine;  
    BITMAPFILEHEADER fileHead; // заголовок растрового файла  
    BITMAPINFOHEADER infoHead; // заголовок информационного блока  
    BITMAPINFOHEADER* pInfoHead;  
    BYTE* aBitmapBits; // массив пикселов  
    int fileHeadSize;  
    int infoHeadSize;
```

¹ Для создания более продвинутой программы желательно сначала ознакомиться с главами, посвященными работе с меню и диалоговыми окнами.

```
    int imageSize;
    string error;

    ifstream inpFile;
    ofstream outFile;
};

// Kdib.cpp
#include "Kdib.h"

Kdib::Kdib() {
    fileHeadSize = sizeof(BITMAPFILEHEADER);
    fileHead.bfType = 0x4d42;
    aBitmapBits = NULL;
}

Kdib::~Kdib() {
    if (pInfoHead) delete [] pInfoHead;
    if (aBitmapBits) delete [] aBitmapBits;
    if (outFile) outFile.close();
}

BOOL Kdib::CreateDib24(int w, int h, const char* fileName) {
    width = w;
    height = h;
    bytesPerLine = ((width * 24 + 31) / 32) * 4;
    imageSize = bytesPerLine * height;

    infoHeadSize = sizeof(BITMAPINFOHEADER);
    fileHead.bfSize = fileHeadSize + infoHeadSize + bytesPerLine * height;
    fileHead.bfOffBits = fileHeadSize + infoHeadSize;

    infoHead.biSize = infoHeadSize;
    infoHead.biWidth = width;
    infoHead.biHeight = height;
    infoHead.biPlanes = 1;
    infoHead.biBitCount = 24;
    infoHead.biCompression = BI_RGB;
    infoHead.biSizeImage = imageSize;

    aBitmapBits = new BYTE[imageSize];
    memset(aBitmapBits, 0, imageSize);

    outFile.open(fileName, ios::out | ios::binary | ios::trunc);
    if (!outFile) return FALSE;
    else return TRUE;
}

BOOL Kdib::LoadFromFile(const char* fileName) {
    inpFile.open(fileName, ios::in | ios::binary);
    if (!inpFile) {
        error = "Неверное имя файла или каталога.";
        return FALSE;
    }
    inpFile.read((char*)&fileHead, fileHeadSize);
    if (fileHead.bfType != 0x4d42) {
        error = "Это не BMP-файл";
```

Листинг 3.5 (продолжение)

```

        return FALSE;
    }
    infoHeadSize = fileHead.bfOffBits - fileHeadSize;
    int fileSize = fileHead.bfSize;
    imageSize = fileSize - (fileHeadSize + infoHeadSize);

    pInfoHead = (BITMAPINFOHEADER*)(new BYTE [infoHeadSize]);
    inpFile.read((char*)pInfoHead, infoHeadSize);

    width = pInfoHead->biWidth;
    height = pInfoHead->biHeight;

    aBitmapBits = new BYTE[imageSize];
    inpFile.read((char*)aBitmapBits, imageSize);
    return true;
}

//=====
int KDib::Draw(HDC hdc, int xDst, int yDst, int wDst, int hDst,
               int xSrc, int ySrc, int wSrc, int hSrc, DWORD rop) {
    return StretchDIBits(hdc, xDst, yDst, wDst, hDst, xSrc, ySrc, wSrc, hSrc,
                         aBitmapBits, (CONST BITMAPINFO*)pInfoHead, DIB_RGB_COLORS, rop);
}

//=====
void KDib::SetPixel(int x, int y, COLORREF color) {
    int row = y;
    int col = 3 * x;

    aBitmapBits[row*bytesPerLine + col] = GetBValue(color);
    aBitmapBits[row*bytesPerLine + col+1] = GetGValue(color);
    aBitmapBits[row*bytesPerLine + col+2] = GetRValue(color);
}

//=====
void KDib::StoreDib24() {
    // Запись заголовка BMP-файла
    outFile.write((char*)&fileHead, fileHeadSize);
    outFile.write((char*)&infoHead, infoHeadSize);
    // Запись массива пикселов
    outFile.write((char*)aBitmapBits, imageSize);
}

///////////////////////////////
// CreateCharBmp.cpp
#include <windows.h>
#include "KWnd.h"
#include "KDib.h"

#define ANSI_CODE 65
#define FILE_NAME "symbol.bmp"

KDib bmp;

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;

```

```
KWnd mainWnd("CreateCharBmp", hInstance, nCmdShow, WndProc);

while (GetMessage(&msg, NULL, 0, 0)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
return msg.wParam;
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    TEXTMETRIC tm;
    SIZE sz;
    static int width, height;
    COLORREF color;
    int x, y;
    BOOL isFileCreated;
    static char line[2];

    switch (uMsg)
    {

        case WM_CREATE:
            hDC = GetDC(hWnd);
            GetTextMetrics(hDC, &tm);
            line[0] = ANSI_CODE;
            line[1] = 0;
            GetTextExtentPoint32(hDC, line, 1, &sz);
            width = sz.cx;
            height = tm.tmHeight;

            isFileCreated = bmp.CreateDib24(width, height, FILE_NAME);
            if (!isFileCreated)
                MessageBox(hWnd, "Файл \"FILE_NAME\" не создан.", "Error", MB_OK);

            ReleaseDC(hWnd, hDC);
            break;

        case WM_PAINT:
            hDC = BeginPaint(hWnd, &ps);
            SetBkColor(hDC, RGB(0,0,0));
            SetTextColor(hDC, RGB(255,255,255));
            TextOut(hDC, 0, 0, line, 1);
            // Сканирование изображения символа (от последней строки к первой)
            // Запись в массив пикселов DIB
            for (y = 0; y < height; ++y) {
                for (x = 0; x < width; ++x) {
                    color = GetPixel(hDC, x, height-1-y);
                    bmp.SetPixel(x, y, color);
                }
            }
            bmp.StoreDib24();
            EndPaint(hWnd, &ps);
    }
}
```

продолжение ↗

Листинг 3.5 (продолжение)

```

break;

case WM_DESTROY:
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
/////////////////////////////////////////////////////////////////

```

Конечно, требуется сделать несколько пояснений по интерфейсу и реализации класса KDib. Мы надеемся, что большая часть кода понятна благодаря тому, что вы внимательно прочли теоретический материал по DIB.

Такие поля класса, как width, height, bytesPerLine, fileHead, infoHead и aBitmapBits, не требуют особых комментариев. Но для чего нужен указатель на BITMAPINFOHEADER, который хранится в поле pInfoHead? Чуть позже, рассматривая метод LoadFromFile, мы увидим, что это поле используется для хранения адреса блока памяти, выделяемого из кучи при помощи оператора new. Выделенный блок памяти предназначен для заголовка информационного блока и всего того, что может следовать за ним, например цветовых масок и/или цветовой таблицы. Дело в том, что формат загружаемого файла может быть любым, и поэтому заранее неизвестно, сколько памяти потребуется для указанных компонентов.

Кроме традиционных для класса методов, таких как конструктор и деструктор, класс KDib содержит методы, перечисленные в следующем списке:

- ❑ Метод CreateDib24 создает в памяти 24-разрядный растр с заданными шириной и высотой. В процессе создания осуществляется инициализация полей заголовка растрового файла и заголовка информационного блока. После этого выделяется память для хранения массива пикселов. В завершение открывается дисковый файл с заданным именем, предназначенный для сохранения в нем создаваемого растра.
- ❑ Метод SetPixel осуществляет прямой доступ к пикселям в 24-разрядном растре. Цветовая информация для синей, зеленой и красной составляющих цвета пикселя записывается в соответствующие элементы массива aBitmapBits. Адреса элементов вычисляются через номер строки развертки y и номер пикселя в строке x.
- ❑ Метод StoreDib24 сохраняет созданный растр, записывая его на диск в формате BMP.
- ❑ Метод LoadFromFile загружает BMP-файл с заданным именем, считывая его с диска в оперативную память. В процессе загрузки сначала читается заголовок растрового файла, затем вычисляются размер заголовка информационного блока в совокупности с цветовой таблицей, который указывается в поле infoHeadSize. Помимо этого в поле imageSize указывается вычисленный размер изображения в байтах. Полученные величины используются для указания размеров динамически выделяемой памяти под заголовок информационного блока pInfoHead и под массив пикселов aBitmapBits. После выделения указанной памяти соответствующие компоненты растра читаются из файла.
- ❑ Метод Draw выводит растр на поверхность графического устройства, используя функцию StretchDIBits.

Теперь можно перейти к рассмотрению модуля CreateCharBmp.cpp. Обратите внимание на то, что в глобальной области видимости имеются следующие объявления:

- ❑ Объект bmp класса KDib.
- ❑ Макрос **ANSI_CODE**, определяющий десятичный код ANSI-символа. Значение 65 соответствует прописной букве «А» английского алфавита.
- ❑ Макрос **FILE_NAME**, задающий имя BMP-файла, сохраняемого в текущем каталоге программы.

Разбирая код оконной процедуры WndProc, обратите внимание на следующие ее особенности:

- ❑ В блоке обработки сообщения **WM_CREATE** текстовые метрики текущего шрифта читаются в переменную **tm** при помощи функции **GetTextMetrics**. Из всех метрик потребуется только высота символа, которая запоминается в переменной **height**. После этого формируется С-строка **line**, предназначенная для последующего отображения в клиентской области окна программы. В первый байт строки записывается код **ANSI_CODE**, во второй — нуль.
- ❑ Теперь можно определить ширину символа, заданного кодом **ANSI_CODE**. Для этого вызывается функция **GetTextExtentPoint32** с передачей ей в качестве второго аргумента строки **line**. Искомая ширина запоминается в переменной **width**. В завершение при помощи метода **bmp.CreateDib24** создается DIB-растр в памяти приложения.
- ❑ В блоке обработки сообщения **WM_PAINT** устанавливаются белый цвет текста и черный цвет фона графических элементов. Затем вызывается функция **TextOut**, которая отвечает за прорисовку нужного символа.
- ❑ После этого изображение символа построчно сканируется от последней строки к первой при помощи функции **GetPixel**. Получаемый цвет пикселя заносится в растр вызовом метода **bmp.SetPixel**. Заполненный растр сохраняется на диске вызовом метода **bmp.StoreDib24**.

Теперь проверьте работу приложения, откомпилировав и запустив проект¹. Программа должна создать файл **symbol.bmp**, содержащий изображение, показанное на рис. 3.6. Изображение на рисунке увеличено в 8 раз при помощи редактора MS PAINT. Сетка на рисунке также нанесена инструментами этого редактора.

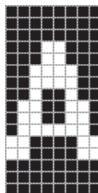


Рис. 3.6. Изображение из файла **symbol.bmp**

В рассмотренной программе используются не все методы класса KDib. Так, методы **LoadFromFile** и **Draw** будут применены только во втором примере.

¹ Не забудьте добавить к проекту файлы **KWnd.h** и **KWnd.cpp**, код которых приведен в листинге 1.2.

Просмотрщик BMP-файла

Программа, приведенная в листинге 3.6, является примитивным просмотрщиком BMP-файлов. Спектр ее возможностей действительно не слишком велик. Она может воспроизводить только несжатые файлы. Это раз. У ее окна нет полос прокрутки, и поэтому приложение может показывать только те файлы, которые помещаются на экране дисплея. Это два. И наконец, имя файла задается при помощи макроса в тексте программы, что говорит о сугубо учебном характере этого проекта.

В программе используется тот же класс `KDib`, с которым вы познакомились в предыдущем приложении. Естественно, код этого класса не будет приводиться повторно. Для сборки приложения вам просто нужно взять файлы `KDib.h` и `KDib.cpp` из предыдущего проекта и подключить их к данному проекту.

Листинг 3.6. Проект BmpFileViewer

```
///////////
// BmpFileViewer.cpp
#include <windows.h>
#include "KWnd.h"
#include "KDib.h"

#define FILE_NAME "YoungHacker.bmp"

KDib bmp;

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====================================================
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("BmpFileViewer", hInstance, nCmdShow, WndProc);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

//=====================================================
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    RECT rect;
    int dX, dY;
    int ws, hs, wd, hd;
    static BOOL isFileLoaded;

    switch (uMsg)
    {
    case WM_CREATE:
        hDC = GetDC(hWnd);
        isFileLoaded = bmp.LoadFromFile(FILE_NAME);
```

```
if (!isFileLoaded) {
    MessageBox(hWnd, "Файл \" FILE_NAME \" не загружен.", "Error",
               MB_OK);
    MessageBox(hWnd, bmp.GetError(), "Error", MB_OK);
    break;
}
// Подогнать размеры окна программы под размер растра bmp
GetClientRect(hWnd, &rect);
dX = bmp.GetWidth() - rect.right;
dY = bmp.GetHeight() - rect.bottom;
GetWindowRect(hWnd, &rect);
InflateRect(&rect, dX/2, dY/2);
MoveWindow(hWnd, rect.left, rect.top,
           rect.right-rect.left, rect.bottom-rect.top, TRUE);

ReleaseDC(hWnd, hDC);
break;

case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);

    wd = ws = bmp.GetWidth();
    hd = hs = bmp.GetHeight();
    bmp.Draw(hDC, 0, 0, wd, hd, 0, 0, ws, hs, SRCCOPY);

    EndPaint(hWnd, &ps);
    break;

case WM_DESTROY:
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
////////////////////////////////////////////////////////////////
```

Возможно, первое, на что вы обратили внимание, — это макрос, задающий имя BMP-файла, предназначенного для загрузки и отображения в окне программы. В данном случае файл называется *YoungHacker.bmp*. Но где же его взять? Можно заглянуть на сайт издательства «Питер» www.piter.com и найти там листинги примеров к данной книге. В соответствующем каталоге нетрудно отыскать и этот графический файл. Кроме того, можно выбрать любой другой BMP-файл, поместить его в каталог проекта и записать его имя в указанном макросе.

В остальном код приложения достаточно прозрачен. Работающая программа выводит на экран изображение, показанное на рис. 3.7.

Следует обратить внимание на аргументы метода *bmp.Draw*, который реализован при помощи функции *StretchDIBits*. Значения аргументов определяют, что изображение выводится в своем исходном размере (без масштабирования) и в своей исходной ориентации.

Давайте проведем следующий эксперимент для иллюстрации возможностей функции *StretchDIBits*. Нужно внести небольшие изменения в оконную процедуру. Добавьте в нее определения следующих переменных:

```
int xd0, yd0, xd1, yd1, xd2, yd2, xd3, yd3;
```



Рис. 3.7. Вывод раstra из файла YoungHacker.bmp при помощи программы BmpFileViewer



Рис. 3.8. Результат работы модифицированной программы BmpFileViewer

А блок обработки сообщения WM_PAINT замените новым фрагментом кода:

```
case WM_PAINT:  
    hdc = BeginPaint(hWnd, &ps);  
    ws = bmp.GetWidth();  
    hs = bmp.GetHeight();  
    wd = ws/2 - 4;  
    hd = hs/2 - 4;  
    xd0 = 2;      yd0 = 2;  
    xd1 = ws/2 + 2; yd1 = 2;  
    xd2 = 2;      yd2 = hs/2 + 2;  
    xd3 = ws/2 + 2; yd3 = hs/2 + 2;  
    SetStretchBltMode(hdc, STRETCH_HALFTONE);  
    bmp.Draw(hdc, xd0, yd0, wd, hd, 0, -ws, hs, SRCCOPY);  
    bmp.Draw(hdc, xd1, yd1, wd, hd, 0, 0, ws, hs, SRCCOPY);  
    bmp.Draw(hdc, xd2, yd2, wd, hd, ws, hs, -ws, -hs, SRCCOPY);  
    bmp.Draw(hdc, xd3, yd3, wd, hd, 0, hs, ws, -hs, SRCCOPY);  
    EndPaint(hWnd, &ps);  
    return 0;
```

Теперь программа выведет картинку, показанную на рис. 3.8.

Аппаратно-зависимые растры

Аппаратно-зависимый растр (DDB) представляет собой объект GDI, который функционирует под управлением GDI и драйверов устройств. Он обладает тем же статусом, что и объекты логического пера или логической кисти. При создании DDB-растра Windows определяет его внутренний формат и выделяет память из области памяти GDI. После этого все операции с растром выполняются через дескриптор объекта GDI, имеющий тип HBITMAP. DDB-растры также часто называют «растровыми объектами GDI». Напомним, что особенностью формата DDB является то, что система всегда создает битовый образ в памяти с учетом характеристик конкретного графического оборудования.

Создание DDB-растра в программе

Win32 GDI предлагает следующие функции для создания растровых объектов GDI:

```
HBITMAP CreateBitmap(int nWidth, int nHeight, UINT cPlanes,  
                      UINT cBitsPerPel, CONST VOID* lpvBits);  
HBITMAP CreateBitmapIndirect(CONST BITMAP* lpbm);  
HBITMAP CreateCompatibleBitmap(HDC hdc, int nWidth, int nHeight);
```

Функция `CreateBitmap` создает растр DDB с заданными шириной `nWidth`, высотой `nHeight`, количеством плоскостей `cPlanes`, количеством битов на пиксель `cBitsPerPel`, после чего инициализирует массив пикселов данными из массива `lpvBits`. Параметры `cPlanes` и `cBitsPerPel` определяют лишь минимальные требования к раству. В свою очередь, GDI поручает драйверу устройства выбор ближайшего доступного формата с кодировкой по крайней мере `cPlanes × cBitsPerPel` бит/пиксель.

Функция `CreateBitmapIndirect` делает практически то же самое, но параметры растра передаются ей через структуру типа `BITMAP`:

```
typedef struct tagBITMAP {  
    LONG   bmType;        // тип растра (поле должно быть равно 0)  
    LONG   bmWidth;       // ширина растра в пикселях  
    LONG   bmHeight;      // высота растра в пикселях
```

```

    LONG    bmWidthBytes; // число байтов на строку развертки
            // (значение должно быть четным)
    WORD    bmPlanes;   // число цветовых плоскостей
    WORD    bmBitsPixel; // число битов на пикセル
    LPVOID bmBits;     // указатель на массив битов с данными инициализации
} BITMAP;

```

К сожалению, обе функции имеют существенные ограничения, влияющие на их практическое применение. Они позволяют создавать либо монохромный растр, инициализированный данными из массива, либо цветной растр, но без инициализации. При создании растра без инициализации параметр `lpvBits` или поле `lpbm.bmBits` должны иметь значение `NULL`. Попытка создать цветной растр, инициализированный данными из конкретного массива, как правило, заканчивается неудачей, так как созданный битовый образ оказывается несовместимым с конкретным графическим устройством.

Функция `CreateCompatibleBitmap` предназначена для создания DDB-растра с заданными шириной и высотой, который заведомо совместим с контекстом устройства `hdc`. Но битовый образ, создаваемый этой функцией, также является неинициализированным.

Создание DDB-растра из упакованного DIB

Итак, рассмотренные выше функции мало пригодны для создания инициализированных цветных DDB-растров. В то же время аппаратно-независимый растр обладает хорошими средствами для описания стандартных цветовых форматов. Поэтому GDI содержит функцию `CreateDIBitmap`, которая создает инициализированный DDB-растр на базе DIB, то есть, по существу, преобразует упакованный DIB в DDB. Функция имеет следующий прототип:

```

HBITMAP CreateDIBitmap(HDC hdc,      CONST BITMAPINFOHEADER* lpbmih,
                      DWORD fdwInit, CONST VOID* lpbInit, CONST BITMAPINFO* lpbm,
                      UINT fuUsage);

```

В своей реализации она сначала создает DDB-растр, совместимый с контекстом устройства `hdc`, а затем преобразует DIB в DDB. Параметр `lpbmih` содержит адрес заголовка информационного блока DIB. Параметр `fdwInit` определяет необходимость инициализации DDB-растра. Если он равен `CBM_INIT`, то следующие три параметра ссылаются на упакованный DIB-растр, используемый для инициализации. Параметр `lpbInit` указывает на массив пикселов, а параметр `lpbm` — на заголовок информационного блока DIB, преобразованный к типу `BITMAPINFO`. Последний параметр, `fuUsage`, сообщает, содержит цветовая таблица индексы палитры (`DIB_PAL_COLORS`) или цвета RGB (`DIB_RGB_COLORS`).

Честно говоря, не совсем понятно, для чего нужна эта функция, поскольку если DIB-растр загружен, как, например, в коде листинга 3.6, то никаких проблем с его выводом не будет. Для этого вызывается функция `StretchDIBits` (в классе `KDib` она инкапсулирована в метод `Draw`). Функция `CreateDIBitmap` позволяет преобразовать этот же DIB-растр, предварительно загруженный в память, в объект DDB-растра, который затем может быть отображен при помощи функции `BitBlt` (подробности вывода DDB-растров рассматриваются ниже). Но остается вопрос: а что дает эта более сложная схема?

Все же основные способы создания DDB-растров сводятся к рассматриваемым ниже двум функциям.

Загрузка DDB-растра из ресурсов приложения

В Windows-программировании DIB-растры (BMP-файлы) часто присоединяются к модулю в виде ресурса¹, а затем загружаются в программу функцией LoadBitmap, которая преобразует растр к типу HBITMAP:

```
HBITMAP LoadBitmap(  
    HINSTANCE hInstance, // дескриптор экземпляра приложения  
    LPCTSTR lpBitmapName // имя растрового ресурса  
);
```

Как показано в главе 5, имя ресурса может быть либо строкой с завершающим нуль-символом, например szBmpName1, либо целочисленным идентификатором, например IDB_1. В первом случае параметру lpBitmapName передается адрес строки szBmpName1, а во втором — выражение, преобразующее целочисленный идентификатор к типу LPCTSTR при помощи макроса MAKEINTRESOURCE, то есть параметр получает значение MAKEINTRESOURCE(IDB_1).

Растровые ресурсы хранятся в модулях Win32 в формате упакованного DIB-растра. Функция LoadBitmap находит растровый ресурс, загружает его в память, получает дескриптор упакованного DIB-растра и затем на его базе создает DDB-растр, совместимый с текущим экранным режимом.

Пример использования функции LoadBitmap можно найти в листинге 5.4 и в листинге 12.1.

Приложение может также вызвать функцию LoadBitmap для загрузки предопределенных в системе растров с изображениями различных кнопок, которые использует Win32 API. Для этого необходимо передать параметру hInstance значение NULL, а второму параметру — одно из значений OBM_BTNCORNERS, OBM_BTFSIZE и других аналогичных констант.

Загрузка DDB-растра из BMP-файла

Если по какой-то причине вам удобно загружать битовый образ не из ресурса, а непосредственно из BMP-файла, то можно воспользоваться многопроцессорной функцией LoadImage, предназначеннной для загрузки пиктограмм, курсоров и растров. Функция имеет следующий прототип:

```
HANDLE LoadImage(  
    HINSTANCE hinst, // дескриптор экземпляра приложения  
    LPCTSTR lpszName, // имя или идентификатор изображения  
    UINT uType, // тип изображения  
    int cxDesired, // ширина изображения  
    int cyDesired, // высота изображения  
    UINT fuLoad // флаги загрузки  
);
```

Имя загружаемого растра передается параметру lpszName, при этом параметр fuLoad должен иметь значение LR_LOADFROMFILE, а параметр uType — значение IMAGE_BITMAP. Установка нулевых значений для параметров cxDesired и cyDesired приводит к использованию текущих размеров загружаемого растра.

В случае успешного завершения функция возвращает дескриптор полученного DDB-растра. Если загрузить изображение не удалось, то возвращается значение NULL.

Пример использования функции LoadImage приведен в листинге 3.7.

¹ Более подробно ресурсы Windows-приложения рассматриваются в главе 5.

Получение информации о графическом объекте

После создания или загрузки изображения в формате DDB приложению необходимо узнать его размеры, чтобы использовать их для отображения раstra. Для решения этой проблемы можно вызвать функцию `GetObject`, имеющую следующий прототип:

```
int GetObject(
    HGDIOBJ hgdiobj, // дескриптор графического объекта
    int cbBuffer, // размер буфера для информации об объекте
    LPVOID lpvObject // адрес буфера для информации об объекте
);
```

Функция является универсальной, поэтому в параметре `hgdiobj` может быть указан дескриптор одного из таких объектов, как перо, кисть, шрифт, палитра или растр типа `HBITMAP`. Информацию об объекте функция помещает в буфер с адресом `lpvObject`. В табл. 3.7 показано, каким должен быть тип буфера в зависимости от типа объекта.

Таблица 3.7. Тип буфера, на который указывает параметр `lpvObject`

Тип объекта	Тип буфера
<code>HBITMAP</code>	<code>BITMAP</code>
<code>HBITMAP</code> , возвращенный функцией <code>CreateDIBSection</code>	<code>DIBSECTION</code> , если поле <code>cbBuffer</code> установлено в значение <code>sizeof(DIBSECTION)</code> , или <code>BITMAP</code> , если поле <code>cbBuffer</code> установлено в значение <code>sizeof(BITMAP)</code>
<code>HPALETTE</code>	<code>WORD</code> — количество цветов в логической палитре
<code>HPEN</code>	<code>LOGPEN</code>
<code>HPEN</code> , возвращенный функцией <code>ExtCreatePen</code>	<code>EXTLOGPEN</code>
<code>HBRUSH</code>	<code>LOGBRUSH</code>
<code>HFONT</code>	<code>LOGFONT</code>

Если при вызове функции параметр `lpvObject` равен `NULL`, то функция возвращает количество байтов, которое необходимо зарезервировать в буфере `lpvObject` для размещения информации об объекте.

Если параметр `lpvObject` не равен `NULL`, то при успешном завершении функция возвращает количество сохраненных в буфере байтов. При возникновении ошибки функция возвращает нулевое значение.

Когда функция `GetObject` применяется к объекту типа `HBITMAP`, то возвращаемая информация содержит данные только о ширине и высоте раstra, а также о цветовом формате изображения. Поле `bmBits` структуры `BITMAP` при этом равно нулю.

Отображение DDB-растра

Несмотря на то что DDB-растры играют важную роль в Windows-программировании, в Win32 API нет функции для непосредственного отображения DDB. В GDI задача отображения DDB решается обобщенно, как задача пересылки прямоугольного массива пикселов с одного графического устройства (*устройства-источника*) на другое графическое устройство (*устройство-приемник*).

При отображении раstra устройством-приемником обычно является экран дисплея, и его представителем выступает контекст дисплея.

А что в этом случае является устройством-источником? В роли этого устройства выступает виртуальное графическое устройство, имитируемое в памяти в виде растра. Для его представления в GDI предусмотрен контекст устройства в памяти (или совместимый контекст). В главе 2 была дана краткая характеристика этого вида контекста устройства. Напомним, что совместимый контекст создается следующей функцией:

```
HDC CreateCompatibleDC (HDC hdc);
```

где `hdc` – дескриптор существующего контекста устройства, с которым должен быть совместим создаваемый контекст. Функция `CreateCompatibleDC` может использоватьсь только с устройствами, поддерживающими растровые операции. Чаще всего `hdc` является дескриптором контекста дисплея, который относится к клиентской области окна приложения. Если этот параметр имеет значение `NULL`, то функция создает контекст в памяти, совместимый со всем экраном дисплея. Если функция завершается успешно, то она возвращает дескриптор совместимого контекста устройства.

Когда совместимый контекст устройства создан, с ним по умолчанию связан монохромный растр из одного пикселя. Поэтому, прежде чем использовать этот контекст, необходимо выбрать в него DDB-растр при помощи функции `SelectObject`.

Типичная схема отображения DDB-растра с дескриптором `hBitmap` на графическое устройство с дескриптором контекста устройства `hDC` выглядит следующим образом:

```
// Определить объект совместимого контекста
HDC hMemDC;
// Создать совместимый контекст
hMemDC = CreateCompatibleDC(hDC);
// Выбрать DDB-растр в совместимый контекст
SelectObject(hMemDC, hBitmap);
// Скопировать изображение из hMemDC в hDC
BitBlt(hDC, 0, 0, bmpWidth, bmpHeight, hMemDC, 0, 0, SRCCOPY);
// Удалить совместимый контекст
DeleteDC(hMemDC);
```

Очевидно, что устройство-источник и устройство-приемник в рассмотренной схеме можно поменять местами. В этом случае будет осуществляться копирование изображения с устройства `hDC` на устройство `hMemDC`, то есть в DDB-растр.

Функции блиттинга

Для поддержки блиттинга¹ GDI использует функции `BitBlt` и `StretchBlt`. Рассмотрим подробнее первую из них.

Название функции `BitBlt` происходит от слов «bit block transfer», означающих «перенос битового блока». На самом деле функция `BitBlt` позволяет делать намного больше, чем просто перенос блока пикселов. Эта функция может выполнить одну из 256 логических растровых операций над тремя наборами пикселов. Прототип функции имеет следующий вид:

```
BOOL BitBlt(HDC hdcDest, int nXDest, int nYDest, int nWidth, int nHeight,
             HDC hdcSrc, int nXSrc, int nYSrc, DWORD dwRop);
```

Функция `BitBlt` передает прямоугольный блок пикселов с поверхности устройства-источника `hdcSrc` на поверхность устройства-приемника `hdcDest`. Исходный прямоугольник определяется параметрами `nXSrc`, `nYSrc`, `nWidth`, `nHeight` в логической

¹ В компьютерной графике **блиттингом** называют копирование изображений в память видеодрайвера.

системе координат контекста устройства-источника. Приемный прямоугольник определяется параметрами `nXDest`, `nYDest`, `nWidth`, `nHeight` в логической системе координат контекста устройства-приемника. Оба контекста устройства должны поддерживать растровые операции `RC_BITBLT`.

Из списка параметров функции видно, что ширина и высота исходного и приемного прямоугольников задаются одними и теми же параметрами — `nWidth` и `nHeight`. Поэтому если в устройстве-приемнике и в устройстве-источнике установлен режим отображения по умолчанию `MM_TEXT`, в котором логические единицы совпадают с физическими единицами, то перенос блока пикселов происходит без масштабирования.

В других режимах отображения исходный и приемный прямоугольники могут иметь разные размеры в системах координат устройства. В таком случае исходное изображение масштабируется по размерам приемного прямоугольника.

Контексты устройства-источника и устройства-приемника не обязательно должны быть разными. Функцию `BitBlt` можно применять и для переноса блока пикселов в пределах той же поверхности, на которой находится исходный прямоугольник.

Последний параметр `dwRop` задает тернарную растровую операцию, то есть способ объединения исходного пикселя, приемного пикселя и кисти для формирования нового значения приемного пикселя. Эти операции будут рассмотрены позже, а пока что будет использоваться простейшая растровая операция `SRCCOPY`, которая просто заменяет пиксели приемника значениями пикселов источника.

Вторая функция блиттинга `StretchBlt` имеет следующий прототип:

```
BOOL StretchBlt(HDC hdcDest, int nXDest, int nYDest, int nWidthDest,
    int nHeightDest, HDC hdcSrc, int nXSrc, int nYSrc, int nWidthSrc,
    int nHeightSrc, DWORD dwRop);
```

Она делает практически то же самое, что и функция `BitBlt`, но в списке параметров размеры исходного и приемного прямоугольников определяются независимо друг от друга. Это позволяет задавать масштабирование при переносе блока пикселов, когда исходный и приемный прямоугольники имеют разные размеры в логических координатах. Еще одно различие — функция может осуществлять зеркальные отражения изображения так же, как это делает функция `StretchDIBits`, если параметры исходного (или приемного) прямоугольника задавать по правилам, приведенным в табл. 3.6.

Пример работы с DDB

Программа, приведенная в листинге 3.7, осуществляет загрузку DDB-растра из BMP-файла, после чего отображает рисунок в своем окне. По существу, здесь решается та же задача, что и в программе `BmpFileViewer` (см. листинг 3.6), но другими средствами.

Листинг 3.7. Проект `BmpFileViewer2`

```
///////////
// BmpFileViewer2.cpp
#include <windows.h>
#include "KWnd.h"

#define FILE_NAME "YoungHacker.bmp"

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
```

```
//=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("BmpFileViewer2", hInstance, nCmdShow, WndProc);
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    RECT rect;
    static int bmpWidth, bmpHeight;
    static HBITMAP hBitmap; // дескриптор битового образа
    HINSTANCE hInst;
    HDC hMemDC; // дескриптор совместимого контекста
    BITMAP bm;
    int dx, dy;

    switch (uMsg)
    {
    case WM_CREATE:
        hDC = GetDC(hWnd);
        hInst = (HINSTANCE)GetClassLong(hWnd, GCL_HMODULE);
        hBitmap = (HBITMAP)LoadImage(hInst, FILE_NAME, IMAGE_BITMAP, 0, 0,
            LR_LOADFROMFILE);
        if (hBitmap == NULL) {
            MessageBox(hWnd, "Файл " FILE_NAME " не загружен", "Error", MB_OK);
            break;
        }
        GetObject(hBitmap, sizeof(bm), &bm);
        bmpWidth = bm.bmWidth;
        bmpHeight = bm.bmHeight;

        // Подогнать размеры окна программы под размер растра hBitmap
        GetClientRect(hWnd, &rect);
        dX = bmpWidth - rect.right;
        dY = bmpHeight - rect.bottom;
        GetWindowRect(hWnd, &rect);
        InflateRect(&rect, dX/2, dY/2);
        MoveWindow(hWnd, rect.left, rect.top,
            rect.right-rect.left, rect.bottom-rect.top, TRUE);
        ReleaseDC(hWnd, hDC);
        break;

    case WM_PAINT:
        hDC = BeginPaint(hWnd, &ps);

        // Создать совместимый контекст
        hMemDC = CreateCompatibleDC(hDC);
```

продолжение ↗

Листинг 3.7. (продолжение)

```

// Выбрать hBitmap в совместимый контекст
SelectObject(hMemDC, hBitmap);
// Скопировать изображение из hMemDC в hDC
BitBlt(hDC, 0, 0, bmpWidth, bmpHeight, hMemDC, 0, 0, SRCCOPY);

DeleteDC(hMemDC);
EndPaint(hWnd, &ps);
break;

case WM_DESTROY:
PostQuitMessage(0);
break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
/////////////////////////////////////////////////////////////////

```

Следует обратить внимание на несколько интересных моментов. Так, DDB-растры *hBitmap* загружается в блоке обработки сообщения *WM_CREATE* при помощи функции *LoadImage*. Дескриптор экземпляра приложения *hInst*, который нужно передать в функцию *LoadImage*, определяется предварительным вызовом функции *GetClassLong(hWnd, GCL_HMODULE)*. После загрузки растра его ширина и высота находятся при помощи функции *GetObject*.

Отображение растра в клиентской области окна реализовано в блоке обработки сообщения *WM_PAINT* по традиционной схеме. Сначала создается совместимый контекст (контекст устройства в памяти) *hMemDC*, после чего в него выбирается битовый образ *hBitmap*. Затем блок пикселов копируется при помощи функции *BitBlt*. В завершение совместимый контекст удаляется.

Результат выполнения этой программы выглядит так же, как и результат выполнения программы *BmpFileViewer* (см. рис. 3.7).

DIB-секции

Каждый из рассмотренных выше типов растров обладает своими недостатками и достоинствами. Сравнительная характеристика DDB и DIB приведена в табл. 3.8.

Таблица 3.8. Сравнение растров DDB и DIB

Характеристика	DDB	DIB
Независимость от аппаратуры	Нет	Да
Рисование на растре средствами GDI	Поддерживается	Не поддерживается
Прямой доступ к массиву пикселов со стороны приложения	Отсутствует	Имеется
Максимальный размер растра	48 Мбайт (используется системная память)	Ограничен объемом виртуального адресного пространства (используется память уровня приложения)

В Win32 API появился новый тип растра, объединяющий достоинства DDB и DIB.

DIB-секцией (DIB section) называется DIB-растр, который позволяет производить непосредственное чтение и запись как со стороны приложения, так и со стороны GDI. Массив пикселов DIB-секции хранится либо в виртуальной памяти приложения, либо в файле, отображаемом на память, который в среде разработчиков операционной системы Windows называется секцией. Этим, видимо, объясняется происхождение термина «DIB-секция».

DIB-секция, как и аппаратно-зависимый растр, является объектом GDI. После ее создания GDI возвращает дескриптор DIB-секции, относящийся к типу HBITMAP. Завершив работу с DIB-секцией, приложение должно вызывать функцию DeleteObject, чтобы освободить связанные с нею ресурсы.

Создание DIB-секции

Для создания DIB-секции используется функция CreateDIBSection:

```
HBITMAP CreateDIBSection(
    HDC hdc,           // дескриптор контекста устройства
    CONST BITMAPINFO* pbmi, // адрес заголовка информационного блока
    UINT iUsage,        // тип данных в цветовой таблице
    VOID** ppvBits,     // адрес массива пикселов
    HANDLE hSection,   // дескриптор объекта файла, отображаемого на память
    DWORD dwOffset      // смещение массива пикселов внутри отображаемого файла
);
```

Основные характеристики создаваемого растра задаются в структуре типа BITMAPINFO, адрес которой передается второму параметру функции. В этой структуре необходимо указать ширину, высоту, цветовую глубину и тип сжатия. DIB-секции поддерживают только тип BI_RGB, то есть только несжатые растры. Также задаются битовые маски, если они нужны, и цветовая таблица, если она используется.

Параметр iUsage определяет тип данных, содержащихся в массиве bmiColors, который является членом структуры BITMAPINFO. Если параметр iUsage имеет значение DIB_PAL_COLORS, то цветовая таблица bmiColors содержит индексы логической палитры. Значение DIB_RGB_COLORS указывает, что цветовая таблица содержит значения в формате RGB.

Параметр ppvBits содержит адрес переменной-указателя, в который GDI заносит адрес массива пикселов DIB-секции.

Параметр hSection может быть равен NULL. В этом случае GDI выделяет память под массив пикселов из виртуальной памяти приложения, а параметр dwOffset игнорируется. Если параметр hSection не равен NULL, то он должен быть дескриптором объекта файла, отображаемого на память, который был создан вызовом функции CreateFileMapping с флагами PAGE_READWRITE или PAGE_WRITECOPY. В этом случае размещение массива пикселов осуществляется в указанном файле со смещением, определяемым параметром dwOffset.

Прямой доступ к пикселям со стороны приложения

Прямой доступ к пикселям со стороны приложения реализуется точно так же, как и при работе с DIB-растром (см. класс KDib в листинге 3.5).

Применительно к DIB-секциям аналогичный доступ реализован в классе KDibSection, определение которого будет дано в главе 12, когда он будет применяться для решения проблемы рисования в реальном времени (см. листинг 12.4).

Отображение DIB-секции

Для вывода DIB-секции на поверхность графического устройства можно использовать два подхода:

- ❑ вывод как DIB-растра при помощи функции `StretchDIBits` или `SetDIBitsToDevice`;
- ❑ вывод как растрового объекта GDI, заданного дескриптором `hBitmap`, с использованием совместимого контекста устройства и функции `BitBlt` или `StretchBlt`.

Пример использования DIB-секций приведен в приложении `ArincReceiver` (глава 12, листинг 12.4).

Тернарные растровые операции

Последний параметр, `dwRop`, в функциях `BitBlt`, `PatBlt` и `StretchBlt` задает растровую операцию, которая указывает для GDI, как должны комбинироваться биты *исходного изображения* с битами *изображения-приемника* и с битами шаблона *текущей кисти*. Результат операции определяет новое состояние изображения-приемника. Поскольку число operandов равно трем, то операция относится к классу тернарных операций.

В Win32 API существует 256 ROP-кодов (*raster-operation codes*). Растровые операции кодируются 32-разрядными двойными словами `DWORD`. В старшем слове хранится один из 256 однобайтных кодов растровой операции, а младшее слово содержит кодировку формулы, по которой растровая операция реализуется драйвером графического устройства.

Разработчики Microsoft присвоили символические имена только пятнадцати тернарным растровым операциям из 256. Видимо, это как раз те операции, которые использует операционная система. Указанные операции приведены в табл. 3.9, при этом использована следующая система обозначений: `P` — шаблон кисти (*pattern*), `S` — источник (*source*), `D` — приемник (*destination*).

Таблица 3.9. Тернарные растровые операции

Pattern:	1 1 1 1 0 0 0	Формула	ROP-код	Имя
Source:	1 1 0 0 1 1 0			
Destination:	1 0 1 0 1 0 1			
Результат:	0 0 0 0 0 0 0	$D = 0$	0x00000042	BLACKNESS
	0 0 0 1 0 0 1	$D = \sim(S \mid D)$	0x001100A6	NOTSRCERASE
	0 0 1 1 0 0 1	$D = \sim S$	0x00330008	NOTSRCCOPY
	0 1 0 0 0 1 0	$D = S \& \sim D$	0x00440328	SRCERASE
	0 1 0 1 0 1 0	$D = \sim D$	0x00550009	DSTINVERT
	0 1 0 1 1 0 0	$D = P \wedge D$	0x005A0049	PATINVERT
	0 1 1 0 0 1 0	$D = S \wedge D$	0x00660046	SRCAINVERT
	1 0 0 0 1 0 0	$D = S \& D$	0x008800C6	SRCCAND
	1 0 1 1 1 0 1	$D = \sim S \mid D$	0x00BB0226	MERGEPAINT
	1 1 0 0 0 0 0	$D = P \& S$	0x00C000CA	MERGECOPY
	1 1 0 0 1 1 0	$D = S$	0x00CC0020	SRCCOPY
	1 1 1 0 1 1 0	$D = S \mid D$	0x00EE0086	SRCPAINT
	1 1 1 1 0 0 0	$D = P$	0x00F00021	PATCOPY
	1 1 1 1 1 0 1	$D = P \mid \sim S \mid D$	0x00FB0A09	PATPAINT
	1 1 1 1 1 1 1	$D = 1$	0x00FF0062	WHITENESS

Полную таблицу тернарных растровых операций можно найти в MSDN (Platform SDK ▶ Windows GDI ▶ Ternary Raster Operations).

Следует обратить внимание на восьмиразрядные двоичные коды, являющиеся результатом каждой операции. Двухзначное шестнадцатеричное число, соответствующее этим битам, есть старшее слово ROP-кода.

При внимательном взгляде на табл. 3.9 можно заметить, что не все растровые операции зависят от всех трех operandов. Поэтому можно рассматривать отдельные подгруппы этих операций.

Инициализирующие растровые операции

Операции **BLACKNESS** и **WHITENESS** не зависят ни от шаблона кисти, ни от источника, ни от приемника. Они обычно используются для инициализации или возврата графической поверхности в исходное состояние. Черный (Black) и белый (White) цвета отличаются от других тем, что все разряды для них установлены либо в 0, либо в 1. Благодаря этому для любого произвольного цвета справедливы следующие утверждения:

```
Black AND C = Black  
Black OR C = C  
Black XOR C = C  
White AND C = C  
White OR C = White  
White XOR C = NOT C
```

Указанные свойства могут использоваться для достижения различных эффектов при выводе растров.

Операции, зависящие только от источника

К таким операциям относятся **SRCCOPY** и **NOTSRCCOPY**.

Операция **SRCCOPY** уже использовалась ранее в примерах. Она просто копирует пиксели источника в приемник. Обычно эта операция применяется для отображения растра в исходном цвете.

Операция **NOTSRCCOPY** копирует в приемник цвет, противоположный цвету источника.

Операция, зависящая только от приемника

Операция **DSTINVERT** меняет цвет пикселя приемника на противоположный.

Операции, не зависящие от источника

В табл. 3.9 к таким операциям относятся **PATINVERT** и **PATCOPY**. Всего существует 16 растровых операций, не зависящих от источника, но не все они имеют символьические имена. В GDI предусмотрена специальная функция для выполнения растровых операций, не зависящих от источника:

```
BOOL PatBlt(HDC hdc, int nXLeft, int nYLeft, int nWidth, int nHeight,  
             DWORD dwRop);
```

Функция **PatBlt** комбинирует текущую кисть с прямоугольным регионом приемника. Вы можете использовать любые коды ROP из полной таблицы операций¹, в которых не задействован источник, а не только те, которые приведены в табл. 3.9.

¹ См. выше ссылку на MSDN.

Операции, не зависящие от шаблона кисти

К таким операциям относятся NOTSRCERASE, SRCERASE, SRCINVERT, SRCAND, MERGEPAINT и SRCPAINT. Следует отметить, что операции SRCINVERT и SRCAND используются системой Windows для отображения пиктограмм и курсоров мыши.

Рассмотрим, например, механизм отображения пиктограмм, которым пользуется Windows. Ресурсы предопределенных пиктограмм хранятся в динамически загружаемой библиотеке Shell32.dll¹. Доступ к библиотеке можно получить, загрузив ее в приложение при помощи функции LoadLibrary, которая возвращает дескриптор экземпляра модуля (hShell32), содержащего соответствующие растровые изображения. Затем, используя индекс пиктограммы, можно загрузить любую пиктограмму из модуля hShell32. Для этого достаточно вызвать функцию LoadImage. Например, если передать функции LoadImage индекс 12, то будет загружена пиктограмма, обозначающая привод CD-ROM.

Функция LoadImage в случае успешного завершения возвращает дескриптор пиктограммы hIcon. Затем при помощи вызова

```
GetIconInfo(hIcon, &iconInfo);
```

можно получить информацию о загруженной пиктограмме, сохранив ее в переменной iconInfo типа ICONINFO. Для нас в этой структуре интересны два поля:

```
HBITMAP hbmMask;  
HBITMAP hbmColor;
```

Первое из этих полей содержит дескриптор растра, представляющего собой *маску пиктограммы*, второе поле — дескриптор растра, представляющего собой *изображение пиктограммы*. На рис. 3.9 показаны маска и изображение для пиктограммы «привод CD-ROM».



Рис. 3.9. Пара растров для вывода пиктограммы «привод CD-ROM»

Оба растра имеют размеры 48 × 48. Но для чего нужны эти два растра? Дело в том, что при выводе пиктограммы в виде прямоугольной области некоторые пиксели этой области должны изменяться, а некоторые должны остаться прежними. Изменяемая область называется *непрозрачной*, и в маске ей соответствует черный цвет. Остальные пиксели образуют *прозрачную область* — в маске ей соответствует белый цвет.

Приложение, код которого приведен в листинге 3.8, демонстрирует, как можно отобразить пиктограмму с учетом ее прозрачной области.

Листинг 3.8. Проект DrawIconWithRop

```
//////////  
// DrawIconWithRop.cpp  
#include <windows.h>
```

¹ Вопросы создания и применения пользовательских пиктограмм рассматриваются в главе 5.

```
#include "KWnd.h"

#define ICON_ID 12

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("Draw icon with ROP", hInstance, nCmdShow, WndProc);
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;

    static HINSTANCE hShell32;
    ICONINFO iconInfo;
    BITMAP bmp;
    HICON hIcon;
    HDC hMemDC;
    int x = 20, y = 20;

    switch (uMsg)
    {

    case WM_CREATE:
        hShell32 = LoadLibrary("Shell32.dll");
        SetClassLong(hWnd, GCL_HBRBACKGROUND,
            (LONG)CreateSolidBrush (RGB(180,140,200)));
        break;

    case WM_PAINT:
        hDC = BeginPaint(hWnd, &ps);
        hMemDC = CreateCompatibleDC(hDC);
        hIcon = (HICON)LoadImage(hShell32, MAKEINTRESOURCE(ICON_ID),
            IMAGE_ICON, 48, 48, LR_DEFAULTCOLOR);
        if (!hIcon) {
            MessageBox(hWnd, "Ошибка загрузки пиктограммы", "Error", MB_OK);
            break;
        }
        GetIconInfo(hIcon, &iconInfo);
        GetObject(iconInfo.hbmMask, sizeof(bmp), &bmp);

        // Маска
        SelectObject(hMemDC, iconInfo.hbmMask);
        BitBlt(hDC, x, y, bmp.bmWidth, bmp.bmHeight, hMemDC, 0, 0, SRCCOPY);
        // Изображение
    }
}
```

Листинг 3.8 (продолжение)

```

SelectObject(hMemDC, iconInfo.hbmColor);
BitBlt(hDC, x+60, y, bmp.bmWidth, bmp.bmHeight, hMemDC, 0, 0,
       SRCCOPY);

// Вывод пиктограммы наложением маски и рисунка
SelectObject(hMemDC, iconInfo.hbmMask);
BitBlt(hDC, x+120, y, bmp.bmWidth, bmp.bmHeight, hMemDC, 0, 0,
       SRCAND);
SelectObject(hMemDC, iconInfo.hbmColor);
BitBlt(hDC, x+120, y, bmp.bmWidth, bmp.bmHeight, hMemDC, 0, 0,
       SRCINVERT);

// Вывод пиктограммы функцией DrawIcon
DrawIcon(hDC, x+188, y, hIcon);
DeleteObject(hMemDC);
DestroyIcon(hIcon);
EndPaint(hWnd, &ps);
break;

case WM_DESTROY:
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
/////////////////////////////////////////////////////////////////

```

Для отображения пиктограммы сначала при помощи функции BitBlt с растровой операцией SRCAND выводится маска. При этом непрозрачная область устанавливается в черный цвет, а прозрачная остается без изменения. Затем в этой же позиции выводится цветной растр изображения, для чего используется вызов BitBlt с растровой операцией SRCINVERT.

Результат работы программы показан на рис. 3.10. В клиентскую область приложения последовательно выводятся: маска (*а*), цветной растр (*б*), пиктограмма (*в*), образованная наложением цветного раstra на маску, и пиктограмма (*г*), нарисованная функцией DrawIcon, которая масштабирует значок по его стандартным размерам 32 × 32 пикселя.

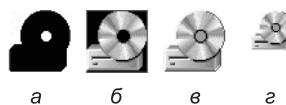


Рис. 3.10. Вывод программы DrawIconWithRop

Метафайлы

Метафайл — это протокол обращений к функциям GDI, сохраненный в двоичном формате. Метафайлы имеют такое же значение для векторной графики, как и битовые образы для растровой графики. Метафайл состоит из набора записей,

соответствующих вызовам графических функций, таких как создание и выбор в контекст устройства пера или кисти, рисование линий, фигур, вывод текста, и иных операций.

При воспроизведении метафайлов достигается такой же результат, как и при непосредственном использовании функций GDI. Разница между их воспроизведением и непосредственным вызовом функций состоит в том, что метафайлы могут храниться в памяти или в файлах на диске, загружаться и воспроизводиться приложением столько раз, сколько это нужно.

Метафайлы используются для передачи изображений между программами через буфер обмена, а также для сохранения изображений в виде файлов на диске. Для их хранения требуется значительно меньше места, чем для хранения растровых изображений. В то же время для отображения метафайлов требуется обычно больше времени, чем для вывода растровых изображений.

Поскольку метафайл описывает изображение в терминах команд графического вывода, то изображение может быть масштабировано при воспроизведении без потери разрешения. Для битовых образов масштабирование всегда связано с потерей качества изображения.

Первоначальный 16-битный формат метафайлов WMF¹ появился в Windows 2.0. Однако этот формат был аппаратно-зависимым и не содержал информацию о размерах изображения, поэтому его использование было связано со многими проблемами. В Win32 появился новый 32-битный формат EMF (Enhanced Metafile Format). Расширенные метафайлы содержат дополнительную информацию о размерах изображения и цветовой палитре, что обеспечивает их аппаратную независимость. Кроме того, они поддерживают все 32-битные функции рисования. Win32 API позволяет использовать файлы форматов WMF и EMF.

Поскольку в новых приложениях рекомендуется использовать формат EMF, то в дальнейшем будут рассматриваться только расширенные метафайлы.

Создание метафайла

Расширенный метафайл является таким же объектом GDI, как, например, объект кисти или объект DIB-секции. Объект расширенного метафайла однозначно определяется своим дескриптором типа **HENHMETAFILE**. Создание метафайла и запись в него команд GDI связаны с использованием метафайлового контекста устройства, который рассматривался во второй главе.

Метафайловый контекст устройства создается функцией **CreateEnhMetaFile**:

```
HDC CreateEnhMetaFile(HDC hdcRef, LPCTSTR lpFilename, CONST RECT* lpRect,  
LPCTSTR lpDescription);
```

Первый параметр, **hdcRef**, указывает на эталонный контекст устройства, данные которого будут использованы при записи EMF. Если этот параметр имеет значение **NULL**, то GDI принимает в качестве эталона текущий экран.

Во втором параметре, **lpFilename**, может передаваться имя файла на диске или значение **NULL**. Если передается **NULL**, то метафайл создается в памяти. Если передается имя файла, то после удаления объекта метафайла функцией **DeleteEnhMetaFile** файловый вариант сохраняется на диске.

¹ WMF – Windows Metafile Format.

Третий параметр, `lpRect`, определяет позицию и размеры сохраняемого изображения в единицах 0,01 мм. Если параметр имеет значение `NULL`, то GDI вычисляет ограничивающий прямоугольник по всем командам вывода.

Последний параметр, `lpDescription`, содержит необязательное текстовое описание, сохраняемое в метафайле. Обычно описание содержит имя приложения и имя документа, разделенные нулевым символом. Это описание должно завершаться двумя нулевыми символами.

Функция `CreateEnhMetaFile` возвращает дескриптор метафайлового контекста устройства. Этот дескриптор передается затем всем графическим функциям GDI для записи «протокола рисования» в метафайл.

Когда рисование завершено, необходимо вызвать функцию `CloseEnhMetaFile`, которая закрывает метафайловый контекст и возвращает дескриптор расширенного метафайла. Этот дескриптор может быть использован для воспроизведения метафайла.

Как говорилось выше, если имя файла было задано при вызове функции `CreateEnhMetaFile`, то сохранение файла на диске происходит после вызова функции `DeleteEnhMetaFile`, которая удаляет объект метафайла из памяти приложения.

Технология создания метафайла иллюстрируется приложением, код которого приведен в листинге 3.9. В программе `CreateMetaFile` имитируется типичная ситуация для графических редакторов, работающих с векторной графикой, когда рисование в клиентской области окна дублируется рисованием в метафайле, который затем сохраняется на диске.

Листинг 3.9. Проект CreateMetaFile

```
////////////////////////////////////////////////////////////////////////
// CreateMetaFile.cpp

#include <windows.h>
#include "KWnd.h"

#define FILE_NAME "Pict1.emf"

void DrawSomething(HDC);
RESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====================================================
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("CreateMetaFile", hInstance, nCmdShow, WndProc);
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

//=====================================================
RESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    RECT rect;
```

```
static HDC hdcEMF;
HENHMETAFILE hemf;

switch (uMsg)
{
case WM_CREATE:
    hdcEMF = CreateEnhMetaFile(NULL, FILE_NAME, NULL,
        "CreateMetaFile\0Pict1\0");
    break;

case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);

    DrawSomething(hDC);
    DrawSomething(hdcEMF);

    EndPaint(hWnd, &ps);
    break;

case WM_DESTROY:
    hemf = CloseEnhMetaFile(hdcEMF);
    DeleteEnhMetaFile(hemf);
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}

void DrawSomething(HDC hdc) {

    RECT r;
    HPEN hPen, hOldPen;
    HBRUSH hBrush, hOldBrush;
    SetRect(&r, 20, 20, 220, 220);

    hPen = CreatePen(PS_SOLID, 10, RGB(255, 160, 140));
    hOldPen = (HPEN)SelectObject(hdc, hPen);
    hBrush = CreateSolidBrush(RGB(140, 160, 255));
    hOldBrush = (HBRUSH)SelectObject(hdc, hBrush);
    Rectangle(hdc, r.left, r.top, r.right, r.bottom);

    hPen = CreatePen(PS_SOLID, 3, RGB(0, 255, 0));
    SelectObject(hdc, hPen);
    hBrush = CreateSolidBrush(RGB(100, 100, 100));
    SelectObject(hdc, hBrush);
    InflateRect(&r, -40, -40);
    Ellipse(hdc, r.left, r.top, r.right, r.bottom);
    SetTextColor(hdc, RGB(255, 255, 255));
    SetBkMode(hdc, TRANSPARENT);
    DrawText(hdc, "YES !", -1, &r, DT_CENTER | DT_SINGLELINE | DT_VCENTER);

    DeleteObject(SelectObject(hdc, hOldPen));
    DeleteObject(SelectObject(hdc, hOldBrush));
}


```

Метафайловый контекст устройства создается в блоке обработки сообщения **WM_CREATE**. Его дескриптор **hdcEMF** объявлен со спецификатором **static**, так как использование объекта метафайла происходит позже, при обработке других сообщений.

В приложении определена функция **DrawSomething**, которая рисует некоторое изображение в контексте устройства **hdc**, передаваемом ей в качестве параметра. Изображение может быть произвольным, но в данном случае отображается квадрат размером 200 × 200 пикселов, внутри квадрата рисуется круг, а уже внутри круга выводится текст.

При обработке сообщения **WM_PAINT** функция **DrawSomething** вызывается дважды: сначала с контекстом дисплея, затем с метафайловым контекстом.

При обработке сообщения **WM_DESTROY** вызываются следующие функции:

```
hemf = CloseEnhMetaFile(hdcEMF);
DeleteEnhMetaFile(hemf);
```

Именно благодаря им созданный метафайл сохраняется на диске.

Откомпилируйте и запустите приложение. В окне программы должно появиться изображение, показанное на рис. 3.11. Если ваш дисплей работает в режиме **True Color**, то рамка, ограничивающая квадрат, будет нежно-кремового цвета, внутренняя область квадрата — светлого серо-голубого цвета, окружность, ограничивающая круг, — светло-зеленого цвета, а сам круг — темно-серого цвета.

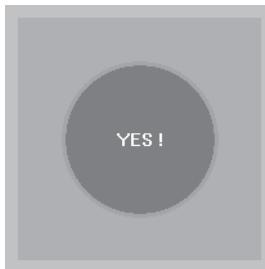


Рис. 3.11. Вывод программы CreateMetaFile

Завершите работу с программой, щелкнув мышью на кнопке закрытия окна. Теперь посмотрите на содержимое папки с вашим проектом. В ней должен появиться файл **Pict1.emf**. Это и есть созданный программой метафайл. Он может быть просмотрен с помощью любого графического редактора, поддерживающего формат **EMF**, например, при помощи приложения **ACDSee**.

Воспроизведение метафайла

Расширенный метафайл воспроизводится функцией **PlayEnhMetaFile**, которая имеет следующий прототип:

```
BOOL PlayEnhMetaFile(HDC hdc, HENHMETAFILE hemf, CONST RECT* lprRect);
```

Первый параметр, **hdc**, задает дескриптор приемного контекста устройства. Второй параметр содержит дескриптор расширенного метафайла. Третий параметр определяет ссылку на ограничивающий прямоугольник изображения в логических координатах контекста **hdc**.

Если метафайл находится на диске, то сначала нужно получить его дескриптор. Это можно сделать с помощью функции `GetEnhMetaFileHeader`:

```
HENHMETAFILE GetEnhMetaFile(LPCTSTR lpszMetaFile);
```

Функция возвращает искомый дескриптор по заданному имени файла `lpszMetaFile`. Файл должен находиться в текущем каталоге, в противном случае строка `lpszMetaFile` должна содержать полный путь к файлу.

Остается последний вопрос: как получить размеры ограничивающего прямоугольника для передачи третьему параметру функции `PlayEnhMetaFile`? Здесь нас выручит функция `GetEnhMetaFileHeader`, читающая заголовок метафайла в структуру типа `ENHMETAHEADER`, так как поле `rclBounds` этой структуры как раз и содержит необходимую информацию.

В листинге 3.10 приведен код приложения, которое воспроизводит созданный ранее метафайл `Pict1.emf`. Перед запуском приложения нужно скопировать этот файл в каталог проекта `PlayMetaFile`.

Листинг 3.10. Проект PlayMetaFile

```
////////////////////////////////////////////////////////////////////////
//    PlayMetaFile.cpp

#include <windows.h>
#include "KWnd.h"

#define FILE_NAME "Pict1.emf"

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("PlayMetaFile", hInstance, nCmdShow, WndProc);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    RECT rect;
    HENHMETAFILE hemf;
    ENHMETAHEADER emh;
    int x1, y1, x2, y2;

    switch (uMsg)
    {
    case WM_PAINT:
        hDC = BeginPaint(hWnd, &ps);
        hemf = GetEnhMetaFile(FILE_NAME);
```

продолжение ↗

Листинг 3.10 (продолжение)

```
if (!hemf)
    MessageBox(hWnd, "Файл \" FILE_NAME \" не найден.", "Error", MB_OK);
else {
    GetEnhMetaFileHeader(hemf, sizeof(ENHMETAHEADER), &emh);
    x1 = emh.rc1Bounds.left; y1 = emh.rc1Bounds.top;
    x2 = emh.rc1Bounds.right; y2 = emh.rc1Bounds.bottom;
    SetRect(&rect, x1, y1, x2, y2);
    PlayEnhMetaFile(hDC, hemf, &rect);
    DeleteEnhMetaFile(hemf);
}
EndPaint(hWnd, &ps);
break;

case WM_DESTROY:
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
}
////////////////////////////////////////////////////////////////
```

Рекомендуем вам поэкспериментировать с масштабированием воспроизводимого метафайла, замения инструкцию

```
SetRect(&rect, x1, y1, x2, y2);
```

одним из следующих вариантов:

```
SetRect(&rect, x1, y1, 2 * x2, y2);
SetRect(&rect, x1, y1, x2, 2 * y2);
SetRect(&rect, x1, y1, 2 * x2, 2 * y2);
SetRect(&rect, x1, y1, 0.5 * x2, 0.5 * y2);
```

и наблюдая, как будет изменяться выводимое изображение.

4 Средства ввода

Приложение, работающее под управлением системы Windows, получает данные от пользователя через устройства ввода, главными из которых являются клавиатура и мышь. Когда сообщения от клавиатуры или мыши воспринимаются стандартными элементами управления, такими как кнопки, поля редактирования и меню, то обработка этих событий реализована в соответствующих оконных классах. Однако иногда необходимо непосредственно обрабатывать сообщения от клавиатуры или мыши. Например, это нужно в приложениях, воспринимающих пользовательский ввод или в главном окне, или в одном из дочерних окон. В Win32 API предусмотрен широкий набор функций для управления клавиатурой и мышью.

Клавиатура

Windows обеспечивает поддержку клавиатуры, независимую от аппаратуры и от национального языка, используемого в системе. Независимость от аппаратуры достигается благодаря использованию программного драйвера клавиатуры. Независимость от национального языка реализуется выбором нужной кодовой страницы¹.

На рис. 4.1 приведена схема обработки сообщения от клавиатуры, используемая в операционной системе.

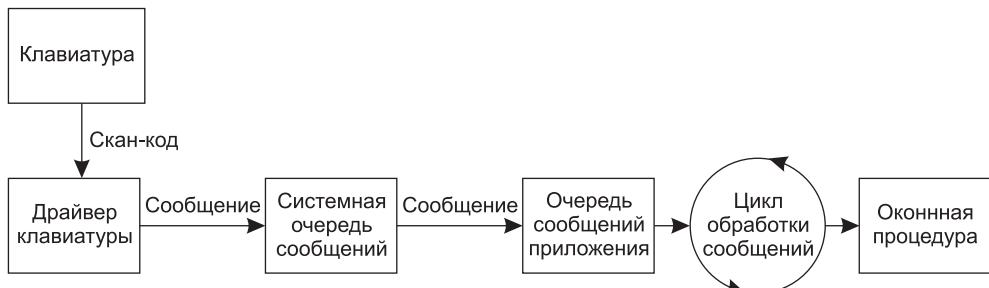


Рис. 4.1. Модель клавиатурного ввода

¹ Вы можете сменить кодовую страницу, запустив системное приложение «Клавиатура» в папке «Панель управления» и выбрав закладку «Языки и раскладки».

Каждой клавише на клавиатуре соответствуют два уникальных целых числа, называемых *скан-кодом нажатия* и *скан-кодом отпускания* клавиши. Скан-код (*scan code*) является аппаратно-зависимым, то есть на клавиатурах разных производителей одна и та же клавиша может иметь разные скан-коды¹. Когда пользователь нажимает или отпускает клавишу, драйвер клавиатуры получает соответствующие скан-коды. Полученный скан-код преобразуется драйвером в *виртуальный код клавиши* (*virtual-key code*). Этот код является уже аппаратно-независимым, поэтому в системе Windows он однозначно идентифицирует нажатую или отпущенную клавишу. Завершающим действием драйвера является создание сообщения, содержащего скан-код, виртуальный код клавиши и другую информацию о нажатой/отпущененной клавише. Это сообщение помещается в системную очередь сообщений.

Windows извлекает сообщение из системной очереди и передает его в очередь сообщений того приложения, которое имеет *активное* окно.

Далее сообщение извлекается в известном вам цикле обработки сообщений, поступая в конечном счете на обработку в соответствующую оконную процедуру.

Фокус ввода

Все приложения, работающие одновременно под управлением Windows, должны иметь возможность получать данные от клавиатуры. Если какое-то приложение имеет более одного окна, то клавиатура должна разделяться и между окнами в рамках одного приложения. Это разделение Windows реализует, используя понятие фокуса ввода. *Фокус ввода* (*keyboard focus*) — это временное свойство окна, которое означает, что все сообщения от клавиатуры направляются именно этому окну. В каждый момент времени из всех окон, имеющихся на экране, только одно окно может иметь фокус ввода.

Понятие фокуса ввода тесно связано с понятием активного окна. *Активное окно* — это окно верхнего уровня, с которым в данный момент работает пользователь. Система выделяет текст заголовка активного окна, поэтому отыскать его на экране довольно просто. Если активное окно минимизировано, то Windows выделяет текст заголовка на панели задач.

Фокусом ввода всегда владеет либо активное окно, либо одно из его дочерних окон. Часто дочерними окнами являются элементы управления — кнопки, переключатели, флажки, текстовые поля и списки, которые обычно размещаются в окне диалога. Элементы управления по-разному показывают, что они находятся в фокусе. Так, вокруг текста кнопки выводится точечная линия, а текстовое поле показывает, что оно получило фокус ввода, при помощи мигающего курсора.

Если активное окно минимизировано, то окна с фокусом ввода нет. Windows продолжает пересыпать программе сообщения клавиатуры, но эти сообщения выглядят иначе, чем сообщения, направляемые активным неминимизированным окнам.

Обрабатывая сообщения WM_SETFOCUS и WM_KILLFOCUS, оконная процедура может определить текущий статус связанного с ней окна. Первое сообщение показывает, что окно получило фокус ввода, второе — что окно потеряло фокус ввода.

¹ По этой причине используется также термин «скан-код OEM» (Original Equipment Manufacturer Scan Code).

Для работы с фокусом ввода предусмотрены следующие функции:

- Функция `HWND SetFocus(HWND hWnd)` устанавливает фокус ввода на окно `hWnd`, возвращая дескриптор окна, которое располагало фокусом до вызова функции.
- Функция `HWND GetFocus()` возвращает дескриптор окна, имеющего фокус ввода в текущий момент.

Клавиши и символы

Клавиатура всегда генерирует некоторые числовые коды, но ее можно рассматривать либо как совокупность отдельных физических клавиш, либо как средство генерации кодов символов.

В первом случае любой генерируемый код должен идентифицировать клавишу и показывать, нажата она или отпущена. Во втором варианте код, вырабатываемый при нажатии клавиши, идентифицирует уникальный символ из набора символов. По умолчанию используется набор символов ANSI¹.

У многих клавиш современного компьютера нет кодов символов. Ни функциональные клавиши, ни клавиши управления курсором их не генерируют. Поэтому в программах, использующих ввод с клавиатуры нетривиальным способом, обычно приходится иметь дело с клавиатурой и как с совокупностью клавиш, и как с генератором символов одновременно.

Клавиши можно разделить на следующие четыре группы:

- *Клавиши-переключатели* — Caps Lock, Num Lock, Scroll Lock и в некоторых случаях клавиша Insert. При нажатии такой клавиши включается ее состояние, а при повторном нажатии — выключается. Как правило, на клавиатуре есть световые индикаторы состояния клавиш-переключателей.
- *Клавиши управления регистром* — Shift, Ctrl и Alt. В нажатом состоянии такая клавиша меняет интерпретацию других клавиш. Иногда такие клавиши называют *клавишами-модификаторами*.
- *Клавиши, не генерирующие символов*, — функциональные клавиши, клавиши управления курсором, Pause, Delete.
- *Символьные клавиши* — буквы, цифры и другие символы, пробел, Tab, Backspace, Esc и Enter. Впрочем, клавиши Tab, Backspace, Esc и Enter также можно рассматривать как клавиши без символов.

Сообщения клавиатуры, которые приложение получает от Windows, можно разделить на «аппаратные» (*keystrokes*) и «символьные» (*characters*). Это разделение соответствует указанной выше двойственной интерпретации клавиатуры.

Аппаратные сообщения

Аппаратные сообщения от клавиатуры приведены в табл. 4.1. Обычно различают *системные* и *несистемные* сообщения. Для системных сообщений используется префикс SYS в составе идентификатора кода сообщения.

¹ См. главу 2, раздел «Рисование текста».

Таблица 4.1. Аппаратные сообщения от клавиатуры

Категория	Клавиша нажата	Клавиша отпущена
Несистемные сообщения	WM_KEYDOWN	WM_KEYUP
Системные сообщения	WM_SYSKEYDOWN	WM_SYSKEYUP

Системные сообщения WM_SYSKEYDOWN и WM_SYSKEYUP, вырабатываемые обычно при нажатии клавиш в сочетании с клавишей Alt, более важны для Windows, чем для приложения. Эти сообщения вызывают команды программного или системного меню. Они также используются для вызова системных функций, таких как смена активного окна (Alt + Tab), или как быстрые клавиши системного меню (Alt в сочетании с функциональной клавишей). Поскольку система Windows самостоятельно отрабатывает всю логику Alt-клавиш, то вам, фактически, не нужно обрабатывать эти сообщения. Ваша оконная процедура в конце концов получит другие сообщения, являющиеся следствием этих аппаратных сообщений клавиатуры (например, выбор команды меню).

Обычно сообщения о нажатии и отпускании клавиши появляются парами. Однако если пользователь оставит клавишу нажатой так долго, что включится автоповтор, то Windows пошлет оконной процедуре серию сообщений WM_KEYDOWN и одно сообщение WM_KEYUP, когда, в конце концов, клавиша будет отпущена.

Для всех аппаратных сообщений клавиатуры 32-разрядная переменная lParam, передаваемая в оконную процедуру, содержит 6 полей (табл. 4.2).

Таблица 4.2. Интерпретация полей параметра lParam

Разряды	Назначение	Описание
15 . . . 0	Счетчик повторений	Число повторений данного сообщения. Равно либо единице, либо числу нажатий в режиме автоповтора, когда пользователь удерживает клавишу нажатой. Для сообщения WM_KEYUP всегда равен единице
23 . . . 16	Скан-код	Восьмибитный скан-код OEM
24	Флаг расширенной клавиатуры	Устанавливается в 1 для дополнительных клавиш расширенной клавиатуры IBM (функциональные клавиши в верхней части клавиатуры, клавиши Alt и Ctrl на правой стороне клавиатуры и некоторые другие клавиши)
29	Код контекста	Устанавливается в 1, если нажата клавиша Alt (для сообщений WM_KEYDOWN и WM_KEYUP всегда равен 0)
30	Флаг предыдущего состояния клавиши	Единица, если клавиша была нажата, и нуль, если клавиша была отпущена
31	Флаг нового состояния клавиши	Нуль, если клавиша нажата, и единица, если клавиша отпущена

Хотя некоторая информация в lParam при обработке аппаратных сообщений может оказаться полезной, гораздо более важен параметр wParam, содержащий виртуальный код клавиши.

Виртуальный код клавиши (*virtual key code*) идентифицирует нажатую или отпущенную клавишу и является аппаратно-независимым. Идентификаторы виртуальных клавиш определены в файле winuser.h. В табл. 4.3 приведены наиболее часто используемые коды.

Таблица 4.3. Виртуальные коды клавиш

Десятичное значение	Шестнадцатеричное значение	Идентификатор	Клавиатура IBM
3	03	VK_CANCEL	Ctrl + Break
8	08	VK_BACK	Backspace
9	09	VK_TAB	Tab
13	0D	VK_RETURN	Enter
16	10	VK_SHIFT	Shift
17	11	VK_CONTROL	Ctrl
18	12	VK_MENU	Alt
19	13	VK_PAUSE	Pause
20	14	VK_CAPITAL	Caps Lock
27	1B	VK_ESCAPE	Esc
32	20	VK_SPACE	Пробел
33	21	VK_PRIOR	Page Up
34	22	VK_NEXT	Page Down
35	23	VK_END	End
36	24	VK_HOME	Home
37	25	VK_LEFT	Стрелка влево
38	26	VK_UP	Стрелка вверх
39	27	VK_RIGHT	Стрелка вправо
40	28	VK_DOWN	Стрелка вниз
44	2C	VK_SNAPSHOT	Print Screen
45	2D	VK_INSERT	Insert
46	2E	VK_DELETE	Delete
48...57	30...39		0...9 (на основной клавиатуре)
65...90	41...5A		A...Z
112...123	70...7B	VK_F1...VK_F12	F1...F12
144	90	VK_NUMLOCK	Num Lock
145	91	VK_SCROLL	Scroll Lock

Обычно оконная процедура обрабатывает только небольшую часть из всех аппаратных сообщений клавиатуры, игнорируя остальные. Например, она может обрабатывать только те сообщения WM_KEYDOWN, которые содержат виртуальные коды для клавиш управления курсором, клавиши Shift и функциональных клавиш. В то же время аппаратные сообщения для символьных клавиш могут игнорироваться. Дело в том, что ввод информации от символьных клавиш удобнее обрабатывать, используя символьное сообщение WM_CHAR.

Символьные сообщения

Когда нажимается символьная клавиша, генерируемый код зависит от многих факторов. На его формирование влияет положение клавиш-переключателей и клавиш управления регистром, а также текущая кодовая страница, установлен-

ная в системе. Все эти факторы учитывает функция `TranslateMessage`, вызываемая в цикле обработки сообщений, который находится в теле функции `WinMain` и имеет вид

```
while (GetMessage(&msg, NULL, 0, 0)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

Напомним, что функция `GetMessage` извлекает очередное сообщение из очереди и помещает его в структурную переменную `msg`, а функция `DispatchMessage` вызывает соответствующую оконную процедуру.

Между этими двумя функциями располагается вызов функции `TranslateMessage`, которая преобразует аппаратные сообщения клавиатуры в символьные сообщения. Если аппаратным сообщением является `WM_KEYDOWN`, а нажатая клавиша в сочетании с положением клавиш-модификаторов соответствует некоторому символу, то `TranslateMessage` помещает символьное сообщение в очередь сообщений.

В результате системной обработки сообщения `WM_KEYDOWN` может быть инициировано символьное сообщение `WM_CHAR` или `WM_DEADCHAR`.

Стоит привести краткие пояснения о назначении сообщения `WM_DEADCHAR`. На некоторых неамериканских клавиатурах имеются так называемые *немые клавиши* (*dead keys*), предназначенные для ввода диакритических знаков, таких как «умляут», например. Диакритические знаки вводятся в паре с обычными символами, в результате чего формируется составной символ. Например, для ввода на немецкой клавиатуре символа «U» (U-умляут) пользователь сначала нажимает немую клавишу «умляут», а затем — клавишу U. К счастью, вам не нужно обрабатывать в вашей программе сообщение `WM_DEADCHAR`, поскольку Windows берет это на себя, вырабатывая в результате сообщение `WM_CHAR`, которое содержит ANSI-код буквы с диакритическим знаком.

Таким образом, обработка символьных сообщений обычно сводится к обработке сообщения `WM_CHAR`. Параметр `lParam` сообщения `WM_CHAR`, передаваемый в оконную процедуру, имеет то же значение, что и параметр `lParam` породившего его аппаратного сообщения. Параметр `wParam` содержит код символа ANSI.

Символьные сообщения передаются в оконную процедуру в промежутке между аппаратными сообщениями клавиатуры. Например, если переключатель `Caps Lock` выключен¹, а пользователь нажимает и отпускает клавишу A, то оконная процедура получит следующие три сообщения:

Сообщение	Клавиша или код
<code>WM_KEYDOWN</code>	Виртуальная клавиша A
<code>WM_CHAR</code>	ANSI-код символа 'a'
<code>WM_KEYUP</code>	Виртуальная клавиша A

Если пользователь, удерживая клавишу `Shift`, нажимает клавишу A, отпускает клавишу A и затем отпускает клавишу `Shift`, то оконная процедура получит пять сообщений:

¹ Состояние клавиши-переключателя `Caps Lock` контролируется при помощи светового индикатора <`Caps Lock`>.

Сообщение	Клавиша или код
WM_KEYDOWN	Виртуальная клавиша VK_SHIFT
WM_KEYDOWN	Виртуальная клавиша A
WM_CHAR	ANSI-код символа A
WM_KEYUP	Виртуальная клавиша A
WM_KEYUP	Виртуальная клавиша VK_SHIFT

Работа с кареткой

Каретка — это небольшой символ в виде вертикальной или горизонтальной черты. Иногда каретка отображается в виде прямоугольника, который показывает пользователю место на экране, где будет отображен очередной символ, вводимый с клавиатуры. Если вы создаете программу, реализующую функции упрощенного текстового редактора, то необходимо позаботиться и об управлении кареткой.

Для этого в Win32 API предусмотрены функции, приведенные в следующем списке:

- Функция `CreateCaret` создает связанную с окном каретку.
- Функция `SetCaretPos` устанавливает положение каретки в окне.
- Функция `ShowCaret` показывает каретку.
- Функция `HideCaret` прячет каретку.
- Функция `DestroyCaret` удаляет каретку.

Однако вы не можете просто создать каретку при обработке сообщения `WM_CREATE` и удалить ее при обработке сообщения `WM_DESTROY`. Каретка является общесистемным ресурсом и должна разделяться между всеми работающими приложениями. Следует помнить, что появление каретки в окне имеет смысл только в том случае, когда окно имеет фокус ввода. Оконная процедура может контролировать моменты, когда окно получает или теряет фокус ввода, обрабатывая сообщения `WM_SETFOCUS` и `WM_KILLFOCUS`.

Это диктует основные правила использования каретки. Оконная процедура вызывает функцию `CreateCaret` при обработке сообщения `WM_SETFOCUS` и функцию `DestroyCaret` при обработке сообщения `WM_KILLFOCUS`.

Рассмотрим подробней использование функции `CreateCaret`, имеющей следующий прототип:

```
BOOL CreateCaret(HWND hWnd, HBITMAP hBitmap, int nWidth, int nHeight);
```

Первый параметр функции содержит дескриптор окна, которое становится владельцем создаваемой каретки.

Второй параметр определяет форму каретки. В нем можно указать дескриптор раstra, единичное значение или значение `NULL`.

Если параметр `hBitmap` равен `NULL`, то каретка имеет форму прямоугольного блока шириной `nWidth` и высотой `nHeight`, закрашенного черной однородной кистью.

Если параметр `hBitmap` равен единице, то форма каретки будет такой же, что и для значения `NULL`, но заполняться блок будет черно-белой кистью с шахматным рисунком, что выглядит внешне как серый цвет. При единичной ширине каретки она будет выглядеть как пунктирная линия.

Если параметру `hBitmap` передается дескриптор растра, то форма каретки определяется этим растром. В этом случае параметры `nWidth` и `nHeight` игнорируются.

Параметры `nWidth` и `nHeight` задают ширину и высоту каретки в логических единицах. Их использование уже было описано выше. Если параметр `nWidth` имеет нулевое значение, то используется предопределенная в системе ширина рамки для окна, которая обычно равна одному пикселу. Если `nHeight` имеет нулевое значение, то используется предопределенная в системе высота рамки для окна, которая тоже обычно равна одному пикселу¹.

В случае успешного завершения функция `CreateCaret` возвращает ненулевое значение. Нулевое значение служит признаком ошибки.

После создания каретки функцией `CreateCaret` она все еще остается скрытой. Чтобы каретка стала видимой, необходимо вызвать функцию `ShowCaret`. Перед уничтожением каретки с помощью функции `DestroyCaret` рекомендуется убрать ее с экрана функцией `HideCaret`.

Примитивный текстовый редактор

Программа `TypeR`, код которой приведен в листинге 4.1, демонстрирует принципы и технику обработки аппаратных и символьных сообщений клавиатуры. Программу можно рассматривать как примитивный текстовый редактор. Пользователь может набирать в окне текст, переходить на следующую строку, нажимая клавишу `Enter`, перемещать каретку по уже набранному тексту при помощи клавиш управления курсором, вводить новый текст в том месте, на которое указывает каретка, или удалять предшествующий символ. Однако у этого редактора нет ни полос прокрутки, ни функций поиска и замены строк, ни возможности сохранять текст в файле.

Для хранения текста, введенного с клавиатуры, в программе используется объект класса `string` из стандартной библиотеки C++ вместо традиционной С-строки. Благодаря автоматическому управлению выделением и освобождением памяти в этом классе мы можем писать более компактный программный код, не отвлекаясь на детали реализации.

Чтобы использовать в программе объекты класса `string`, необходимо в начале файла добавить следующие директивы:

```
#include <string>
using namespace std;
```

Переменная (объект класса `string`) для хранения введенных символов объявляется следующим оператором:

```
static string text;
```

Спецификатор `static` необходим для сохранения текущего значения переменной `text` до следующего вызова оконной процедуры².

Хотя предполагается, что читатель знаком с классом `string`, мы все же напомним назначение тех операций и методов класса, которые используются в коде.

¹ Предопределенные в системе ширину и высоту рамки для окна можно получить вызовами функций `GetSystemMetrics(SM_CXBORDER)` и `GetSystemMetrics(SM_CYBORDER)` соответственно.

² Локальные переменные, не имеющие спецификатора `static`, инициализируются заново при каждом вызове функции `WndProc`.

Добавление символа `symb` в конец строки осуществляется с помощью операции `+=`:

```
text += symb;
```

Метод `insert` позволяет вставлять символ `symb` в произвольную позицию `index`:

```
text.insert(index, symb);
```

Для удаления `n` символов, начиная с позиции `index`, вызывается метод `erase`:

```
text.erase(index, n);
```

В любой момент можно узнать текущий размер строки при помощи метода `size`, например, в выражениях, подобных следующему:

```
nLines = text.size() / nCharPerLine;
```

И наконец, содержимое строки `text` можно преобразовать к типу обычной С-строки при помощи метода `c_str`. А это пригодится при вызове функции `TextOut`, поскольку она принимает в качестве четвертого параметра адрес С-строки.

Обратите внимание на обработку сообщений `WM_KEYDOWN`, `WM_CHAR`, `WM_SETFOCUS`, `WM_KILLFOCUS`, а также на использование редактором `Typer` системного *фиксированного* шрифта для вывода текста. Создание текстового редактора для пропорционального шрифта было бы более трудной задачей.

Листинг 4.1. Проект `Typer`¹

```
////////////////////////////////////////////////////////////////////////
// Typer.cpp
#include <windows.h>
#include <string>
using namespace std;

#include <KWnd.h>

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd(<Typer>, hInstance, nCmdShow, WndProc);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
```

продолжение ↗

¹ Не забудьте добавить к проекту файлы `KWnd.h` и `KWnd.cpp`, код которых приведен в листинге 1.2 (глава 1).

Листинг 4.1 (продолжение)

```
TEXTMETRIC tm;

static string text; // буфер для хранения введенного текста
string symb;

static int cxChar, cyChar, cxClient, cyClient;
static int nCharPerLine, nClientLines;
static int xCaret = 0, yCaret = 0;
int curIndex;
int nLines; // число «полных» строк текста
int nTailChar; // число символов в последней строке
int x, y, i;

switch (uMsg) {

case WM_CREATE:
    hDC = GetDC(hWnd);
    SelectObject(hDC, GetStockObject(SYSTEM_FIXED_FONT));
    GetTextMetrics(hDC, &tm);
    cxChar = tm.tmAveCharWidth;
    cyChar = tm.tmHeight;
    ReleaseDC(hWnd, hDC);
    break;

case WM_SIZE:
    // получить размеры окна в пикселях
    cxClient = LOWORD(lParam);
    cyClient = HIWORD(lParam);
    // вычислить размеры окна в символах (по 'x') и строках (по 'y')
    nCharPerLine = max(1, cxClient / cxChar);
    nClientLines = max(1, cyClient / cyChar);

    if (hWnd == GetFocus())
        SetCaretPos(xCaret * cxChar, yCaret * cyChar);
    break;

case WM_SETFOCUS:
    // создать и показать каретку
    CreateCaret(hWnd, NULL, 0, cyChar);
    SetCaretPos(xCaret * cxChar, yCaret * cyChar);
    ShowCaret(hWnd);
    break;

case WM_KILLFOCUS:
    // спрятать и уничтожить каретку
    HideCaret(hWnd);
    DestroyCaret();
    break;

case WM_KEYDOWN:
    nLines = text.size() / nCharPerLine;
    nTailChar = text.size() % nCharPerLine;

    switch (wParam) {

case VK_HOME:    xCaret = 0;
    break;
```

```
case VK_END:    if (yCaret == nLines)
                  xCaret = nTailChar;
                else
                  xCaret = nCharPerLine - 1;
                break;
  case VK_PRIOR: yCaret = 0;
                break;
  case VK_NEXT:   yCaret = nLines;
                  xCaret = nTailChar;
                break;
  case VK_LEFT:   xCaret = max(xCaret - 1, 0);
                break;
  case VK_RIGHT:  xCaret = min(xCaret + 1, nCharPerLine - 1);
                  if ((yCaret == nLines) && (xCaret > nTailChar))
                    xCaret = nTailChar;
                break;
  case VK_UP:     yCaret = max(yCaret - 1, 0);
                break;
  case VK_DOWN:   yCaret = min(yCaret + 1, nLines);
                  if ((yCaret == nLines) && (xCaret > nTailChar))
                    xCaret = nTailChar;
                break;
  case VK_DELETE: curIndex = yCaret * nCharPerLine + xCaret;
                  if (curIndex < text.size()) {
                    text.erase(curIndex, 1);
                    InvalidateRect(hWnd, NULL, TRUE);
                  }
                break;
}
}

SetCaretPos(xCaret * cxChar, yCaret * cyChar);
break;

case WM_CHAR:
switch (wParam) {
  case '\b': // символ 'backspace'
    if (xCaret > 0) {
      xCaret--;
      SendMessage(hWnd, WM_KEYDOWN, VK_DELETE, 1);
    }
    break;

  case '\t': // символ 'tab'
    do { SendMessage(hWnd, WM_CHAR, ' ', 1L); }
    while (xCaret % 8 != 0);
    break;

  case '\r': case '\n': // возврат каретки или перевод строки
    for (i = 0; i < nCharPerLine - xCaret; ++i)
      text += ' ';
    xCaret = 0;
    if (++yCaret == nClientLines)
      MessageBox(hWnd, "Нет места в окне", "Error", MB_OK);
    yCaret--;
}
}
```

Листинг 4.1 (продолжение)

```

        break;

    default: // любой другой символ
        curIndex = yCaret * nCharPerLine + xCaret;
        if (curIndex == text.size())
            text += (char)wParam;
        else {
            symb = (char)wParam;
            text.insert(curIndex, symb);
        }
    }

    InvalidateRect(hWnd, NULL, TRUE);

    if (++xCaret == nCharPerLine) {
        xCaret = 0;
        if (++yCaret == nClientLines) {
            MessageBox(hWnd, "Нет места в окне", "Error", MB_OK);
            yCaret--;
        }
    }
    break;
}

SetCaretPos(xCaret * cxChar, yCaret * cyChar);
break;

case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);
    SelectObject(hDC, GetStockObject(SYSTEM_FIXED_FONT));

    if (text.size()) {
        nLines = text.size() / nCharPerLine;
        nTailChar = text.size() % nCharPerLine;
        for (y = 0; y < nLines; ++y)
            TextOut(hDC, 0, y*cyChar, text.c_str() + y*nCharPerLine,
                    nCharPerLine);
        TextOut(hDC, 0, y*cyChar, text.c_str() + y*nCharPerLine,
                nTailChar);
    }
    EndPaint(hWnd, &ps);
    break;

case WM_DESTROY:
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
/////////////////////////////////////////////////////////////////

```

Откомпилируйте проект и посмотрите, как будет работать текстовый редактор *TypeR*. Попытайтесь изменить параметры каретки. Для этого найдите в блоке обработки сообщения **WM_SETFOCUS** следующую инструкцию:

```
CreateCaret(hWnd, NULL, 0, cyChar);
```

Поменяйте ее сначала на инструкцию

```
CreateCaret(hWnd, 1, 0, cyChar);
```

а затем на инструкцию

```
CreateCaret(hWnd, 1, 10, cyChar);
```

наблюдая каждый раз после компиляции, как будет меняться форма каретки.

Мышь

Мышь — это устройство ввода информации с одной или более кнопками. Win32 API поддерживает однокнопочную, двухкнопочную или трехкнопочную мышь, а также мышь с колесиком. Наличие мыши можно определить при помощи функции `GetSystemMetrics`:

```
fMouse = GetSystemMetrics (SM_MOUSEPRESENT);
```

Если мышь установлена, то значение `fMouse` будет равно `TRUE`. Количество кнопок у мыши определяется следующим вызовом:

```
cButtons = GetSystemMetrics (SM_CMOUSEBUTTONS);
```

Если мышь не установлена, то функция вернет нулевое значение.

Для двухкнопочной мыши, которая наиболее популярна среди пользователей, сложилась некоторая традиция интерпретации ее кнопок. Левая кнопка считается главной и выполняет самые распространенные действия, к которым относятся выбор предметов, перетаскивание значков и инициирование действий. Правая кнопка чаще всего используется для вызова контекстного меню. Оно появляется там, где находится курсор мыши, и набор его функций соответствует возможностям, доступным в этой области.

Трехкнопочная мышь не отличалась особой популярностью у пользователей Windows, пока Microsoft не выпустила модель IntelliMouse. Хотя эту мышь трудно считать интеллектуальной в прямом смысле этого слова, ее новый конструктивный элемент в виде небольшого колесика, расположенного между левой и правой кнопками, позволяет использовать множество дополнительных функций. При нажатии на колесико оно функционирует как средняя кнопка. Но кроме этого колесико можно прокручивать, и программы, поддерживающие эту функцию, будут реагировать прокруткой или масштабированием документа. Это намного удобнее, чем работать мышью на полосе прокрутки, так как колесико выполняет свои функции даже тогда, когда курсор мыши находится за пределами окна с документом.

Наличие мыши с колесиком можно определить при помощи вызова все той же функции `GetSystemMetrics`:

```
fMouseWheel = GetSystemMetrics(SM_MOUSEWHEELPRESENT);
```

Функция возвращает значение `TRUE`, если у мыши есть колесо.

Хотя мышь является важнейшим средством ввода для пользователя, разработчики программ для Windows часто дублируют, если это возможно, «мышиный» интерфейс соответствующим клавиатурным интерфейсом. При разработке новых программ весьма желательно придерживаться этой концепции, ориентированной на обеспечение максимума удобств для конечного пользователя.

Терминология, связанная с мышью

Когда пользователь перемещает мышь, Windows передвигает по экрану небольшую растровую картинку, которая выступает в качестве *курсора мыши* (*mouse cursor*).

У курсора мыши есть *горячая точка* (*hot spot*) — пиксел, соответствующий положению курсора на экране. Когда говорят о позиции курсора мыши, то имеют в виду именно позицию горячей точки. Например, горячая точка стандартного курсора в виде стрелки (*IDC_ARROW*) находится на кончике стрелки. Другой курсор, в виде перекрестья (*IDC_CROSS*), имеет горячую точку в центре крестообразного шаблона.

Следует помнить, что курсор, используемый по умолчанию, для конкретного окна задается при определении структуры класса окна. Например, следующим оператором:

```
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
```

Действия пользователя с мышью принято описывать следующими терминами:

- *Щелчок* — нажатие и отпускание кнопки мыши.
- *Двойной щелчок* — быстрое двойное нажатие и отпускание кнопки мыши.
- *Перетаскивание* — перемещение мыши при нажатой кнопке.

Кнопки трехкнопочной мыши называются *левой*, *средней* и *правой* кнопками. В связанных с мышью идентификаторах, которые определены в заголовочных файлах Windows, используются аббревиатуры *LBUTTON*, *MBUTTON* и *RBUTTON*. На двухкнопочной мыши имеются только левая и правая кнопки. Единственная кнопка однокнопочной мыши является левой.

Сообщения мыши

Изучая в предыдущем разделе ввод с клавиатуры, мы видели, что Windows посылает сообщения клавиатуры только тому окну, которое имеет фокус ввода. С мышью Windows ведет себя несколько иначе. Оконная процедура получает сообщения мыши, и когда мышь проходит через окно, и при щелчке внутри окна, даже если окно не активно или не имеет фокуса ввода.

Если мышь перемещается по клиентской области окна, то оконная процедура получает сообщение *WM_MOUSEMOVE*. Если кнопка мыши *нажимается* или *отпускается* внутри клиентской области, то оконная процедура получает следующие сообщения:

Кнопка	Нажатие	Отпускание	Нажатие (второй щелчок)
Левая	WM_LBUTTONDOWN	WM_LBUTTONUP	WM_LBUTTONDOWNDBLCLK
Средняя	WM_MBUTTONDOWN	WM_MBUTTONUP	WM_MBUTTONDOWNDBLCLK
Правая	WM_RBUTTONDOWN	WM_RBUTTONUP	WM_RBUTTONDOWNDBLCLK

Для всех этих сообщений значение параметра *lParam* содержит положение мыши. При этом в младшем слове находится значение координаты *x*, а в старшем слове — значение координаты *y*. Отсчет координат ведется от левого верхнего угла клиентской области окна. Эти значения можно извлечь из *lParam* при помощи макроросов *LOWORD* и *HIGHWORD*.

Значение параметра wParam показывает состояние кнопок мыши и клавиш Shift и Ctrl. Вы можете проверить параметр wParam при помощи соответствующих битовых масок:

Константа	Описание
MK_LBUTTON	Левая клавиша нажата
MK_MBUTTON	Средняя клавиша нажата
MK_RBUTTON	Правая клавиша нажата
MK_SHIFT	Клавиша Shift нажата
MK_CONTROL	Клавиша Ctrl нажата

Если пользователь щелкнет левой кнопкой мыши в клиентской области неактивного окна, то Windows сделает это окно активным, а затем передаст оконной процедуре сообщение WM_LBUTTONDOWN.

Обработка двойного щелчка

Два последовательных щелчка воспринимаются системой как двойной щелчок, если они происходят в течение достаточно короткого промежутка времени. Пользователь может самостоятельно задать этот интервал при помощи системного приложения Мышь (Mouse), которое можно найти в группе Панель управления.

Следует особо подчеркнуть, что окно будет получать сообщения о двойном щелчке (DBLCLK) только в том случае, если стиль соответствующего класса окна содержит флаг CS_DBLCLKS. Поэтому перед регистрацией класса окна нужно присвоить полю style структуры WNDCLASS значение, включающее этот флаг. Впрочем, если используется объект класса KWnd, то следует модифицировать стиль класса окна вызовом следующих инструкций:

```
style = GetClassLong(hWnd, GCL_STYLE);
SetClassLong(hWnd, GCL_STYLE, style | CS_DBLCLKS);
```

Если класс окна определен без флага CS_DBLCLKS и пользователь делает двойной щелчок левой кнопкой мыши, то оконная процедура последовательно получает сообщения WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDOWN и WM_LBUTTONUP.

Если класс окна определен с флагом CS_DBLCLKS, то после двойного щелчка оконная процедура получит сообщения WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDOWN-DBLCLK и WM_LBUTTONUP. Легко заметить, что в этом случае второе сообщение WM_LBUTTONDOWN заменяется сообщением WM_LBUTTONDOWNDBLCLK.

Пример обработки сообщения WM_LBUTTONDOWNDBLCLK приведен в листинге 4.4.

Обработка сообщений от колеса мыши

При нажатии на колесо мыши Windows генерирует такие же сообщения, какие вырабатываются при нажатии средней кнопки трехкнопочной мыши. Прокрутка же колесика вызывает сообщение WM_MOUSEWHEEL.

Если вам нужно обрабатывать это сообщение в коде программы, то в начале файла добавьте следующую директиву:

```
#include <zmouse.h>
```

В заголовочном файле zmouse.h определены все необходимые константы.

Младшая часть параметра wParam сообщения WM_MOUSEWHEEL интерпретируется точно так же, как для других сообщений мыши, то есть показывает состояние кнопок мыши и клавиш Shift и Ctrl.

Старшая часть параметра wParam содержит значение, отображающее дистанцию, пройденную колесом. Оно рассчитывается как количество шагов колеса при прокрутке, умноженное на коэффициент WHEEL_DELTA. В заголовочном файле zmouse.h этот коэффициент определен со значением 120.

Обработка сообщения WM_MOUSEWHEEL довольно проста, и ее пример можно найти в листинге 4.5.

Теперь рассмотрим примеры программного кода.

Рисуем мышью

В любом графическом редакторе, как правило, предусмотрена возможность рисовать линии произвольной формы с помощью мыши. Пользователь устанавливает курсор на начальную точку линии, нажимает левую кнопку мыши и, не отпуская кнопку, перемещает мышь, рисуя прямые, кривые и другие замысловатые линии. Рисование линии заканчивается, когда левая кнопка отпускается.

Попробуем реализовать эту операцию в программе Scribble (листинг 4.2).

Листинг 4.2. Проект Scribble

```
//////////  
// Scribble.cpp  
#include <windows.h>  
#include "KWnd.h"  
  
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);  
//////////  
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,  
    LPSTR lpCmdLine, int nCmdShow)  
{  
    MSG msg;  
    KWnd mainWnd("Scribble", hInstance, nCmdShow, WndProc);  
  
    while (GetMessage(&msg, NULL, 0, 0)) {  
        TranslateMessage(&msg);  
        DispatchMessage(&msg);  
    }  
    return msg.wParam;  
}  
  
=====  
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)  
{  
    static HDC hDC;  
    static int x, y; // позиция курсора мыши  
    static BOOL bTracking = FALSE;  
  
    switch (uMsg)  
    {  
        case WM_CREATE:  
            hDC = GetDC(hWnd);
```

```
break;

case WM_LBUTTONDOWN:
    bTracking = TRUE;
    // начальная позиция
    x = LOWORD(lParam);
    y = HIWORD(lParam);
    MoveToEx(hdc, x, y, NULL);
    break;

case WM_LBUTTONUP:
    bTracking = FALSE;
    break;

case WM_MOUSEMOVE:
    if (bTracking) {
        // новая позиция
        x = LOWORD(lParam);
        y = HIWORD(lParam);
        LineTo(hdc, x, y);
    }
    break;

case WM_DESTROY:
    ReleaseDC(hWnd, hdc);
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
////////////////////////////////////////////////////////////////
```

Процесс рисования очередной линии начинается с нажатия левой кнопки мыши. Программа, обрабатывая сообщение `WM_LBUTTONDOWN`, устанавливает флаг слежения за мышью `bTracking` в значение `TRUE` и задает начальную позицию пера при помощи функции `MoveToEx`. После этого при обработке сообщения `WM_MOUSEMOVE` программа просто проводит линию в текущую позицию курсора мыши. Когда пользователь отпускает кнопку мыши, флаг `bTracking` сбрасывается в `FALSE` и рисование прекращается.

Обратите особое внимание на то, с каким контекстом устройства работают рисующие функции `MoveToEx` и `LineTo`. Дескриптор контекста дисплея `hDC` объявлен со спецификатором `static`. Это раз. Программа получает контекст устройства при помощи функции `GetDC` в блоке обработки сообщения `WM_CREATE`, а освобождает его с помощью `ReleaseDC` в блоке обработки сообщения `WM_DESTROY`. Это два. Все это позволяет обновлять клиентскую область непосредственно в ответ на пользовательский ввод информации с помощью мыши, не прибегая к механизмам, генерирующему сообщение `WM_PAINT`. Иными словами, обеспечивается минимальная задержка между вводом пользователя и реакцией приложения. Статический класс памяти для переменной `hDC` здесь необходим потому, что требуется сохранять значение атрибута «текущая позиция пера», используемого и модифицируемого функцией `LineTo`, между двумя последовательными вызовами функции `WndProc`.

На рис. 4.2 показан результат испытаний «графического редактора» Scribble.



Рис. 4.2. Рисование мышью в редакторе Scribble

Но что это?.. Как только пользователь пытается изменять размеры окна, ухватившись мышью за его рамку, изображение тотчас бесследно исчезает! Ах, да... Мы же забыли про обработку сообщения WM_PAINT, которое генерируется при изменении размеров окна!

Давайте устраним эту оплошность. Но что же будет рисовать приложение в блоке обработки сообщения WM_PAINT? Ведь информация о траекториях движения мыши нигде не сохранена! Значит, надо где-то сохранять массивы координат позиций мыши для каждой рисуемой линии. Сложность проблемы состоит в том, что количество линий заранее неизвестно, так же как и количество точек в каждой линии может быть произвольным. Таким образом, использование статических массивов будет крайне неудобным.

На выручку в таких ситуациях приходит замечательный класс `vector` из стандартной библиотеки шаблонов STL. Объект класса `vector` — это, фактически, динамический массив с переменным размером, обслуживаемый набором удобных для программиста методов. Аналогично классу `string` класс `vector` обеспечивает автоматическое управление выделением и освобождением памяти.

Чтобы использовать в программе объекты класса `vector`, необходимо в начало файла с исходным кодом добавить следующие директивы:

```
#include <vector>
using namespace std;
```

Класс `vector` является шаблонным классом, поэтому при объявлении его объекта необходимо указать в угловых скобках имя конкретного типа, как показано ниже:

```
vector<int> v1; // v1 - вектор элементов типа int
```

Мы, конечно, не будем рассматривать здесь все методы класса `vector`, а остановимся кратко только на тех, которые нужны для нашей программы.

После объявления вектора `v1`, приведенного выше, этот вектор пуст, то есть не содержит ни одного элемента. Добавление нового элемента в конец вектора осуществляется при помощи метода `push_back`:

```
v1.push_back(value);
```

Эта инструкция добавляет новый элемент типа `int` в конец вектора `v1` и присваивает ему значение переменной `value`.

Количество элементов в векторе в любой момент времени можно определить при помощи метода `size`.

Доступ к произвольному элементу вектора осуществляется либо через индекс, как в обычном массиве, либо при помощи так называемого *итератора*, что похоже на доступ через указатель в обычном массиве. В следующем фрагменте кода показан второй способ доступа к элементам вектора на примере обхода всех элементов вектора `v1`:

```
vector<int>::iterator it; // объявление итератора it для класса vector<int>
for (it = v1.begin(); it != v1.end(); ++it) {
    // *it - значение элемента с адресом it
}
```

Здесь `begin()` — метод, возвращающий адрес первого элемента вектора `v1`, а `end()` — метод, возвращающий адрес воображаемого элемента, который мог бы следовать за последним элементом вектора `v1`.

Если нужно удалить все элементы из вектора и сделать его пустым, то применяется вызов метода `clear`.

Теперь все необходимые методы класса `vector` рассмотрены, и можно вернуться к доработке программы.

Сначала нужно определиться с тем, какие переменные должны быть добавлены для сохранения информации о траекториях движения мыши. Каждая позиция курсора мыши описывается координатами `x` и `y`. В Win32 API есть структура `POINT`, которая позволяет объединять эти два значения. Поэтому, казалось бы, можно объявить следующий объект:

```
static vector<POINT> curve;
```

для накопления точек, описывающих текущую кривую линию. Но, к сожалению, в определении структуры `POINT` нет конструктора, который позволял бы использовать анонимные экземпляры класса¹ `POINT(x, y)`, и поэтому компилятор не пропустит выражения следующего вида:

```
curve.push_back(POINT(x, y));
```

Значит, нам нужно определить свою структуру `Point`, в которой устранен указанный недостаток:

```
struct Point {
    Point(int _x = 0, int _y = 0) : x(_x), y(_y) {}
    int x;
    int y;
};
```

Теперь определение объекта для накопления точек, представляющих текущую кривую линию, будет иметь следующий вид:

```
static vector<Point> curve;
```

Накопление заканчивается, как только пользователь отпускает кнопку мыши, и оконная процедура получает сообщение `WM_LBUTTONDOWN`. Сформированный вектор `curve` необходимо сохранить в другом векторе `curves`, который будет хранящим для накопления объектов `curve`:

```
static vector<vector<Point>> curves;
```

¹ Напомним, что структура является частным случаем класса, в котором все члены по умолчанию считаются открытыми (`public`). Анонимный экземпляр класса — это временный объект, создаваемый явным вызовом конструктора.

В каком-то смысле объект `curves` является аналогом двумерного массива. Его элементами являются объекты типа `vector<Point>`. Обратите внимание на пробел между двумя последними угловыми скобками. Если его не поставить, то компилятор выдаст сообщение об ошибке.

Текст приложения `ScribbleAdvanced`, являющегося улучшенной версией `Scribble`, приведен в листинге 4.3.

Листинг 4.3. Проект ScribbleAdvanced

```
////////////////////////////////////////////////////////////////////////
// ScribbleAdvanced.cpp
#include <windows.h>
#include <vector>
using namespace std;

#include "KWnd.h"

struct Point {
    Point(int _x = 0, int _y = 0) : x(_x), y(_y) {}
    int x;
    int y;
};

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====================================================
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("ScribbleAdvanced", hInstance, nCmdShow, WndProc);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

//=====================================================
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    static HDC hDC; // контекст устройства для рисования мышью
    HDC hdc; // контекст устройства для восстановления
              // рисунка при обработке WM_PAINT
    PAINTSTRUCT ps;
    static int x, y; // позиция курсора мыши
    static BOOL bTracking = FALSE;

    static vector<Point> curve;
    static vector<vector<Point>> curves;
    vector<Point>::iterator it;
    int i, j;

    switch (uMsg)
    {
        case WM_CREATE:
            hDC = GetDC(hWnd);
```

```
break;

case WM_LBUTTONDOWN:
    bTracking = TRUE;
    // начальная позиция
    x = LOWORD(lParam);
    y = HIWORD(lParam);
    MoveToEx(hdc, x, y, NULL);
    curve.push_back(Point(x, y));
    break;

case WM_LBUTTONUP:
    if (bTracking) {
        bTracking = FALSE;
        curves.push_back(curve);
        curve.clear();
    }
    break;

case WM_MOUSEMOVE:
    if (bTracking) {
        // новая позиция
        x = LOWORD(lParam);
        y = HIWORD(lParam);
        LineTo(hdc, x, y);
        curve.push_back(Point(x, y));
    }
    break;

case WM_PAINT:
    hdc = BeginPaint(hWnd, &ps);

    for (i = 0; i < curves.size(); ++i) {
        it = curves[i].begin();
        MoveToEx(hdc, it->x, it->y, NULL);
        for (it + 1; it != curves[i].end(); ++it)
            LineTo(hdc, it->x, it->y);
    }
    EndPaint(hWnd, &ps);
    break;

case WM_DESTROY:
    ReleaseDC(hWnd, hdc);
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
////////////////////////////////////////////////////////////////
```

Кроме тех дополнений, о которых упоминалось ранее, в этой программе появился также блок обработки сообщения WM_PAINT. Заметьте, что рисование здесь производится с дескриптором контекста дисплея hdc, который отличается от дескриптора hDC, используемого при обработке сообщений от мыши.

Обратите внимание на то, что доступ к элементам вектора `curves` осуществляется по индексу, а доступ к элементам его элементов (то есть к объектам типа `vector<Point>`) — с использованием итератора `it`. Здесь у вас может возникнуть вопрос: «Как же так, ранее говорилось, что значением элемента с адресом `it` является выражение `*it?` Да, все правильно, но учтите, что наш элемент является структурой типа `Point`, а в коде нужен доступ к ее полям `x` и `y`. Поэтому вызов функции `LineTo` мог бы выглядеть так:

```
LineTo(hdc, (*it).x, (*it).y);
```

Этот вызов тоже работал бы правильно. Но язык C++ содержит более удобную синтаксическую конструкцию для такого случая, позволяя вместо `(*it).x` записать `it->x`. Поэтому вызов `LineTo` имеет следующий вид:

```
LineTo(hdc, it->x, it->y);
```

Теперь можно протестировать усовершенствованную версию программы и убедиться в том, что рисунок сохраняется после изменений размеров окна.

Эластичные прямоугольники

Графические редакторы, как правило, поддерживают операцию выделения некоторой прямоугольной области. Пользователь нажимает левую кнопку мыши, фиксируя одну из вершин прямоугольника, и перемещает мышь к противоположной вершине. В процессе перемещения мыши построение фигуры считается еще не законченным, и она выводится пунктирным контуром, который все время изменяется. При отпускании кнопки мыши прямоугольник фиксируется, обозначаясь, например, другим цветом или другим стилем линии контура.

Приложение `ElasticRect`, приведенное в листинге 4.4, демонстрирует реализацию подобной операции. Кроме того, в нем показана обработка двойного щелчка мыши, по которому программа рисует квадрат фиксированного размера.

Рисование линий и фигур происходит на некотором фоне или на некотором изображении, которое расположено в клиентской области окна. Для простоты дальнейшего изложения этот фон или существующее изображение будет называться *холстом*.

В основе рисования эластичных прямоугольников обычно лежит следующая алгоритмическая идея. Получив сообщение `WM_MOUSEMOVE`, программа должна стереть предшествующее изображение прямоугольника, затем получить координаты новой позиции мыши и нарисовать новое изображение прямоугольника.

Проще всего стирание текущего изображения фигуры можно организовать, если в контексте устройства установлен режим рисования `R2_XORPEN` или `R2_NOTXORPEN`. В этом случае повторный вывод фигуры приводит к восстановлению того состояния холста, которое было перед первым выводом этой фигуры. Чем же различаются эти два режима или эти две растровые операции: `R2_XORPEN` и `R2_NOTXORPEN`?

Если фигура рисуется черным пером на белом холсте, то используйте бинарную растровую операцию `R2_NOTXORPEN`. Если белым пером на черном холсте, то — `R2_XORPEN`. Почему? Для ответа на этот вопрос рассмотрим реализацию этих операций в монохромной системе с кодировкой 1 бит/пикセル («0» — черный цвет, «1» — белый цвет), показанную в табл. 4.4 и 4.5.

Таблица 4.4. Операция R2_XORPEN

Холст	Перо	Результат операции R2_XORPEN	Примечание
0	0	0	
0	1	1	Цвет пера
1	0	1	Цвет холста
1	1	0	

Таблица 4.5. Операция R2_NOTXORPEN

Холст	Перо	Результат операции R2_NOTXORPEN	Примечание
0	0	1	
0	1	0	Цвет холста
1	0	0	Цвет пера
1	1	1	

Когда фигура рисуется первый раз, то наибольший интерес представляют ситуации, в которых цвет пера и холста различаются, что соответствует второй и третьей строкам в таблицах. Естественно, мы хотели бы в результате этого рисования получить цвет пера, а не холста. В графе «Примечание» показано, что такой результат достигается при рисовании белым пером на черном холсте для растровой операции R2_XORPEN и при рисовании черным пером на белом холсте для растровой операции R2_NOTXORPEN.

Теперь вернемся к представлению цвета в формате RGB. Так как для белого и черного цветов значения одного бита, приведенные в табл. 4.4 и 4.5, просто тиражируются по всем разрядам, то выявленные рекомендации остаются в силе.

Все сказанное выше справедливо для рисования фигур сплошным черным или белым пером (стиль PS_SOLID). Если же эластичный прямоугольник рисуется прерывистым пером (например, имеющим стиль PS_DOT), то ваша задача как программиста упрощается — вы можете использовать любую растровую операцию: и R2_XORPEN, и R2_NOTXORPEN. Также любая из этих операций может быть применена в случае рисования сплошным цветным пером.

Но хватит теории, пора предъявить код (листинг 4.4).

Листинг 4.4. Проект ElasticRect

```
//////////  
// ElasticRect.cpp  
#include <windows.h>  
#include "KWnd.h"  
  
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);  
=====  
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,  
    LPSTR lpCmdLine, int nCmdShow)  
{  
    MSG msg;  
    KWnd mainWnd("Elastic Rectangle", hInstance, nCmdShow, WndProc);  
  
    while (GetMessage(&msg, NULL, 0, 0)) {  
        TranslateMessage(&msg);
```

продолжение ↗

Листинг 4.4 (продолжение)

```
        DispatchMessage(&msg);
    }
    return msg.wParam;
}
//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    UINT style;
    static HDC hDC;
    static int x1, y1, x2, y2;
    static BOOL bTracking = FALSE;
    static HBRUSH hOldBrush;
    static HPEN hDotPen, hOldPen;

    switch (uMsg)
    {
    case WM_CREATE:
        style = GetClassLong(hWnd, GCL_STYLE);
        SetClassLong(hWnd, GCL_STYLE, style | CS_DBLCLKS);

        hDotPen = CreatePen(PS_DOT, 1, RGB(0,0,0));
        hDC = GetDC(hWnd);
        hOldBrush = (HBRUSH)SelectObject(hDC, GetStockObject(HOLLOW_BRUSH));
        break;

    case WM_LBUTTONDOWN:
        bTracking = TRUE;
        SetROP2(hDC, R2_NOTXORPEN);
        x1 = x2 = LOWORD(lParam);
        y1 = y2 = HIWORD(lParam);

        hOldPen = (HPEN)SelectObject(hDC, hDotPen);
        Rectangle(hDC, x1, y1, x2, y2);
        break;

    case WM_LBUTTONUP:
        if (bTracking) {
            bTracking = FALSE;
            SetROP2(hDC, R2_COPYPEN);
            // нарисовать окончательный прямоугольник
            x2 = LOWORD(lParam);
            y2 = HIWORD(lParam);
            SelectObject(hDC, hOldPen);
            Rectangle(hDC, x1, y1, x2, y2);
        }
        break;

    case WM_MOUSEMOVE:
        if (bTracking) {
            // стереть предшествующий прямоугольник
            Rectangle(hDC, x1, y1, x2, y2);

            // нарисовать новый прямоугольник
            x2 = LOWORD(lParam);
            y2 = HIWORD(lParam);
            Rectangle(hDC, x1, y1, x2, y2);
        }
    }
```

```
        break;

    case WM_LBUTTONDOWN:
        bTracking = FALSE;
        x1 = LOWORD(lParam);
        y1 = HIWORD(lParam);
        Rectangle(hdc, x1, y1, x1 + 100, y1 + 100);
        break;

    case WM_DESTROY:
        SelectObject(hdc, hOldBrush);
        ReleaseDC(hWnd, hdc);
        PostQuitMessage(0);
        break;

    default:
        return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }
    return 0;
}
////////////////////////////////////////////////////////////////
```

Особое внимание следует обратить на код обработки сообщений **WM_CREATE**, **WM_LBUTTONDOWN** и **WM_LBUTTONUP**.

В блоке обработки сообщения **WM_CREATE** модифицируется стиль класса окна при помощи добавления флага **CS_DBLCLKS**. Без этой модификации не будет обрабатываться двойной щелчок мыши. Затем создается прерывистое перо **hDotPen** со стилем **PS_DOT** и инициализируется контекст дисплея **hDC**, с которым будет идти вся дальнейшая работа. В контекст устройства выбирается пустая кисть (**HOLLOW_BRUSH**), чтобы рисуемые прямоугольники были прозрачными.

При обработке сообщения **WM_LBUTTONDOWN** программа устанавливает растровую операцию **R2_NOTXORPEN** и выбирает в контекст устройства прерывистое перо **hDotPen**. Дальнейшее рисование в блоке обработки сообщения **WM_MOUSEMOVE** будет идти этим пером с установленным режимом рисования.

При обработке сообщения **WM_LBUTTONUP** программа возвращает растровую операцию по умолчанию **R2_COPYPEN** и рисует окончательный вид прямоугольника пером по умолчанию (**hOldPen**).

Для упрощения кода опять была пропущена обработка сообщения **WM_PAINT**, и поэтому так же, как и в программе **Scribble**, нарисованные фигуры исчезают при изменении размеров окна. Но подход к решению этой проблемы был рассмотрен в предыдущей программе **ScribbleAdvanced**, поэтому здесь можно применить аналогичное решение.

Улучшенное приложение для просмотра текстовых файлов

Во второй главе при изучении полос прокрутки и вывода текста было создано приложение **TextViewer** (см. листинг 2.2), предназначенное для чтения и вывода на экран текстового файла. В том варианте приложения сообщения мыши не учитывались, но при работе с полосами прокрутки Windows берет на себя первичную обработку этих сообщений. В результате этой обработки Windows направляет окну

сообщения с префиксом `SB_`. Именно эти сообщения и обрабатывала оконная процедура приложения.

Теперь, вооруженные новыми знаниями, мы хотели бы усовершенствовать приложение `TextViewer`, добавив в него обработку колесика мыши, а также возможность для пользователя общаться с программой через клавиатурный интерфейс. Второе требование означает, что все операции, выполняемые мышью, должны быть продублированы на клавиатуре.

Есть два пути решения второй задачи. Обрабатывая сообщение `WM_KEYDOWN`, можно было бы просто продублировать полностью код, отвечающий за обработку сообщений `WM_VSCROLL` и `WM_HSCROLL`. Но это некрасиво и непрактично с точки зрения дальнейшего сопровождения программы. Второй подход заключается в том, что при появлении аппаратного сообщение клавиатуры, которое дублирует действие мыши, следует вызывать тот код, который уже был написан для обработки сообщений, начинающихся префиксом `SB_`. Фактически, это означает, что оконная процедура должна послать сама себе сообщение `SB_`.

Win32 API имеет несколько функций для отправки сообщений. В данной ситуации лучше использовать функцию `SendMessage`, имеющую следующий прототип:

```
HRESULT SendMessage(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);
```

Параметры функции имеют тот же смысл, что и параметры, передаваемые в оконную процедуру. Когда вызывается функция `SendMessage`, Windows вызывает оконную процедуру с дескриптором окна `hWnd`, передавая ей эти четыре параметра. После того как оконная процедура заканчивает обработку сообщения, Windows передает управление следующей за вызовом `SendMessage` инструкции.

В листинге 4.5 приводится текст усовершенствованного приложения для просмотра текстовых файлов¹.

Листинг 4.5. Проект TextViewerAdv

```
////////////////////////////////////////////////////////////////
// TextViewerAdv.cpp
#include <windows.h>
#include <zmouse.h> // для обработки колесика мыши

#include "KWnd.h"
#include "KDocument.h"

#define FILE_NAME "D:\\Program files\\Microsoft Visual
Studio\\VC98\\MFC\\SRC\\README.TXT"

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

KDocument doc;
//=====================================================
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
```

¹ Не забудьте добавить к проекту файлы `KWnd.h`, `KWnd.cpp`, `KDocument.h`, `KDocument.cpp`. Тексты последних двух файлов были приведены в листинге 2.2.

```
if (!doc.Open(FILE_NAME))
    return 0;

KWnd mainWnd("Text Viewer Advanced", hInstance, nCmdShow, WndProc, NULL,
    CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, CS_HREDRAW | CS_VREDRAW,
    WS_OVERLAPPEDWINDOW | WS_VSCROLL | WS_HSCROLL);

while (GetMessage(&msg, NULL, 0, 0)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
return msg.wParam;
}

////////// CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    TEXTMETRIC tm;
    int cxClient=0, cyClient=0;
    static int xInc, yInc;
    short status;

    switch (uMsg)
    {
        case WM_CREATE:
            hDC = GetDC(hWnd);
            GetTextMetrics(hDC, &tm);
            doc.Initialize(&tm);
            ReleaseDC(hWnd, hDC);
            break;

        case WM_SIZE:
            hDC = GetDC(hWnd);
            cxClient = LOWORD(lParam);
            cyClient = HIWORD(lParam);
            if (cxClient > 0)
                doc.ScrollSettings(hWnd, cxClient, cyClient);
            ReleaseDC(hWnd, hDC);
            break;

        case WM_VSCROLL:
            switch(LOWORD(wParam)) {
                case SB_LINEUP:
                    yInc = -1; break;
                case SB_LINEDOWN:
                    yInc = 1; break;
                case SB_PAGEUP:
                    yInc = -(int)doc.vsi.nPage; break;
                case SB_PAGEDOWN:
                    yInc = (int)doc.vsi.nPage; break;
                case SB_THUMBTRACK:
                    yInc = HIWORD(wParam) - doc.vsi.nPos; break;
                case SB_TOP:
                    yInc = -doc.vsi.nPos; break;
                case SB_BOTTOM:
```

продолжение ↗

Листинг 4.5 (продолжение)

```

yInc = doc.vsi.nMax - doc.vsi.nPos;    break;
default:
    yInc = 0;
}
doc.UpdateVscroll(hWnd, yInc);
break;

case WM_HSCROLL:
switch(LOWORD(wParam)) {
case SB_LINELEFT:
    xInc = -1;  break;
case SB_LINERIGHT:
    xInc = 1;  break;
case SB_PAGELEFT:
    xInc = -0.8 * (int)doc.hsi.nPage;  break;
case SB_PAGERIGHT:
    xInc = 0.8 * (int)doc.hsi.nPage;  break;
case SB_THUMBTRACK:
    xInc = HIWORD(wParam) - doc.hsi.nPos;  break;
case SB_TOP:
    xInc = -doc.hsi.nPos;  break;
case SB_BOTTOM:
    xInc = doc.hsi.nMax - doc.hsi.nPos;  break;
default:
    xInc = 0;
}
doc.UpdateHscroll(hWnd, xInc);
break;

case WM_MOUSEWHEEL:
if !(LOWORD(wParam) & MK_SHIFT) {
    xInc = -3 * (short)HIWORD(wParam) / WHEEL_DELTA;
    doc.UpdateHscroll(hWnd, xInc);
}
else {
    yInc = -3 * (short)HIWORD(wParam) / WHEEL_DELTA;
    doc.UpdateVscroll(hWnd, yInc);
}
break;

case WM_KEYDOWN:
switch (wParam) {
case VK_UP:    SendMessage(hWnd, WM_VSCROLL, SB_LINEUP, 0);
    break;
case VK_DOWN:  SendMessage(hWnd, WM_VSCROLL, SB_LINEDOWN, 0);
    break;
case VK_LEFT:   SendMessage(hWnd, WM_HSCROLL, SB_LINELEFT, 0);
    break;
case VK_RIGHT:  SendMessage(hWnd, WM_HSCROLL, SB_LINERIGHT, 0);
    break;
case VK_PRIOR:  SendMessage(hWnd, WM_VSCROLL, SB_PAGEUP, 0);
    break;
case VK_NEXT:   SendMessage(hWnd, WM_VSCROLL, SB_PAGEDOWN, 0);
    break;

case VK_HOME:
    status = GetKeyState(VK_CONTROL);
}

```

```
    if (0x80 & status) SendMessage(hWnd, WM_VSCROLL, SB_TOP, 0);
    else                  SendMessage(hWnd, WM_HSCROLL, SB_TOP, 0);
    break;

case VK_END:
    status = GetKeyState(VK_CONTROL);
    if (0x80 & status) SendMessage(hWnd, WM_VSCROLL, SB_BOTTOM, 0);
    else                  SendMessage(hWnd, WM_HSCROLL, SB_BOTTOM, 0);
    break;
}
break;

case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);
    doc.PutText(hWnd, hDC);
    EndPaint(hWnd, &ps);
    break;

case WM_DESTROY:
    PostQuitMessage(0);
    break;

default:
    return (DefWindowProc(hWnd, uMsg, wParam, lParam));
}
return 0;
}
////////////////////////////////////////////////////////////////
```

В этом приложении добавлены обработка сообщения **WM_MOUSEWHEEL** и обработка сообщения **WM_KEYDOWN**.

Обратите особое внимание на *преобразование* к типу *short* выражения (*short* **HIWORD(wParam)**) в блоке обработки сообщения **WM_MOUSEWHEEL**. Если этого не сделать, вы будете поражены неприятным поведением программы.

При обработке сообщения **WM_MOUSEWHEEL** учитывается состояние клавиши **Shift**. Если эта клавиша не нажата, то прокрутка колесика мыши управляет вертикальным скроллингом, если нажата — горизонтальным скроллингом.

Сравните работу первоначального варианта приложения для просмотра текстовых файлов **TextViewer** и новой программы **TextViewerAdv**.

5

Ресурсы Windows-приложения

Ресурсы являются составной частью приложений для Windows. В них определяются такие объекты, как пиктограммы, курсоры, растровые образы, таблицы строк, меню, диалоговые окна и многие другие.

Для некоторых видов ресурсов система содержит предопределенные (стандартные) объекты. Например, в листинге 1.1 уже использовались стандартная пиктограмма и стандартный курсор, для чего задавались соответствующие значения тем или иным полям класса окна.

Все нестандартные ресурсы должны быть определены в *файле описания ресурсов* (*resource script*), который является ASCII-файлом с расширением .rc. Хотя теоретически такой файл можно подготовить в обычном текстовом редакторе, подобная технология используется крайне редко, поскольку любая интегрированная среда содержит удобные *редакторы ресурсов*, максимально упрощающие и автоматизирующие этот процесс.

Когда программист начинает работу над новым проектом, обычно в нем нет никаких ресурсов. При попытке создать новый ресурс или импортировать существующий ресурс интегрированная среда MS Visual Studio 6.0 вызывает соответствующий редактор ресурсов. После создания ресурса его нужно сохранить в составе проекта при помощи команды меню *File ▶ Save*. Среда разработки Visual Studio предлагает для файла описания ресурсов имя по умолчанию *script1.rc*. Традиционно программисты предпочитают давать файлу описания ресурсов то же имя, которое используется и для самой программы (например, *MyApp.rc*).

Файл описания ресурса транслируется *компилятором ресурсов* (файл *rc.exe* в составе интегрированной среды). В результате образуется бинарный файл с расширением .res. Затем компоновщик включает полученный файл в выполняемый файл программы вместе с обычным кодом и данными программы из файлов с расширениями .obj и .lib. При работе в среде Visual Studio после выполнения команды *Build* все эти шаги производятся автоматически.

При загрузке в память исполняемого кода программы Windows обычно оставляет ресурсы на диске. Потом они загружаются в память только по мере необходимости.

Редакторы ресурсов

Редакторы ресурсов содержат инструменты и интерфейсы для быстрого и удобного обслуживания ресурсов приложения. В составе MS Visual Studio 6.0 имеется следующий набор редакторов ресурсов:

- графический редактор, который помогает создавать пиктограммы, курсоры, растровые образы;
- редактор меню;
- редактор таблиц быстрых клавиш;
- редактор диалоговых окон;
- редактор таблиц строк;
- редактор панелей инструментов;
- редактор информации о версии;
- редактор ресурсов, определяемых программистом.

Работа с конкретными редакторами описывается в процессе изучения тех ресурсов, которые они обслуживают.

В этой главе рассматриваются пиктограммы, курсоры, растровые образы, таблицы строк и ресурсы, определяемые программистом¹. Кроме этого приводятся функции для воспроизведения звуковых файлов. В главах 6, 7 и 8 рассматриваются ресурсы меню, окна диалога и панели инструментов соответственно.

Пиктограммы

Пиктограммы — это небольшие растровые изображения, применяемые Windows для визуального представления приложений, файлов и каталогов. Пиктограмма, включенная в состав приложения, выводится на экран в левом верхнем углу строки заголовка окна приложения. Кроме того, Windows выводит на экран значок пиктограммы в списке программ меню **Start** (Пуск), на панели задач, когда окно приложения свернуто, и в списке файлов, отображаемых программой Windows Explorer в открытой папке. Также пиктограммы отображаются на рабочем столе, предоставляя доступ к системным папкам и к ярлыкам исполняемых файлов.

Ранние версии Windows могли отображать пиктограммы размером только 32×32 пикселя. Начиная с Windows 95, пиктограммы могут иметь один из трех типовых размеров: 16×16 пикселов для *малых пиктограмм*, 32×32 пикселов для *стандартных пиктограмм* и 48×48 пикселов для *больших пиктограмм*. Обычно малые и стандартные пиктограммы являются 16-цветными, а большие пиктограммы могут использовать 256 цветов. Маленький значок используется в заголовке приложений, на панели задач и в списках программ меню **Start**. Пиктограммы,

¹ В литературе и документации (MSDN) принят термин «ресурсы, определяемые пользователем». Но он не очень удачен, так как в качестве *пользователя* здесь подразумевается именно *программист*. Это приводит к путанице, когда в контексте упоминается *настоящий пользователь* — человек, эксплуатирующий вашу программу. Поэтому мы будем применять термин «ресурсы, определяемые программистом».

отображающиеся на рабочем столе, имеют стандартный размер. Для окон папок, открываемых Windows Explorer, пользователь может настроить режим показа маленьких или стандартных значков.

Большие значки (48×48) могут появиться вместо стандартных значков, если только изменить настройку для размера пиктограмм в системном реестре Windows. Пользователь может сделать это при помощи диалогового окна, которое отображается после выполнения команды Панель управления ▶ Экран ▶ Оформление. В диалоговом окне Свойства: Экран следует выбрать тип элемента Значок и установить нужную величину в окне Размер.

Графический редактор Visual Studio, используемый в режиме создания пиктограммы, формирует файл с расширением .ico, содержащий один или несколько значков из следующего списка:

- Стандартный формат 32×32 , 16 цветов (вариант по умолчанию).
- Стандартный формат 32×32 , 256 цветов.
- Большой формат 48×48 , 256 цветов.
- Монохромный формат 32×32 , черно-белая пиктограмма.
- Маленький формат 16×16 , 16 цветов.

Создавая приложение, разработчик может включить в него только стандартную пиктограмму. В этом случае Windows сформирует маленький значок из стандартного, просто удалив каждый второй столбец и каждую вторую строку. Но иногда качество такого значка получается неудовлетворительным. В таких случаях стоит добавить изображение малой пиктограммы и соответствующий код для ее загрузки.

Рассмотрим процесс проектирования конкретного приложения, в котором используется ресурс пиктограммы. Потом в это приложение будут включены ресурсы специализированных курсоров. На следующем этапе к приложению будет добавлен ресурс битового образа с изображением государственного флага России. В последней версии программы мы заставим ее проигрывать звуковой файл с государственным гимном России. Итак, приступим...

Создайте новый проект типа Win32 Application с именем Russia.

Добавьте к нему новый файл с именем Russia.cpp, а затем введите в файл код, приведенный в листинге 5.1.

Листинг 5.1. Проект Russia (начальная редакция)

```
//////////  
// Russia.cpp  
#include <windows.h>  
#include "KWnd.h"  
  
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);  
=====  
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,  
    LPSTR lpCmdLine, int nCmdShow)  
{  
    MSG msg;  
    KWnd mainWnd("Russia today", hInstance, nCmdShow, WndProc, NULL, 0, 0,  
        400, 300);  
  
    while (GetMessage(&msg, NULL, 0, 0)) {  
        TranslateMessage(&msg);
```

```

        DispatchMessage(&msg);
    }
    return msg.wParam;
}
//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_DESTROY:
        PostQuitMessage(0);
        break;

    default:
        return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }
    return 0;
}
///////////

```

Скопируйте в папку проекта Russia файлы KWnd.h и KWnd.cpp, которые использовались в проекте Hello2 (листинг 1.2), и добавьте их в состав проекта Russia. После этого рабочий стол Visual Studio будет выглядеть примерно так, как показано на рис. 5.1.

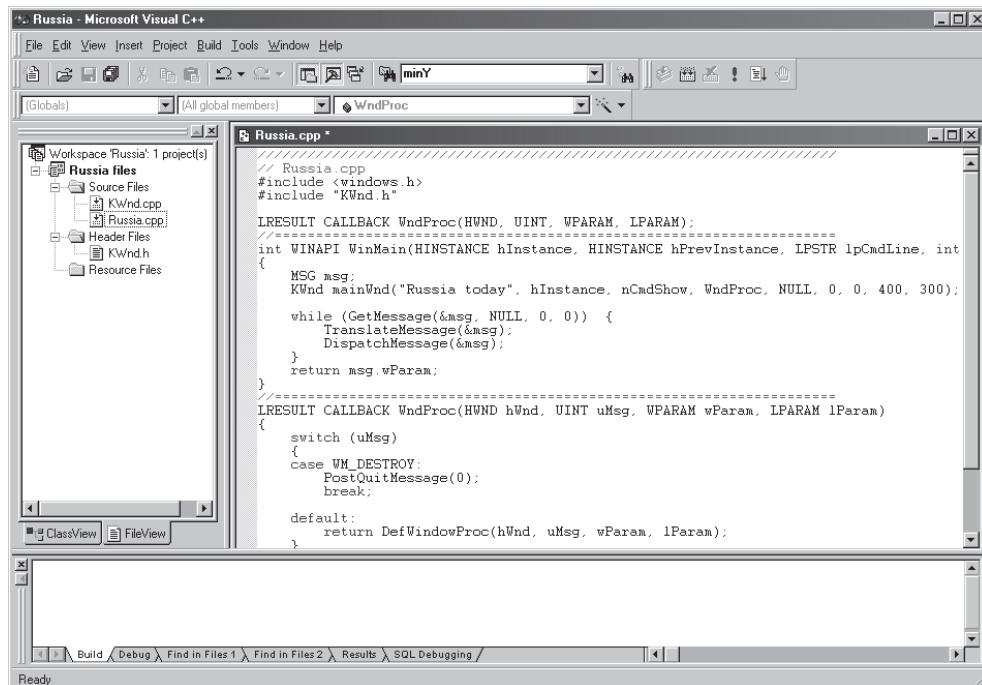


Рис. 5.1. Рабочий стол Visual Studio после создания проекта Russia

Обратите внимание на то, что окно **Workspace** содержит только две вкладки: **ClassView** и **FileView**. В настоящий момент открыта вкладка **FileView** с древовидным списком файлов, включенных в состав проекта.

Создание пиктограммы с помощью графического редактора

Общий сценарий добавления к проекту новой пиктограммы включает следующие шаги:

- Вызов графического редактора.** В главном меню Visual Studio выполните команду **Insert ▶ Resource**. Это приведет к отображению диалогового окна **Insert Resource** (рис. 5.2).

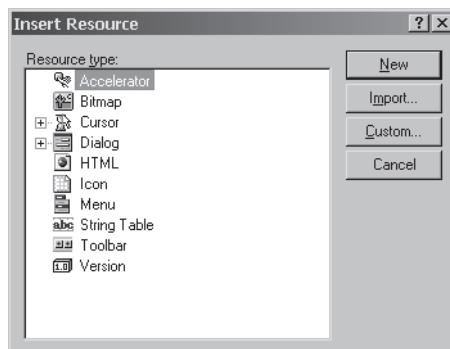


Рис. 5.2. Диалоговое окно Insert Resource

В окне нужно выбрать тип ресурса **Icon** и нажать кнопку **New**. Появится окно графического редактора **Script1 — IDI_ICON1** с заготовкой стандартной пиктограммы (рис. 5.3).

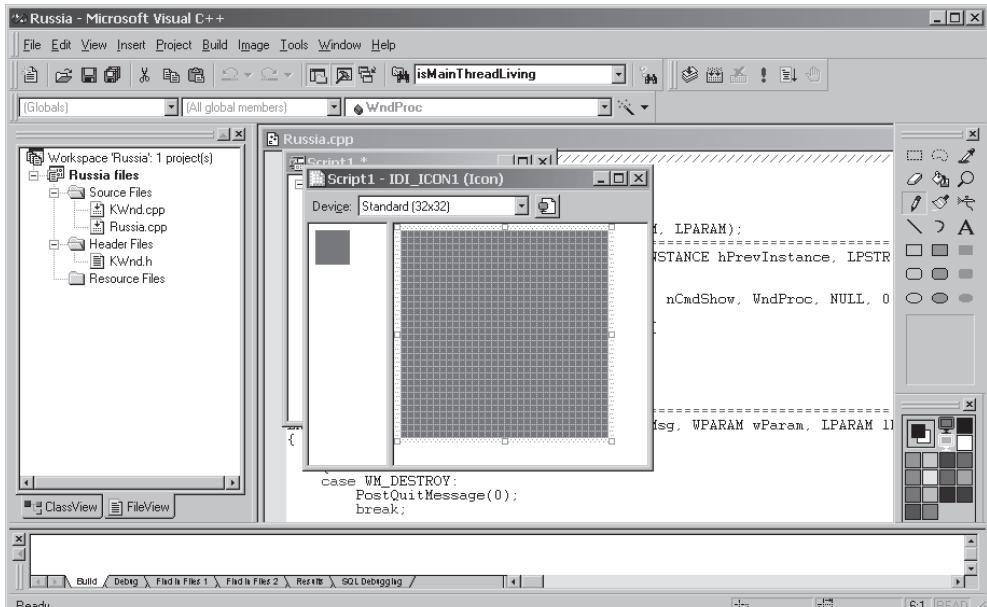


Рис. 5.3. Рабочий стол Visual Studio после вызова графического редактора

Обратите внимание на две панели инструментов в правой части рабочего стола. Панель **Graphics** содержит инструменты рисования, а панель **Colors** — палитру доступных цветов. Если по каким-либо причинам эти панели не будут видны, нужно щелкнуть правой кнопкой мыши на их месте и в появившемся контекстном меню включить соответствующие флагки **Graphics** и **Colors**.

2. **Выбор формата пиктограммы.** Комбинированный список **Device** в окне графического редактора поначалу содержит только один формат **Standard (32×32)**. Но справа от этого списка располагается кнопка **New Device Image**. Если ее нажать, то появится окно **New Icon Image** со списком **Target device** (рис. 5.4).

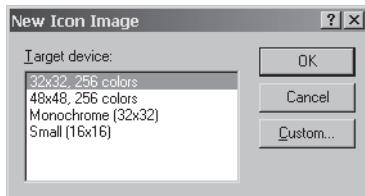


Рис. 5.4. Диалоговое окно New Icon Image

Вы можете выбрать из списка любой дополнительный формат и, нажав кнопку **OK**, добавить его в список **Device**.

Для программы **Russia.cpp** нужно создать стандартную и малую пиктограммы. Поэтому выберите в предложенном списке строку **Small(16×16)** и нажмите кнопку **OK**. Окно графического редактора переключится на заготовку малой пиктограммы.

3. **Назначение ресурсу идентификатора.** Обратите внимание на заголовок в окне графического редактора: **Script1 — IDI_ICON1 (Icon)**. Первая часть заголовка содержит имя файла описания ресурсов. По умолчанию редактор предлагает имя **Script1**, если создаваемый ресурс является первым в данном проекте. Если же некий ресурс уже добавлялся ранее, то на этом месте отображается то имя, под которым был сохранен файл описания ресурсов (см. п. 5).

Вторая часть заголовка содержит идентификатор создаваемого ресурса. По умолчанию редактор предлагает имя **IDI_ICON1**. Если вы будете добавлять в проект другие пиктограммы, то редактор предложит идентификаторы по умолчанию **IDI_ICON2**, **IDI_ICON3** и т. д. Для долгосрочного сопровождения проекта такая система наименований — не самое лучшее решение. Поэтому рекомендуется присваивать ресурсам идентификаторы, отражающие их семантику. В создаваемой программе пиктограмма будет изображать трехцветный российский флаг, поэтому более подходящим оказался бы идентификатор **IDI_TRICOLOUR**.

Для изменения идентификатора нужно дважды щелкнуть левой кнопкой мыши на пустом месте в окне редактирования. На экране появится окно **Icon Properties** (рис. 5.5).

На вкладке **General** располагается текстовое поле **ID** для ввода идентификатора и текстовое поле **File name** с именем файла, в котором будет сохранено изображение пиктограммы. Введем для нашей программы идентификатор ресурса **IDI_TRICOLOUR** и изменим имя файла на **tricol.ico**.

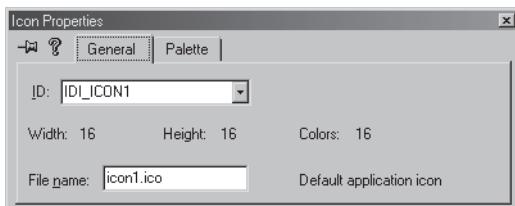


Рис. 5.5. Окно Icon Properties

Прежде чем закрыть это окно, переключитесь на вкладку **Palette**. Здесь вы увидите 16 доступных цветов для изображения пиктограммы. Это те же самые цвета, которые содержит панель инструментов **Colors**. Но если вы будете работать с 256-цветным форматом, то палитра будет содержать 256 цветов, и выбирать текущий цвет для рисования можно будет только здесь.

Закройте окно **Icon Properties**, нажав кнопку с крестиком в полосе заголовка. После этого заголовок окна графического редактора изменится на **Script1 — IDI_TRICOLOUR**.

4. **Рисование.** Сейчас в окне графического редактора находится заготовка малой пиктограммы (16×16). Так же, как на рис. 5.3, она представляет собой квадрат, все пиксели которого окрашены в темно-бирюзовый (dark-cyan) цвет. На самом деле пиксели этого цвета интерпретируются графическим редактором как прозрачные¹.

Пользуясь инструментами рисования на панели **Graphics**, создайте нужный рисунок. Текущий цвет выбирается на панели **Colors**. Процедура рисования очень похожа на создание рисунка в графическом редакторе MS Paint, поэтому не будем подробно ее описывать. Результат рисования малой пиктограммы показан на рис. 5.6.

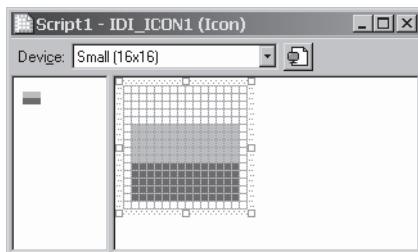


Рис. 5.6. Окно графического редактора с изображением малой пиктограммы

Черно-белый рисунок не передает всей информации. На самом деле верхняя часть изображения содержит белый цвет, средняя часть — голубой цвет и нижняя — красный цвет. Обрамляет пиктограмму рамка белого цвета толщиной 1 пикселя².

¹ Вы можете проверить это, оставив на каком-нибудь изображении часть пикселов заготовки в исходном состоянии.

² Если вы загрузите исходный код проекта *Russia* из файлов к книге, доступных на сайте издательства «Питер» (www.piter.com), то сможете увидеть все ресурсы-изображения в их истинном цвете.

Теперь создайте изображение стандартной пиктограммы (32×32), переключившись на ее заготовку при помощи списка Device. Размеры стандартной пиктограммы позволяют сделать ее более информативной. На средней части флага надо нарисовать текстовую надпись «Russia», а рамку значка выделить светло-серым цветом (рис. 5.7).

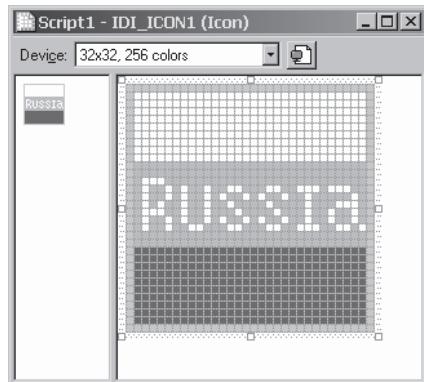


Рис. 5.7. Окно графического редактора с изображением стандартной пиктограммы

Если вам понадобится создать рисунок, использующий 256-цветную палитру, то цвет нужно будет выбирать на вкладке Palette в окне Icon Properties (см. рис. 5.5).

5. **Сохранение созданного ресурса в файле описания ресурсов.** В главном меню Visual Studio выполните команду File ▶ Save.

Если создаваемый ресурс — не первый и файл описания ресурсов уже существует, то описание созданного ресурса будет просто добавлено в существующий файл.

Если создаваемый ресурс является первым в проекте, то появится окно диалога Save As (Сохранить как). По умолчанию в окне File Name (Имя файла) предлагается имя Script1.rc. Замените часть имени Script1 на имя приложения, в данном случае на имя Russia, и нажмите кнопку Save (Сохранить). Теперь в заголовке окна графического редактора будет отображаться строка Russia.rc — IDI_TRICOLOUR.

Теперь необходимо добавить файл описания ресурсов к проекту. Для этого в окне Workspace щелкните правой кнопкой мыши на папке Resource Files и в открывшемся контекстном меню выберите пункт Add Files to Folder. В открывшемся окне диалога Insert Files нужно выбрать необходимый файл. В нашем примере к проекту добавляется файл Russia.rc, и после этого рабочий стол Visual Studio будет иметь вид, показанный на рис. 5.8.

Обратите внимание на изменения, произошедшие на вкладке FileView. Папка Resource Files теперь содержит два файла: файл описания ресурсов Russia.rc и графический файл tricol.ico. Кроме того, в окне Workspace появилась новая вкладка ResourceView. Если переключиться на нее, то можно увидеть список ресурсов проекта, содержащий пока только папку Icon с единственным ресурсом IDI_TRICOLOUR.

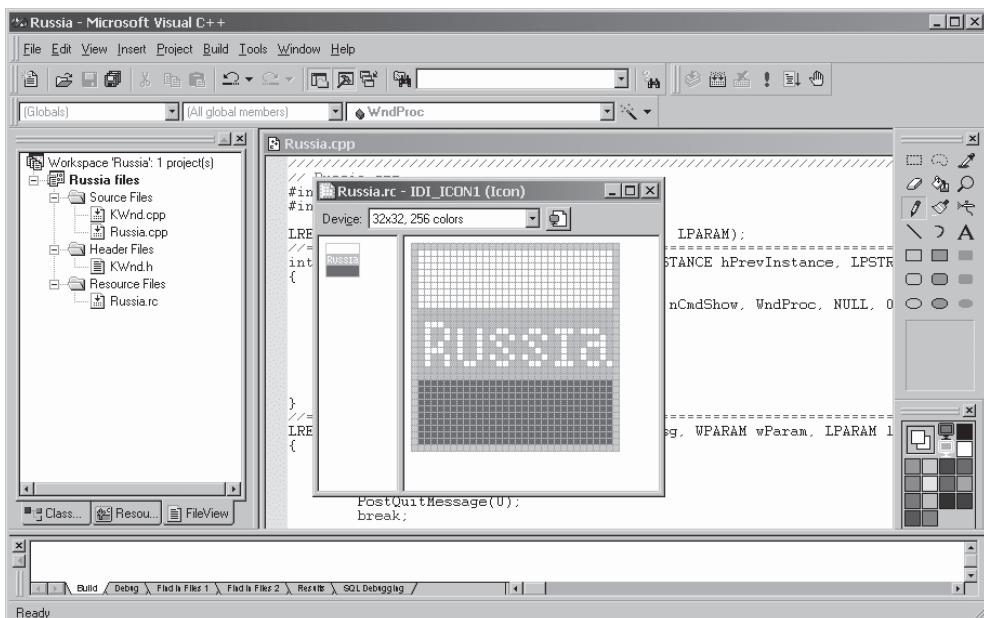


Рис. 5.8. Рабочий стол Visual Studio после добавления к проекту файла описания ресурсов

6. **Добавление к проекту заголовочного файла resource.h** (только для первого создаваемого ресурса). Наряду с файлом описания ресурсов, редактор ресурсов создает еще и заголовочный файл resource.h, содержащий определения используемых именованных констант. Если создаваемый ресурс — первый, то кроме добавления к проекту файла описания ресурсов надо также добавить и указанный заголовочный файл. Для этого в окне Workspace щелкните правой кнопкой мыши на папке Header Files и в открывшемся контекстном меню выполните команду Add Files to Folder. Далее в открывшемся окне диалога Insert Files следует выбрать файл resource.h.

ПРИМЕЧАНИЕ

Пока заголовочный файл resource.h используется только в одном проекте, нас устраивает его имя, создаваемое редактором ресурсов. Если же проект предназначен для создания динамически загружаемой библиотеки (DLL), экспортирующей ресурсы в другие проекты, то следует побеспокоиться об уникальности имени ее заголовочного файла. Более подробно этот вопрос рассматривается в главе 11.

На этом завершается рассмотрение процедуры создания и включения в проект новых пиктограмм. В разделе «Использование ресурса в приложении» будет показано, что нужно добавить в код программы, чтобы эти пиктограммы стали значками приложения.

Импорт существующей пиктограммы

Изложенный выше сценарий описывает создание новой пиктограммы. Но графический редактор позволяет добавлять к программе уже существующую пиктограмму, позаимствованную из другого приложения.

Предположим, что вы хотите использовать в своей программе пиктограмму, изображение которой находится в файле AnyIcon.ico. Для начала требуется скопировать файл AnyIcon.ico в папку вашего приложения.

После этого выполните следующую последовательность действий:

- ❑ В главном меню Visual Studio выполните команду **Insert ▶ Resource**. В появившемся диалоговом окне **Insert Resource** выберите тип ресурса **Icon** и нажмите кнопку **Import**.
- ❑ В появившемся диалоговом окне **Import Resource** убедитесь, что открыта папка вашего приложения, и щелкните на имени файла **AnyIcon.ico**. В результате будет открыто окно графического редактора с изображением импортируемой пиктограммы.
- ❑ Назначьте идентификатор ресурсу пиктограммы в соответствии с п. 3 описанного выше сценария.
- ❑ Сохраните ресурс в проекте и выполните действия по модификации состава проекта, как это показано в пп. 5 и 6 описанного выше сценария.

Изложенная процедура описывает использование графического редактора в *режиме импорта*. Аналогично могут импортироваться и другие графические ресурсы, например, курсоры и растровые изображения.

Просмотр и редактирование ресурсов приложения

Ресурсы приложения можно просматривать в двух разных режимах, вызывая соответствующий графический редактор или открывая файл описания ресурсов в текстовом режиме.

Просмотр ресурса с вызовом соответствующего редактора

В окне **Workspace** перейдите на вкладку **ResourceView** и сделайте двойной щелчок мышью на идентификаторе интересующего вас ресурса. При этом будет открыт соответствующий редактор ресурса, и в его окне появится изображение, связанное с этим ресурсом, как, например, показано на рис. 5.8. После этого ресурс можно редактировать при помощи средств редактора. Если в определение ресурса были внесены изменения, то они должны быть сохранены при помощи команды меню **File ▶ Save**.

Просмотр файла описания ресурсов в текстовом режиме

Закройте все окна тех редакторов ресурсов, которые открыты в настоящий момент. После этого в главном меню Visual Studio выполните команду **File ▶ Open**. В результате появится окно диалога **Open** (Открыть), внешний вид которого показан на рис. 5.9.

Убедитесь, что в текстовом поле **Папка** выбрано имя папки текущего проекта (в данном случае — **Russia**). Если это не так, то следует найти искомую папку в комбинированном списке **Папка**. Затем откройте комбинированный список **Open as** и выберите строку **Text**. После этого останется только щелкнуть на имени нужного файла (в данном примере **Russia.rc**). В результате в окне редактора Visual Studio появится текст файла описания ресурсов.

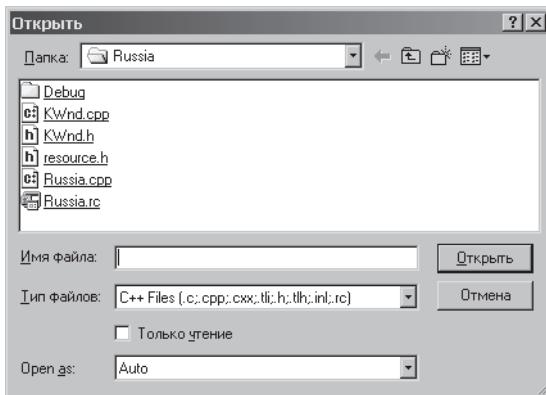


Рис. 5.9. Диалоговое окно Open

Не приводя здесь полного текста файла *Russia.rc*, обратим ваше внимание на следующую строку:

```
IDI_TRICOLOUR      ICON      DISCARDABLE    "tricol.ico"
```

в которой содержится описание ресурса созданной пиктограммы.

Это так называемое *однострочное описание ресурса*. Оно применяется для описания пиктограмм, курсоров и битовых образов. Синтаксис подобного описания выглядит следующим образом:

```
имя_ресурса    тип_ресурса    DISCARDABLE    имя_файла
```

Имя_ресурса интерпретируется Windows либо как C-строка с завершающим нулевым символом, либо как целочисленный идентификатор. Если *имя_ресурса* представляет собой ASCII-изображение десятичного или шестнадцатеричного числа, то Windows воспринимает его как целочисленный идентификатор. В противном случае *имя_ресурса* интерпретируется как C-строка, хотя формальных признаков строки (обрамляющих кавычек) у этого имени нет.

Похоже, что вариант C-строки использовался в ранних версиях Windows, поскольку редакторы ресурсов Visual Studio в генерируемом ими коде всегда определяют имя ресурса в виде целочисленной константы. Так, если заглянуть в текст файла *resource.h* в проекте *Russia*, можно найти там следующую строку:

```
#define IDI_TRICOLOUR 101
```

То есть на самом деле после подстановки макроопределения описание пиктограммы будет иметь следующий вид:

```
101      ICON      DISCARDABLE    "tricol.ico"
```

Таким образом, однострочное описание ресурса, находящееся в файле *Russia.rc*, связывает пиктограмму *IDI_TRICOLOUR* с графическим файлом *tricol.ico*, содержащим два растровых изображения: стандартной и малой пиктограмм.

Атрибут *DISCARDABLE* (выгружаемый) использовался в ранних версиях Windows, позволяя системе освобождать занимаемую ресурсом оперативную память, когда возникали проблемы со свободной памятью. В Win32 он фактически игнорируется.

Использование ресурса в приложении

Для получения дескриптора пиктограммы, относящегося к типу HICON, вызывается функция LoadIcon или LoadImage.

Беглое знакомство с функцией LoadIcon у нас уже состоялось в главе 1. Но здесь она будет рассмотрена более подробно. Функция LoadIcon загружает ресурс пиктограммы из выполняемого файла (.exe), ассоциированного с экземпляром приложения hInstance:

```
HICON LoadIcon(HINSTANCE hInstance, LPCTSTR lpIconName);
```

В качестве второго параметра, определяющего имя ресурса, функция принимает строку с завершающим нулевым символом. Ранее говорилось, что в файле описания ресурсов, подготовленном с помощью редактора ресурсов, *имя_ресурса* для пиктограммы представляет собой целочисленный идентификатор. Для решения проблемы преобразования целого числа в указатель на строку ресурса необходимо использовать макрос MAKEINTRESOURCE (make an integer into resource string), определенный в файле winuser.h следующим образом:

```
#define MAKEINTRESOURCE(i) (LPSTR)((DWORD)((WORD)(i)))
```

Этот макрос преобразует число в указатель, но при этом старшие 16 разрядов устанавливаются в нулевое значение. Так Windows узнает, что второй параметр функции LoadIcon является числом, а не указателем на символьную строку.

Например, для загрузки пиктограммы в программе Russia будет использоваться следующий вызов:

```
LoadIcon(hInstance, MAKEINTRESOURCE(IDI_TRICOLOUR));
```

Если параметр hInstance имеет значение NULL, то функция LoadIcon используется для загрузки предопределенных пиктограмм. Возможные значения второго аргумента для предопределенных пиктограмм приведены в табл. 1.5 (глава 1). Эта возможность использовалась во всех предыдущих приложениях, так как в классе KWnd поле hIcon оконного класса wc инициализируется при помощи вызова LoadIcon(NULL, IDI_APPLICATION).

В случае успешного завершения функция LoadIcon возвращает дескриптор загруженной пиктограммы. В случае неудачи возвращается значение NULL. Если по каким-то причинам LoadIcon не смогла загрузить указанную пиктограмму, то она загружает пиктограмму по умолчанию IDI_WINLOGO.

Обращаем ваше внимание на то, что функция LoadIcon предназначена для загрузки только стандартных пиктограмм. Поэтому не пытайтесь загрузить с ее помощью малую пиктограмму (16 × 16). Иначе вас ждет горькое разочарование. Для загрузки пиктограмм других размеров используйте функцию LoadImage.

Функция LoadImage предназначена для загрузки изображений разных типов: пиктограмм, курсоров, битовых образов. Она имеет следующий прототип:

```
HANDLE LoadImage(HINSTANCE hinst, LPCTSTR lpszName, UINT uType,  
int cxDesired, int cyDesired, UINT fuLoad);
```

Второй параметр функции определяет загружаемое изображение. Если параметр hinst не равен NULL и параметр fuLoad не содержит флаг LR_LOADFROMFILE, то параметр lpszName интерпретируется аналогично второму параметру функции LoadIcon.

Если параметр `hinst` равен `NULL`, а параметр `fuLoad` по-прежнему не содержит флаг `LR_LOADFROMFILE`, то параметр `lpszName` специфицирует OEM-изображение¹. Идентификаторы OEM-изображений определены в `winuser.h` и имеют следующие префиксы: `OBM_` для растров, `OIC_` для пиктограмм и `OCR_` для курсоров.

Если параметр `hinst` равен `NULL`, а параметр `fuLoad` содержит флаг `LR_LOADFROMFILE`, то `lpszName` задает имя файла, в котором хранится изображение загружаемого ресурса.

Третий параметр, `uType`, определяет тип изображения и может принимать значения `IMAGE_BITMAP`, `IMAGE_CURSOR` и `IMAGE_ICON`.

Четвертый параметр, `cxDesired`, задает ширину изображения в пикселях. Если он равен нулю и параметр `fuLoad` содержит флаг `LR_DEFAULTSIZE`, то для вычисления ширины изображения функция использует значения метрики `SM_CXICON` или `SM_CXCURSOR`. Если же параметр равен нулю, а флаг `LR_DEFAULTSIZE` не используется, то функция использует фактическую ширину изображения.

Пятый параметр, `cyDesired`, задает высоту изображения в пикселях. Если он равен нулю и параметр `fuLoad` содержит флаг `LR_DEFAULTSIZE`, то для вычисления высоты изображения функция использует значения метрик `SM_CXICON` или `SM_CXCURSOR`. Если же параметр равен нулю, а флаг `LR_DEFAULTSIZE` не используется, то функция использует фактическую высоту изображения.

Шестой параметр, `fuLoad`, позволяет указывать опции загрузки. Он может содержать один или несколько флагов, указанных в табл. 5.1 (приведены наиболее часто употребляемые флаги. Полный список см. в MSDN).

Таблица 5.1. Опции загрузки для функции `LoadImage`

Флаг	Значение
<code>LR_DEFAULTCOLOR</code>	Флаг по умолчанию. Означает, что изображение не монохромное
<code>LR_CREATEDIBSECTION</code>	Если <code>uType</code> имеет значение <code>IMAGE_BITMAP</code> , то функция возвращает DIB-секцию
<code>LR_DEFAULTSIZE</code>	Влияет на интерпретацию четвертого и пятого параметров функции
<code>LR_LOADFROMFILE</code>	Влияет на интерпретацию второго параметра
<code>LR_MONOCHROME</code>	Изображение является черно-белым

Вернемся к нашей программе `Russia`.

Чтобы обе загруженные пиктограммы — стандартная и малая — стали значками данного приложения, их дескрипторы должны быть присвоены полям `hIcon` и `hIconSm` структуры оконного класса главного окна приложения. Но так как в нашей программе главное окно создается при помощи объекта класса `KWnd`, то эти поля уже проинициализированы в конструкторе класса `KWnd` (см. текст файла `KWnd.cpp`). Поэтому решение заключается в модификации оконного класса при помощи функции `SetClassLong`. Модификация оконного класса производится всегда только в блоке обработки сообщения `WM_CREATE`, когда окно уже создано, но еще не показано на экране.

Новая редакция кода `Russia.cpp` приведена в листинге 5.2.

¹ OEM — Original Equipment Manufacturers (производители оригинального оборудования).

Листинг 5.2. Проект Russia (1-я редакция)

```
//////////  
// Russia.cpp  
#include <windows.h>  
#include "KWnd.h"  
#include "resource.h"  
  
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);  
=====  
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,  
    LPSTR lpCmdLine, int nCmdShow)  
{  
    MSG msg;  
    KWnd mainWnd("Russia today", hInstance, nCmdShow, WndProc, NULL, 0, 0,  
        400, 300);  
  
    while (GetMessage(&msg, NULL, 0, 0)) {  
        TranslateMessage(&msg);  
        DispatchMessage(&msg);  
    }  
    return msg.wParam;  
}  
=====  
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)  
{  
    HINSTANCE hInst;  
    HICON hIcon; // дескриптор стандартной пиктограммы  
    HICON hIconSm; // дескриптор малой пиктограммы  
  
    switch (uMsg)  
    {  
        case WM_CREATE:  
            hInst = GetModuleHandle(NULL);  
            hIcon = LoadIcon(hInst, MAKEINTRESOURCE(IDI_TRICOLOUR));  
            hIconSm = (HICON)LoadImage(hInst, MAKEINTRESOURCE(IDI_TRICOLOUR),  
                IMAGE_ICON, 16, 16, LR_DEFAULTCOLOR);  
            SetClassLong(hWnd, GCL_HICON, (LONG)hIcon);  
            SetClassLong(hWnd, GCL_HICONSM, (LONG)hIconSm);  
            break;  
  
        case WM_DESTROY:  
            PostQuitMessage(0);  
            break;  
  
        default:  
            return DefWindowProc(hWnd, uMsg, wParam, lParam);  
    }  
    return 0;  
}  
//////////
```

В блоке обработки сообщения **WM_CREATE** вызывается функция **LoadIcon** для загрузки стандартной пиктограммы и функция **LoadImage** для загрузки малой пиктограммы. Дескриптор экземпляра приложения **hInst**, который передается функциям, определяется при помощи функции **GetModuleHandle**.

Полученные дескрипторы `hIcon` и `hIconSm` передаются затем двум вызовам функции `SetClassLong`, которая модифицирует одноименные поля (`hIcon`, `hIconSm`) в структуре оконного класса `mainWnd.wc` главного окна приложения.

В тексте программы появилась также следующая директива:

```
#include "resource.h"
```

без которой имя `IDI_TRICOLOUR` было бы неизвестно компилятору.

После компиляции и запуска приложения на экран будет выведено окно программы с новой пиктограммой (рис. 5.10).

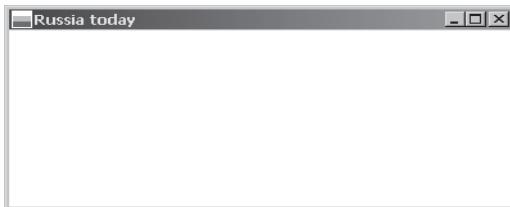


Рис. 5.10. Окно программы Russia (1-я редакция)

Курсоры

Курсыры – это изображения размером 32×32 пиксела, которые отмечают положение курсора (указателя) мыши. Курсоры во многом похожи на пиктограммы. Их главное отличие заключается в наличии *активной точки* (*hotspot*). Активной точкой называется пиксель, который принадлежит изображению курсора и отмечает его точное положение на экране в любой момент времени. В стандартном курсоре, имеющем вид стрелки, активная точка расположена в левом верхнем углу курсора.

Технологию использования в приложении курсоров, определенных программистом, продемонстрируем, продолжая разработку программы *Russia*.

В новой редакции программа будет изменять форму курсора мыши в зависимости от того, в какой зоне клиентской области он находится. В основном курсор мыши будет иметь форму перекрестья, которая задается предопределенным курсором `IDC_CROSS`. Но при приближении к верхней границе клиентской области курсор будет принимать форму стрелки, направленной вверх, а при приближении к нижней границе клиентской области – форму стрелки, направленной вниз. Кроме того, в окне программы будет отображаться информация о текущей позиции курсора, связанной с его активной точкой. Таким образом, к проекту нужно добавить два ресурса курсоров с идентификаторами `IDC_UP` и `IDC_DOWN`.

Процедура создания нового курсора в графическом редакторе очень похожа на рассмотренную выше процедуру создания пиктограммы. На первом шаге в диалоговом окне *Insert Resource* вместо типа ресурса *Icon* выбирается тип *Cursor*. Вызванный графический редактор будет работать в режиме создания курсора.

На этапе рисования, после того как будет создано изображение курсора, необходимо определить его активную точку. Например, при создании курсора `IDC_UP` окно графического редактора с созданным изображением показано на рис. 5.11.

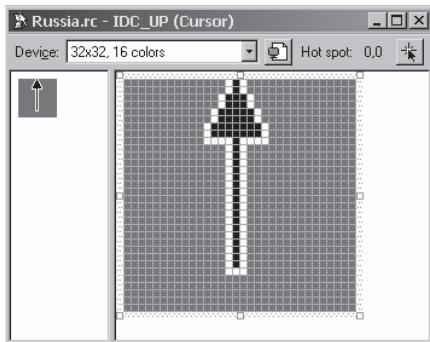


Рис. 5.11. Результат рисования для курсора IDC_UP

В отличие от окна графического редактора, работающего в режиме создания пиктограммы, в этом окне есть кнопка Set Hotspot, располагающаяся на правом краю панели инструментов. Координаты активной точки отображаются слева от этой кнопки и по умолчанию равны (0, 0). Позиция активной точки указывается всегда относительно левого верхнего угла изображения.

Чтобы назначить активную точку, нужно нажать кнопку Set Hotspot, а затем щелкнуть мышью на том пикселе изображения, который должен стать активной точкой. Для нашего курсора это будет пиксель на верхнем окончании стрелки. После указанных действий надпись слева от кнопки примет вид Hot spot: 15, 0.

Аналогичные действия повторим и для курсора IDC_DOWN. Но теперь активной точкой будет пиксель на нижнем конце стрелки (рис. 5.12).

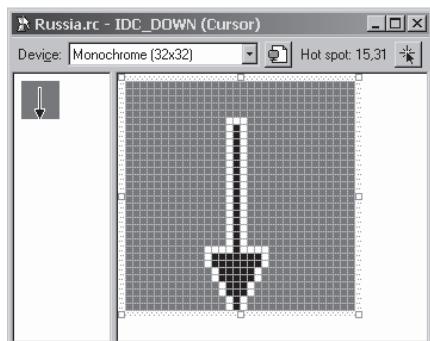


Рис. 5.12. Определение активной точки для курсора IDC_DOWN

Сохраните новые ресурсы в файле описания ресурсов, выполнив команду меню File ▶ Save. Если теперь открыть файл Russia.rc в текстовом режиме, то можно увидеть две новые строчки с определениями:

```
IDC_UP      CURSOR DISCARDABLE    "up.cur"  
IDC_DOWN    CURSOR DISCARDABLE    "down.cur"
```

Завершив определение ресурсов с курсорами, займемся кодом программы.

Необходимо определить дескрипторы используемых курсоров (типа HCURSOR) и загрузить ресурсы при помощи функции LoadCursor, вызов которой аналогичен вызову функции LoadIcon.

В нашей программе будут использоваться три курсора: `hCursor` для вывода перекрестья (предопределенный курсор `IDC_CROSS`), `hCursorUp` для вывода стрелки вверх (`IDC_UP`) и `hCursorDown` для вывода стрелки вниз (`IDC_DOWN`).

Курсор `hCursor` должен стать основным курсором приложения. Для этого нужно модифицировать соответствующее поле оконного класса, вызвав следующую функцию:

```
SetClassLong(hWnd, GCL_HCURSOR, (LONG)hCursor);
```

Для динамического изменения формы курсора в зависимости от его местонахождения применяется функция `SetCursor`:

```
HCURSOR SetCursor(HCURSOR hCursor);
```

Функция делает текущим курсор, который передается ей в качестве параметра. При этом возвращается дескриптор предшествующего курсора.

Отслеживание положения курсора обычно осуществляется при обработке сообщения `WM_MOUSEMOVE`. Параметр оконной процедуры `lParam` в этот момент содержит координаты текущей позиции курсора мыши `x` и `y`, которые размещаются в младшем и старшем словах параметра соответственно. Эта позиция определяется относительно левого верхнего угла клиентской области.

Вторая редакция файла `Russia.cpp` приведена в листинге 5.3.

Листинг 5.3. Проект Russia (2-я редакция)

```
///////////
// Russia.cpp
#include <windows.h>
#include <stdio.h>
#include "KWnd.h"
#include "resource.h"

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====================================================
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("Russia today", hInstance, nCmdShow, WndProc, NULL, 0, 0,
        400, 300);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}
//=====================================================
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HINSTANCE hInst;
    HICON hIcon, hIconSm;
    static HCURSOR hCursor, hCursorUp, hCursorDown;
    static int xPos, yPos; // координаты горячей точки курсора мыши
    static int wClient, hClient; // размеры клиентской области окна
    HDC hDC;
    PAINTSTRUCT ps;
    char text[100];
    RECT rect;
```

```
SetRect(&rect, 10, 0, 200, 30);

switch (uMsg)
{
    case WM_CREATE:
        hInst = GetModuleHandle(NULL);
        hIcon = LoadIcon(hInst, MAKEINTRESOURCE(IDI_TRICOLOUR));
        hIconSm = (HICON)LoadImage(hInst, MAKEINTRESOURCE(IDI_TRICOLOUR),
            IMAGE_ICON, 16, 16, LR_DEFAULTCOLOR);
        SetClassLong(hWnd, GCL_HICON, (LONG)hIcon);
        SetClassLong(hWnd, GCL_HICONSM, (LONG)hIconSm);

        hCursor = LoadCursor(NULL, IDC_CROSS);
        SetClassLong(hWnd, GCL_HCURSOR, (LONG)hCursor);
        hCursorUp = LoadCursor(hInst, MAKEINTRESOURCE(IDC_UP));
        hCursorDown = LoadCursor(hInst, MAKEINTRESOURCE(IDC_DOWN));
        break;

    case WM_SIZE:
        wClient = LOWORD(lParam);
        hClient = HIWORD(lParam);
        break;

    case WM_MOUSEMOVE:
        xPos = LOWORD(lParam);
        yPos = HIWORD(lParam);
        if (yPos < 16)
            SetCursor(hCursorUp);
        if (yPos > hClient - 16)
            SetCursor(hCursorDown);
        InvalidateRect(hWnd, &rect, TRUE);
        break;

    case WM_PAINT:
        hdc = BeginPaint(hWnd, &ps);
        sprintf(text, "xPos = %d, yPos = %d\n", xPos, yPos);
        DrawText(hdc, text, -1, &rect, DT_LEFT);
        EndPaint(hWnd, &ps);
        break;

    case WM_DESTROY:
        PostQuitMessage(0);
        break;

    default:
        return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
}

////////////////////////////////////////////////////////////////
```

Ширина **wClient** и высота **hClient** клиентской области окна определяются приложением при обработке сообщения **WM_SIZE**. Величина **hClient** используется затем в блоке обработки сообщения **WM_MOUSEMOVE**, чтобы определить область в нижней части окна, где курсор должен принять форму стрелки, указывающей вниз.

Вид окна работающей программы, когда курсор находится у верхнего края клиентской области, показан на рис. 5.13.

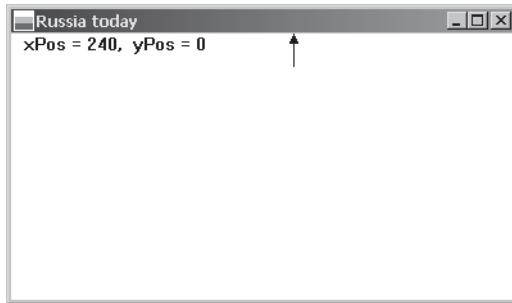


Рис. 5.13. Программа Russia с курсорами, определенными программистом

Еще один момент, на который хотелось бы обратить ваше внимание, — это вызов функции `InvalidateRect` в конце блока обработки сообщения `WM_MOUSEMOVE`:

```
InvalidateRect(hWnd, &rect, TRUE);
```

Этот вызов необходим, чтобы сделать *недействительной*, то есть требующей обновления, часть клиентской области. В результате указанного вызова система генерирует сообщение `WM_PAINT`. Обрабатывая это сообщение, приложение вызывает функцию `DrawText` для отображения текста.

Чтобы исключить наложение нового текста на текст, оставшийся от предыдущего акта рисования, нужно стереть предыдущее изображение, прежде чем выводить новое. Система Windows сделает это самостоятельно, выполняя вызов функции `BeginPaint`, но только при одном условии. «Виновница» генерации сообщения `WM_PAINT` — функция `InvalidateRect` — должна быть вызвана с третьим параметром, равным `TRUE`.

Также особое внимание следует обратить на второй аргумент в вызове функции `InvalidateRect`. Он указывает, что обновляемым регионом будет только небольшой прямоугольник, в котором размещается текст. Если бы использовалось значение `NULL`, то перерисовке подверглась бы вся клиентская область. Вы не заметите разницы, пока большая часть окна занята фоном и в окне не отображаются другие растровые изображения. Во всех других случаях перерисовка всей клиентской области вызывает неприятное мерцание.

Растровые образы

В главе 3 уже достаточно подробно рассматривалось использование в Windows-программировании растровых битовых образов или точечных рисунков. Там же отмечалось, что DDB-растры могут загружаться из ресурсов приложения. Фактически, пиктограммы и курсоры тоже являются особыми видами точечных рисунков.

Растровые образы чаще всего используются для решения двух классов задач. Прежде всего, с их помощью производится отображение на экране картинок. Например, файлы драйверов дисплеев в Windows содержат множество небольших битовых образов, которые используются для рисования стрелок в полосах прокрутки, галочек в раскрывающихся меню, изображений флагжков, переключателей и других служебных изображений.

Также растровые образы применяются для создания кистей. Кисти являются шаблонами пикселов, которые Windows использует для закрашивания изображаемых на экране областей. Пример такого использования битового образа вы можете найти в листинге 12.1 (глава 12).

Растровые образы создаются при помощи графического редактора аналогично созданию пиктограмм. На первом шаге в диалоговом окне *Insert Resource* вместо типа ресурса *Icon* выбирается тип *Bitmap*. Вызванный графический редактор будет работать в режиме создания растрового ресурса. В отличие от пиктограмм, растровые образы не ограничены в размерах.

Когда вы назначаете ресурсу идентификатор (на третьем шаге), окно *Bitmap Properties* (рис. 5.14) позволяет установить любые размеры для рисунка, а также выбрать палитру, содержащую 16 или 256 цветов.

Иногда бывает удобнее создать рисунок во внешнем графическом редакторе, обладающем более широкими возможностями, и добавить его в состав ресурсов приложения, выбрав на первом шаге режим *Импорт*. Ранее уже рассматривался сценарий импорта существующей пиктограммы. Импорт растрового образа осуществляется аналогично.

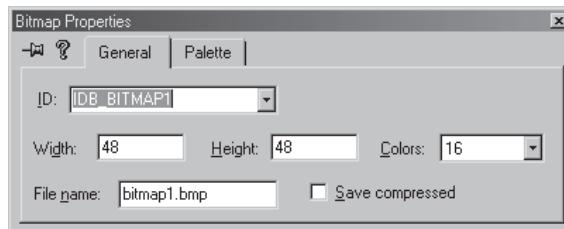


Рис. 5.14. Диалоговое окно Bitmap Properties

Для использования растра в программе необходимо определить дескриптор типа *NBITMAP* и присвоить ему значение, возвращаемое функцией *LoadBitmap*:

```
NBITMAP LoadBitmap(  
    HINSTANCE hInstance, // дескриптор экземпляра приложения  
    LPCTSTR lpBitmapName // имя растрового ресурса  
>);
```

Второй параметр функции интерпретируется так же, как для функции *LoadIcon*.

Растровые ресурсы хранятся в модулях Win32 в формате упакованного DIB-растра. Функция *LoadBitmap* находит растровый ресурс, загружает его в память, получает дескриптор упакованного DIB-растра, затем на его базе создает DDB-растра, совместимый с текущим экранным режимом, и возвращает дескриптор DDB-растра.

Вывод растрового образа в окно осуществляется при помощи функции *BitBlt* или *StretchBlt* с использованием совместимого контекста в памяти. Подробно эта технология рассматривалась в главе 3.

Продолжим разработку программы *Russia*. На этом этапе мы добавим к ней ресурс растрового образа.

Для подготовки изображения воспользуемся внешним графическим редактором MS Paint. Создайте с его помощью новый рисунок размером 400 × 280 пикселов, содержащий изображение российского государственного флага с надписью «Russia today» (рис. 5.15).

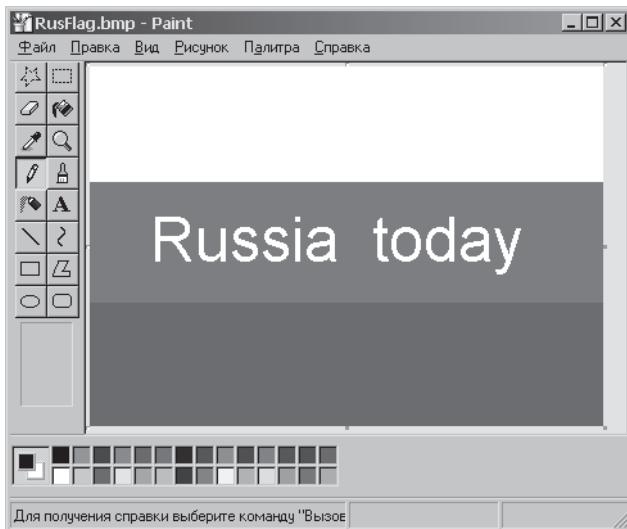


Рис. 5.15. Создание файла RussiaToday.bmp с помощью MS Paint

Этот рисунок нужно сохранить в файле с именем RusFlag.bmp, выбрав 256-цветный формат. Поместите файл с рисунком в папку проекта Russia¹.

Добавьте к нашему приложению ресурс растрового образа в режиме Импорт, связав его с файлом RusFlag.bmp и назначив идентификатор IDB_RUSFLAG.

Если после сохранения ресурса открыть файл Russia.rc в текстовом режиме, то можно найти строку с определением нового ресурса:

```
IDB_RUSFLAG      BITMAP      DISCARDABLE      "RusFlag.bmp"
```

Теперь займемся кодом программы. Для создания третьей редакции проекта нужно сделать несколько дополнительных вставок.

Добавьте в теле оконной процедуры определения переменных

```
static HBITMAP hBmpRusFlag; // дескриптор раstra
HDC hMemDC; // контекст устройства в памяти
static BITMAP bm; // параметры раstra
```

В блоке обработки сообщения WM_CREATE добавьте инструкции

```
hBmpRusFlag = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_RUSFLAG));
GetObject(hBmpRusFlag, sizeof(bm), (LPSTR)&bm);
```

В блоке обработки сообщения WM_PAINT сразу после вызова функции BeginPaint разместите следующий фрагмент кода:

```
hMemDC = CreateCompatibleDC(hDC);
SelectObject(hMemDC, hBmpRusFlag);
BitBlt(hDC, 0, 0, bm.bmWidth, bm.bmHeight, hMemDC, 0, 0, SRCCOPY);
DeleteDC(hMemDC);
```

После компиляции и запуска приложения его основное окно будет выглядеть, как показано на рис. 5.16.

¹ Вы можете взять готовый файл с рисунком, если загрузите исходный код данного проекта из файлов к книге, доступных на сайте издательства «Питер» (www.piter.com).

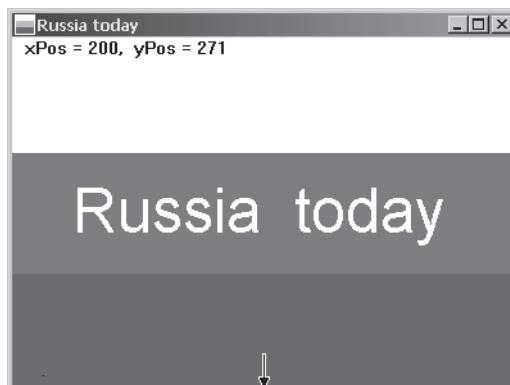


Рис. 5.16. Программа Russia. Курсор мыши находится у нижней границы окна

Перемещая курсор мыши в окне программы, понаблюдайте за изменением формы курсора и за качеством отображения картинки. Теперь проведите такой эксперимент. Замените второй аргумент (`&rect`) в вызове функции `InvalidateRect` значением `NULL` и перекомпилируйте программу. Посмотрите, как изменится ее поведение при перемещении курсора мыши.

Ресурсы, определяемые программистом

Ресурсы, определяемые программистом (*user-defined resource*), позволяют включать в выполняемый файл данные любого типа, после чего программа может легко получить к ним доступ во время своей работы. Эти ресурсы могут иметь любой формат: текстовый, двоичный или даже смешанный.

Предположим, что есть файл `proghelp.txt`, в котором содержится текст подсказок для пользователей приложения. Его можно включить в проект в качестве ресурса, если добавить в файл описания ресурсов следующую строку:

```
HELPTEXT TEXT DISCARDABLE proghelp.txt
```

Имя ресурса (`HELPTEXT`) и его тип (`TEXT`) в этом выражении могут быть любыми. Для имени ресурса стоит использовать прописные буквы, так как компилятор ресурсов в любом случае преобразует символы этого имени в символы верхнего регистра.

В процессе инициализации программы, например при обработке сообщения `WM_CREATE`, можно получить дескриптор этого ресурса, а через него — доступ к данным в файле `proghelp.txt`.

Другой пример: вы можете включить в программу в виде ресурса мультимедийный файл с записанным звуковым сопровождением. Пример такого использования ресурсов, определенных программистом, приводится в разделе «Воспроизведение звуковых файлов».

К сожалению, редактор ресурсов Visual Studio не очень удобен для добавления ресурсов, определяемых программистом. Поэтому проще будет открыть файл

описания ресурсов в текстовом формате и добавить строку описания вашего ресурса в следующем формате:

имя_ресурса тип_ресурса DISCARDABLE имя_файла

Рассмотрим технику доступа к ресурсам, которые определены программистом. Она немного сложнее, чем для ресурсов стандартных типов.

Доступ к данным в ресурсе

Для получения доступа к данным ресурса, определенного программистом, используются функции `FindResource`, `LoadResource` и `LockResource`.

Функция `FindResource` ищет ресурс указанного типа в соответствующем модуле. Она имеет следующий прототип:

```
HRSRC FindResource(HMODULE hModule, LPCTSTR lpName, LPCTSTR lpType);
```

Параметр `hModule` позволяет указывать дескриптор модуля выполняемого файла, который содержит указанный ресурс. Если `hModule` равен `NULL`, то Windows ищет ресурс в модуле, создавшем текущий процесс.

Параметр `lpName` содержит имя ресурса, представленное, как правило, строкой. Но если первым символом в строке является знак «#», то остальная часть представляет целое значение идентификатора ресурса. Параметр `lpType` задает тип ресурса.

При успешном выполнении функция возвращает дескриптор информационного блока указанного ресурса, а не дескриптор самого ресурса. Чтобы получить дескриптор ресурса, нужно передать дескриптор информационного блока функции `LoadResource`. В случае неудачи функция `FindResource` возвращает значение `NULL`.

Функция `LoadResource` загружает ресурс, найденный функцией `FindResource`, в глобальную область памяти. Прототип этой функции выглядит следующим образом:

```
HGLOBAL LoadResource(HMODULE hModule, HRSRC hResInfo);
```

Параметру `hResInfo` должен передаваться дескриптор информационного блока, возвращенный функцией `FindResource`.

Хотя адрес, возвращаемый функцией `LoadResource`, в принципе, можно использовать для доступа к данным ресурса, MSDN рекомендует получать указатель на эти данные, вызывая функцию `LockResource`:

```
LPVOID LockResource(HGLOBAL hResData);
```

Функция осуществляет блокировку указанного ресурса в памяти. Параметр `hResData` — это дескриптор ресурса, который должен быть заблокирован. В качестве параметра необходимо передать адрес, возвращенный функцией `LoadResource`.

В случае успешного выполнения функция `LockResource` возвращает указатель на первый байт данных ресурса, в случае неудачи — значение `NULL`.

После использования данных ресурса его разблокировка не требуется, так как система автоматически удаляет ресурс из памяти при закрытии приложения.

Если вернуться к примеру с файлом `proghelp.txt`, который был добавлен к приложению в виде определенного программистом ресурса, то доступ к данным этого ресурса можно получить при помощи следующего фрагмента кода:

```
HGLOBAL hResource = LoadResource(hInstance,  
    FindResource(hInstance, "HELPTEXT", "TEXT"));  
char* pHelpText = (char*) LockResource(hResource);
```

Другой пример использования ресурсов, определенных программистом, будет рассмотрен в следующем разделе.

Воспроизведение звуковых файлов

Добавление звука к приложению может сделать его более эффективным для пользователя, а в каких-то случаях и более привлекательным. Например, можно использовать специальные звуки или воспроизведение аудиоклипов, чтобы привлечь внимание пользователя при обнаружении ошибок или при завершении длительной операции. Другие очевидные сферы применения музыкальных и звуковых эффектов — это обучающие программы, а также приложения, предназначенные для презентаций или рекламы.

Win32 API поддерживает работу только со звуковыми файлами, записанными в формате **WAVE** (waveform audio file format). Файлы этого формата имеют расширение **.wav**. Они содержат оцифрованные с некоторой частотой 8- или 16-битовые звуковые данные. Хотя при записи **WAVE**-формата информация сжимается как аппаратными, так и программными средствами, она содержит все необходимые данные о реальном звуковом потоке. А потому файлы с расширением **.wav**, к сожалению, довольно громоздки и занимают сотни килобайтов на каждую минуту записи.

Для воспроизведения звуковых файлов вызывается функция **PlaySound**, которая имеет следующий прототип:

```
BOOL PlaySound(
    LPCSTR pszSound, // строка, специфицирующая воспроизводимый звук
    HMODULE hmod,    // дескриптор экземпляра приложения
    DWORD fdwSound   // флаги воспроизведения
);
```

Первый параметр функции задает либо имя файла на диске, либо строку имени звукового ресурса. Если при вызове **PlaySound** первый параметр равен **NULL**, это вызывает прекращение воспроизведения любого «играемого» файла.

Второй параметр используется только при воспроизведении звука из звукового ресурса. Если же функция вызвана для воспроизведения звука из файла, то второй параметр устанавливается в **NULL**.

Третий параметр управляет загрузкой и воспроизведением звука. В табл. 5.2 приведены наиболее часто используемые значения этого параметра.

Таблица 5.2. Значения параметра **fdwSound**

Флаг	Описание
SND_FILENAME	Параметр pszSound представляет собой имя файла
SND_RESOURCE	Параметр pszSound представляет собой идентификатор ресурса
SND_SYNC	Синхронное воспроизведение звука. Возврат из функции PlaySound происходит только по завершении воспроизведения звука
SND_ASYNC	Асинхронное воспроизведение звука. Возврат из функции PlaySound происходит немедленно после начала воспроизведения звука. Для прекращения асинхронного воспроизведения вызывается функция PlaySound , параметр pszSound которой равен NULL

продолжение ➔

Таблица 5.2. (продолжение)

Флаг	Описание
SND_LOOP	Звук воспроизводится циклически до тех пор, пока не будет вызвана функция PlaySound, параметр pszSound которой равен NULL. Вместе с этим флагом должен быть определен флаг SND_ASYNC
SND_NODEFAULT	Звук по умолчанию не используется. Если указанный звук не найден, PlaySound завершает работу, не воспроизводя при этом звук по умолчанию
SND_PURGE	Если параметр pszSound не равен NULL, то прекращается воспроизведение всех экземпляров указанного звука. Если параметр имеет значение NULL, то прекращается воспроизведение всех звуков вызывающего приложения
SND_NOWAIT	Если драйвер занят, то функция немедленно возвращает управление без воспроизведения звука

Функция PlaySound содержится в мультимедийной библиотеке Windows. Чтобы ее имя и расположение стали известны компоновщику Visual Studio, необходимо подключить к проекту мультимедийную библиотеку `winmm.lib`¹.

Таким образом, наиболее простой способ озвучить ваше приложение — это вызвать функцию PlaySound в режиме воспроизведения из файла. Конечно, при этом звуковой файл .wav должен находиться в одной папке с исполняемым exe-файлом программы. Это не очень удачное решение, если предполагается передавать программу сторонним пользователям. Не все специалисты обладают достаточной подготовкой для корректного общения с компьютером. Например, чрезмерно усердный пользователь может освобождать пространство на диске, и вдруг — вот сюрприз! — нет больше звукового файла... Именно поэтому как растровые, так и звуковые образы, используемые в программе, рекомендуется хранить в exe-файле приложения в виде ресурсов.

Функция PlaySound предоставляет такую возможность — режим воспроизведения звука из ресурса. Ее использование довольно просто и в этом случае, так как не требует предварительного вызова функций `FindResource`, `LoadResource` и `LockResource`. Все действия по загрузке и получению доступа к данным ресурса выполняются автоматически самой функцией PlaySound.

Использование функции PlaySound продемонстрируем на примере дальнейшей разработки программы *Russia*. Мы обещали добавить к программе воспроизведение государственного гимна России, и теперь выполним это обещание. В последней версии программы, названной *RussiaToday*, также будут удалены ресурсы курсоров, определенных программистом, и вывод на экран текущей позиции курсора мыши. Эта тема уже пройдена, поэтому код программы можно сделать более прозрачным.

В тексте файла *RussiaToday.rc* необходимо добавить строку описания звукового ресурса

```
MY_SOUND      WAVE      DISCARDABLE    "RusHymn.wav"
```

Необходимо также поместить в папку проекта звуковой файл *RusHymn.wav* с гимном России².

¹ Для Visual Studio 6.0 надо выполнить команду меню Project ▶ Settings и найти на вкладке Link текстовое поле Object/library modules.

² Гимн России можно скачать из Интернета и преобразовать его к формату WAVE либо воспользоваться файлом *RusHymn.wav* в составе данного проекта. Исходные тексты к примерам этой книги доступны на сайте издательства «Питер»: www.piter.com.

Исходный код основного файла программы приведен в листинге 5.4.

Листинг 5.4. Проект RussiaToday

```
////////////////////////////////////////////////////////////////
// RussiaToday.cpp
#include <windows.h>
#include <stdio.h>
#include "KWnd.h"
#include "resource.h"

char szSoundName[] = "MY_SOUND"; // имя звукового ресурса

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====================================================
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("Russia today", hInstance, nCmdShow, WndProc, NULL, 0, 0,
        400, 300);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}
//=====================================================
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HINSTANCE hInst;
    HICON hIcon, hIconSm;
    HDC hDC;
    PAINTSTRUCT ps;
    static HBITMAP hBmpRusFlag;
    HDC hMemDC;
    static BITMAP bm;

    switch (uMsg)
    {
    case WM_CREATE:
        hInst = GetModuleHandle(NULL);
        hIcon = LoadIcon(hInst, MAKEINTRESOURCE(IDI_TRICOLOUR));
        hIconSm = (HICON)LoadImage(hInst, MAKEINTRESOURCE(IDI_TRICOLOUR),
            IMAGE_ICON, 16, 16, LR_DEFAULTCOLOR);
        SetClassLong(hWnd, GCL_HICON, (LONG)hIcon);
        SetClassLong(hWnd, GCL_HICONSM, (LONG)hIconSm);
        hBmpRusFlag = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_RUSFLAG));
        GetObject(hBmpRusFlag, sizeof(bm), (LPSTR)&bm);

        // Воспроизведение звука из ресурса
        PlaySound (szSoundName, hInst, SND_RESOURCE | SND_ASYNC);
        break;

    case WM_PAINT:
        hDC = BeginPaint(hWnd, &ps);
        hMemDC = CreateCompatibleDC(hDC);
        SelectObject(hMemDC, hBmpRusFlag);
```

Листинг 5.4. (продолжение)

```

BitBlt(hDC, 0, 0, bm.bmWidth, bm.bmHeight, hMemDC, 0, 0, SRCCOPY);
DeleteDC(hMemDC);
EndPaint(hWnd, &ps);
break;

case WM_DESTROY:
PostQuitMessage(0);
break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
/////////////////////////////////////////////////////////////////

```

Откомпилировав и запустив программу *RussiaToday* на выполнение, вы увидите почти такое же окно, как на рис. 5.16, только без специализированных курсоров и без отладочного вывода текста, сообщающего о позиции курсора. Но теперь одновременно с появлением окна приложения на экране вы услышите торжественные звуки государственного гимна России!

Если у вас хорошая звуковая карта и качественные колонки, то, несомненно, вам захочется встать и... испытать чувство глубокого удовлетворения от пройденной темы.

А теперь — бонус! Хотя Win32 API поддерживает только WAVE-формат звуковых файлов, все-таки есть способ заставить зазвучать MP3-файлы! Для этого нужно использовать следующий вызов функции *ShellExecute*:

```
ShellExecute(hWnd, "open", "MySound.mp3", NULL, NULL, SW_SHOW);
```

Выполнение функции *ShellExecute* с указанными параметрами приводит к вызову программы, связанной по умолчанию с файлами с расширением .mp3. Скорее всего, это будет приложение WinAmp, если оно установлено на компьютере, либо приложение Windows Media Player. В любом случае вызванное приложение обеспечит воспроизведение файла MySound.mp3.

Таблицы строк

Ресурс *таблицы строк* — это список строк, связанных с уникальными численными идентификаторами. В файле описания ресурсов таблица строк имеет примерно следующий вид:

```

STRINGTABLE DISCARDABLE
BEGIN
    IDS_STRING1      "Открыть файл"
    IDS_STRING2      "Закрыть файл"
    IDS_STRING3      "Выход из программы"
    ...
END

```

Это так называемое *многострочное описание* ресурса. Многострочное описание используется также для ресурсов меню и диалоговых окон.

Необходимость или полезность использования ресурса таблицы строк для начинающих Windows-программистов не вполне очевидна. Казалось бы, использо-

вание обычных символьных строк, объявленных в виде переменных или констант в исходном тексте программы, не приносит никаких проблем. Зачем еще дополнительные строковые ресурсы? Тем не менее, в двух ситуациях использование ресурса таблицы строк дает ощутимые преимущества:

- при модификации программы с целью адаптации для иноязычного пользователя;
- при использовании строковых ресурсов для вывода текста в окнах подсказок (*tooltip*) или в строке состояния (*status bar*) приложения.

Ресурс таблицы строк значительно облегчает перевод программы на другие языки. Если вместо непосредственного использования символьных строк в исходном тексте программы вынести их в файл описания ресурсов, а затем извлекать строки по их идентификаторам, то вся текстовая информация окажется сосредоточенной в одном месте. Поэтому в случае разработки иноязычной версии сделанной ранее программы достаточно перевести текст в файле описания ресурсов и перекомпилировать программу.

Вторая сфера рекомендуемого использования ресурса таблицы строк связана с упрощением вывода подсказок в элементе управления *Tooltip* или в строке состояния приложения. Эти вопросы будут рассмотрены в главе 8.

Для добавления к приложению ресурса символьных строк при помощи редактора ресурсов выполните следующую последовательность действий:

1. В главном меню Visual Studio выполните команду меню **Insert ▶ Resource**. В появившемся диалоговом окне **Insert Resource** укажите тип ресурса **String Table** и нажмите кнопку **New**. В результате будет отображено окно редактора таблицы строк, показанное на рис. 5.17.

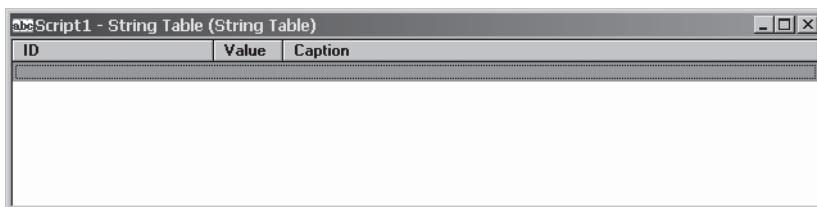


Рис. 5.17. Окно редактора таблицы строк

2. Сделайте двойной щелчок мышью на первой пустой строке в таблице. Появится диалоговое окно **String Properties**, показанное на рис. 5.18.

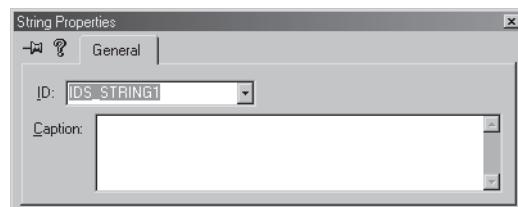


Рис. 5.18. Диалоговое окно для ввода строки

Введите нужный идентификатор в текстовом поле **ID** и символьную строку в окне **Caption**. Закройте диалоговое окно.

3. Повторите действия, описанные в п. 2, для каждой строки, которую необходимо внести в ресурс.
4. Сохраните ресурс с помощью команды меню File ▶ Save.

В составе ресурсов приложения может быть только одна таблица строк. Максимальный размер каждой строки составляет 255 символов. В строке не может быть управляющих символов языка С, за исключением символа табуляции \t. Однако символьные строки могут содержать восьмеричные константы \011 (табуляция), \012 (перевод строки) и \015 (возврат каретки). Эти управляющие символы распознаются функциями DrawText и MessageBox.

Вы можете использовать функцию LoadString для копирования строки из ресурса в символьный буфер szBuffer:

```
LoadString(hInstance, id, szBuffer, iMaxLength);
```

Параметр id соответствует идентификатору строки в файле описания ресурсов, а в параметре iMaxLength указывается максимальное число копируемых символов.

Давайте рассмотрим пример программы, в которой для вывода на экран трех сообщений о статусе некоторого файла используется ресурс таблицы строк.

Создайте проект с именем StringTable и добавьте к нему ресурс таблицы строк по описанной выше процедуре. Введите в таблицу три строки:

Идентификатор	Строка
IDS_FILENOTFOUND	Файл %s не найден
IDS_FILETOOBIG	Файл %s слишком велик для редактирования
IDS_FILE_READONLY	Файл %s доступен только для чтения

После сохранения этой информации в файле описания ресурсов StringTable.rc вы можете открыть его в текстовом режиме и увидеть следующее описание таблицы строк:

```
STRINGTABLE DISCARDABLE
BEGIN
    IDS_FILENOTFOUND      "Файл %s не найден."
    IDS_FILETOOBIG        "Файл %s слишком велик для редактирования."
    IDS_FILE_READONLY      "Файл %s доступен только для чтения."
END
```

Теперь добавьте к проекту файлы KWnd.h, KWnd.cpp и StringTable.cpp. Файл StringTable.cpp должен содержать текст, приведенный в листинге 5.5.

Листинг 5.5. Проект StringTable

```
///////////
// StringTable.cpp
#include <windows.h>
#include <stdio.h>
#include "KWnd.h"
#include "resource.h"

char szAppName[] = "StringTable";

int StatusFileMsg(HWND hwnd, int status, char* fileName);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
```

```
{  
    MSG msg;  
  
    KWnd mainWnd("StringTable", hInstance, nCmdShow, WndProc);  
  
    while (GetMessage(&msg, NULL, 0, 0)) {  
        TranslateMessage(&msg);  
        DispatchMessage(&msg);  
    }  
    return msg.wParam;  
}  
//================================================================  
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)  
{  
    switch (uMsg)  
    {  
        case WM_CREATE:  
            StatusFileMsg(hWnd, IDS_FILEREADONLY, "File_1");  
            StatusFileMsg(hWnd, IDS_FILETOOBIG, "File_2");  
            StatusFileMsg(hWnd, IDS_FILENOFOUND, "File_3");  
            break;  
  
        case WM_DESTROY:  
            PostQuitMessage(0);  
            break;  
  
        default:  
            return DefWindowProc(hWnd, uMsg, wParam, lParam);  
    }  
    return 0;  
}  
//================================================================  
int StatusFileMsg(HWND hwnd, int status, char* fileName)  
{  
    HINSTANCE hInst;  
    char szFormat[80];  
    char szBuffer[200];  
    hInst = GetModuleHandle(NULL);  
    LoadString(hInst, status, szFormat, 80);  
    sprintf(szBuffer, szFormat, fileName);  
  
    return MessageBox(hwnd, szBuffer, szAppName, MB_OK | MB_ICONEXCLAMATION);  
}  
//////////
```

В этом примере определена функция `StatusFileMsg`, отображающая информацию о статусе некоторого файла. Имя файла передается в параметре `fileName`, а его текущий статус — в параметре `status`. Содержание сообщения, соответствующего текущему статусу, извлекается из ресурса таблицы строк. В оконной процедуре функция `StatusFileMsg` вызывается три раза для демонстрации ее работы.

Теперь можно откомпилировать проект и посмотреть, как будет выполняться программа.

В главе 8 приводятся другие примеры программ, в которых используется ресурс таблицы строк. Такими примерами являются приложение `ToolBar` (листинг 8.1) и приложение `Statusbar` (листинг 8.5).

6

Меню и быстрые клавиши

Меню является важнейшим элементом большинства традиционных Windows-приложений. Практически в каждом приложении под строкой заголовка главного окна отображается полоса меню (menu bar), содержащего набор пунктов. Например, приложение Microsoft Visual Studio 6.0 имеет меню, состоящее из пунктов *File*, *Edit*, *View*, *Insert* и многих других, как показано на рис. 6.1.

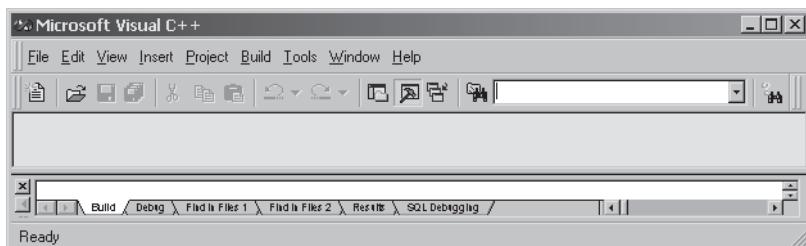


Рис. 6.1. Меню приложения Visual Studio 6.0

Меню большинства Windows-приложений выглядят сходным образом. Это облегчает освоение новых программ для пользователей. Поэтому при разработке новых приложений рекомендуется придерживаться сложившегося стиля оформления меню.

Организация и виды меню

Меню, располагающееся ниже заголовка окна приложения, называется *главным меню* (*main menu*), или *меню верхнего уровня*. Главное меню относится ко всему приложению.

Иногда в приложениях используются *контекстные меню*, появляющиеся под курсором при щелчке правой кнопкой мыши. Такие меню ассоциируются с некоторым объектом, например окном или пиктограммой, на который указывает курсор мыши в момент щелчка. Контекстные меню применяются для доступа к операциям, которые поддерживаются объектом.

Любое меню содержит *пункты меню*. Пункт меню обозначается на полосе меню своим *именем* — словом или короткой фразой.

Имя пункта может содержать выделенный символ, показанный подчеркиванием. Этот символ называют также *мнемоническим символом*. Он определяет *горячую клавишу* (*hotkey*) для выбора пункта при помощи клавиатуры.

Типы пунктов меню

Различают два типа пунктов меню:

- пункт-команда;
- пункт-подменю.

Пункт-команда — это терминальный (конечный) пункт на иерархическом дереве меню. Выбирая такой пункт, пользователь либо изменяет внутреннее состояние приложения, либо заставляет его выполнить некоторое действие. В программном коде пункту-команде сопоставлен уникальный целочисленный *идентификатор*.

Пункт-подменю — это заголовок вызываемого меню следующего, более низкого уровня.

Навигацию по меню пользователь осуществляет при помощи мыши или клавиатуры. Когда курсор мыши указывает на некоторый пункт главного меню, Windows выделяет его, изображая в виде объемной кнопки. Щелчок левой кнопкой мыши осуществляет *выбор* выделенного пункта меню.

Реакция Windows на выбор пункта меню зависит от типа пункта. Если выбрана команда, то Windows посыпает приложению сообщение *WM_COMMAND*, содержащее идентификатор этой команды. Если пользователь выбрал подменю, то Windows выводит на экран прямоугольную полосу этого подменю, называемого также *всплывающим меню* (*popup menu*). Для главного меню все подменю «всплывают» ниже выбранного пункта. На рис. 6.2 показано подменю *File* в окне приложения Visual Studio.

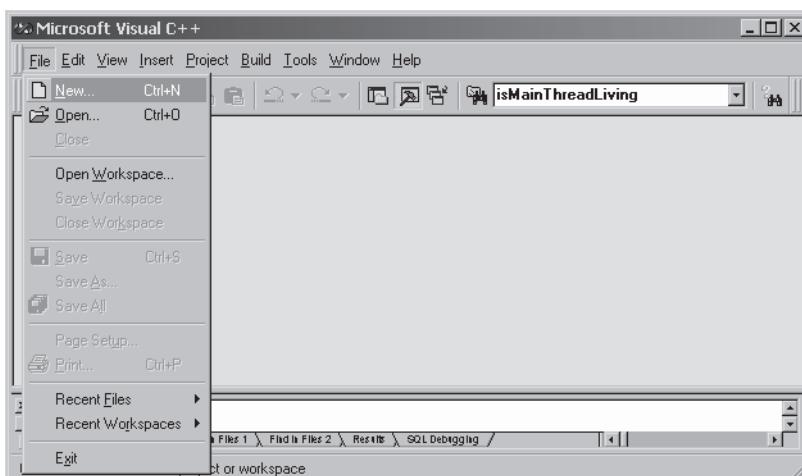


Рис. 6.2. Подменю File для приложения Visual Studio 6.0

В отличие от главного меню, имеющего горизонтальное размещение элементов, пункты подменю размещаются в виде вертикального столбца. При навигации по подменю пункты выделяются инверсией цвета (см. пункт *New*).

Восприятие подменю с большим количеством пунктов облегчается, если они разделены на некоторые логические группы. Разделители обозначаются горизонтальными линиями серого цвета.

Стоит обратить внимание на некоторые различия в изображении пунктов подменю *File*. Имена пунктов *New*, *Open* и некоторых других завершаются многоточием. Это многоточие обращает внимание пользователя на то, что при выборе данного пункта приложение выведет на экран диалоговое окно. Диалоговые окна значительно расширяют возможности графического интерфейса пользователя, но работа с ними будет рассматриваться только в следующей главе. Создавая меню, разработчик должен сам позаботиться о правильном именовании пунктов меню, в том числе и о многоточиях. Хотя это и не обязательно, но, как уже говорилось, следование сложившемуся стилю повышает удобство эксплуатации программы.

Пункты *Recent Files* и *Recent Workspaces* помечены символом стрелки. Это означает, что они открывают подменю следующего уровня. Если пользователь выделяет такой пункт, подведя курсор мыши, то Windows сразу выводит на экран соответствующее подменю, размещая его справа от выделенного пункта. Для этого уровня подменю используется также термин *каскадное меню*.

Справа от имен некоторых пунктов можно увидеть строки *Ctrl+N*, *Ctrl+O* и другие подобные клавиатурные сочетания. Это обозначения так называемых *быстрых клавиш*, работа с которыми рассматривается в конце данной главы.

Системное меню

Каждое окно, имеющее заголовок, может предоставлять доступ к *системному меню*, которое вызывается щелчком левой кнопки мыши на пиктограмме, расположенной в левой части заголовка окна. Системное меню открывается как всплывающее меню. В настоящее время этим меню редко пользуются, так как оно дублирует другие элементы управления окна.

Клавиатурный интерфейс меню

Пользователь может работать с меню без использования мыши, так как Windows предоставляет дублирующий клавиатурный интерфейс.

Чтобы начать работу с главным меню, нужно нажать клавишу *Alt* или *F10*, после чего меню активизируется и в нем автоматически выделяется первый пункт.

Выбор нужного пункта главного меню можно осуществить одним из двух способов:

1. С помощью клавиш со стрелками влево/вправо выделить требуемый пункт и нажать клавишу *Enter*.
2. Нажать «горячую» клавишу, соответствующую мнемоническому символу в имени пункта.

Выбор нужного пункта всплывающего меню осуществляется точно так же, но для навигации используются стрелки вниз/вверх.

При использовании «горячих» клавиш следует помнить, что текущий язык ввода с клавиатуры, установленный пользователем, должен соответствовать языку, на котором отображается имя пункта меню. Иначе выбор пункта не сработает.

Статус пунктов меню

Пункты меню могут быть *разрешенными* (*enabled*), *запрещенными* (*disabled*) и *недоступными* (*grayed*).

По умолчанию пункт меню является разрешенным. Когда пользователь выбирает такой пункт, система посыпает сообщение WM_COMMAND или отображает соответствующее подменю, в зависимости от типа пункта.

Запрещенный и недоступный пункты с точки зрения поведения одинаковы. Их можно выделить, но нельзя выбрать. То есть и при щелчке мышью, и при нажатии клавиши Enter ничего не происходит. Различаются запрещенный и недоступный пункты только своим внешним видом. Запрещенный пункт выглядит так же, как разрешенный, а недоступный пункт отображается серым цветом. Если вы хотите, чтобы пользователь знал, что пункт меню «отменен», делайте его *недоступным*.

В хорошо продуманном интерфейсе пользователя приложение должно управлять статусом пунктов меню в зависимости от текущего состояния программы. Например, команда вставки данных из буфера обмена Windows не имеет смысла, если буфер обмена пуст. В такой ситуации лучше отменить и выделить серым цветом соответствующий пункт меню. Вообще, рекомендуется делать недоступными те пункты меню, использование которых в данный момент бессмысленно или даже небезопасно с точки зрения устойчивости работы приложения.

Каков же смысл использования запрещенных пунктов, которые внешне не отличаются от разрешенных пунктов? MSDN указывает, что их можно использовать в обучающих приложениях, когда некоторое окно выводится на экран только для иллюстрации и не должно реагировать на действия пользователя.

Для изменения статуса пунктов меню применяется функция EnableMenuItem.

Отметка пунктов меню

Иногда пункт меню используется в роли *флажка* (*check box*). Флажок может быть установлен или сброшен. Переход из одного состояния в другое происходит при каждом выборе пункта меню. Установленный флажок помечается символом «галочка», а сброшенный флажок теряет отметку. Флажки обычно объединяются в группы и обеспечивают выбор либо одной, либо нескольких опций одновременно.

Пункты меню могут использоваться также в роли *переключателей* (*radio button*). Переключатели, как и флажки, обычно используются в группе. В отличие от флажков, переключатели связываются только с взаимоисключающими опциями, поэтому в группе можно выбрать только один переключатель. Выбранный переключатель отмечается жирной точкой или маленьким закрашенным кружком.

На рис. 6.3 показано окно приложения MenuDemo1, в котором вызвано каскадное меню, содержащее две группы пунктов. Первая группа пунктов управляет выбором RGB-составляющих цвета. В ней может быть выбрана любая комбинация этих флажков. Вторая группа предназначена для выбора интенсивности цвета из трех вариантов: темный, средний, светлый. Эти пункты интерпретируются как переключатели.

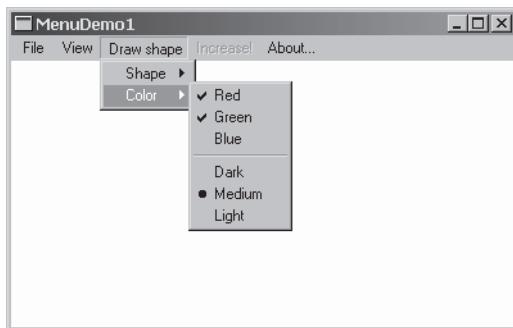


Рис. 6.3. Отметка пунктов-флажков и пунктов-переключателей в подменю Color

Для управления отметкой пунктов-флажков применяется функция `CheckMenuItem`. Для управления отметкой пунктов-переключателей — функция `CheckMenuItem`.

Пункт меню, применяемый по умолчанию

Подменю может содержать один пункт, выполняемый по умолчанию. Имя этого пункта выделяется жирным шрифтом. Когда пользователь открывает подменю двойным щелчком мыши, Windows автоматически выполнит команду по умолчанию, закрыв при этом подменю. Если пункт по умолчанию отсутствует, то двойной щелчок приводит просто к открытию подменю.

Атрибут «применяемый по умолчанию» может быть назначен любому пункту подменю при помощи функции `SetMenuItemDefault`.

Определение меню в виде ресурса

Вы можете создать меню одним из трех способов:

- ❑ на основе шаблона меню, определенного в файле описания ресурсов;
- ❑ при помощи функций `CreateMenu` и `AppendMenu`;
- ❑ на основе шаблона меню, определяемого в памяти во время выполнения программы при помощи функции `LoadMenuIndirect`.

Чаще всего используется первый способ. Именно он и будет рассматриваться в этой главе.

Шаблон меню

Определение меню в файле описания ресурсов имеет следующий вид:

```
имя_меню MENU DISCARDABLE
BEGIN
    Описание 0-го пункта
    Описание 1-го пункта
    ...
    Описание (n-1)-го пункта
END
```

Имя меню интерпретируется Windows либо как С-строка с завершающим нулевым символом, либо как целочисленный идентификатор. Точно такой же механизм использовался для обозначения имени пиктограммы в файле описания ресурсов (см. главу 5).

Синтаксис описания *i*-го пункта меню ($i = 0, 1, \dots, n - 1$) зависит от типа пункта.

Описание *пункта-подменю* имеет следующий вид:

```
POPUP имя_пункта [, параметры]
BEGIN
    Описание 0-го пункта
    Описание 1-го пункта
    ...
    Описание (n-1)-го пункта
END
```

Легко заметить, что за исключением заголовка (первая строка) синтаксис и семантика описаний меню и подменю совпадают.

Описание *пункта-команды* имеет следующий вид:

```
MENUITEM имя_пункта идентификатор [, параметры]
```

Если в имени пункта встречается символ &, то следующий за ним символ является мнемоническим символом.

Если вместо имени пункта использовано слово SEPARATOR, это приведет к тому, что вместо пункта меню будет отображена горизонтальная разделительная линия между группами пунктов.

Более подробно рассматривать определение шаблона меню нет нужды, потому что оно создается автоматически при использовании редактора меню.

Вызов редактора меню

В главном меню Visual Studio выполните команду Insert ▶ Resource. В появившемся диалоговом окне Insert Resource укажите тип ресурса Menu и нажмите кнопку New. В результате будет открыто окно редактора меню с заготовкой полосы нового меню (рис. 6.4).



Рис. 6.4. Окно редактора меню с заготовкой полосы меню

По умолчанию редактор присваивает первому из создаваемых шаблонов меню имя IDR_MENU1. В файле resource.h этот идентификатор определяется как целочисленная константа. Если вы хотите изменить назначенное имя, то сделайте двойной щелчок мышью на полосе меню и в открывшемся диалоговом окне Menu Properties введите необходимый идентификатор в текстовом поле ID. Например, для шаблона главного меню можно оставить имя по умолчанию IDR_MENU1, а для шаблонов контекстных меню назначить имена, отражающие их назначение.

В начале полосы создаваемого меню будет отображен пунктирный прямоугольник нулевого пункта меню. Нумерация пунктов начинается с нуля.

Атрибуты пункта меню

Если сделать двойной щелчок мышью на прямоугольнике пункта меню, то будет отображено диалоговое окно **Menu Item Properties** (рис. 6.5), предназначенное для ввода или изменения атрибутов этого пункта.

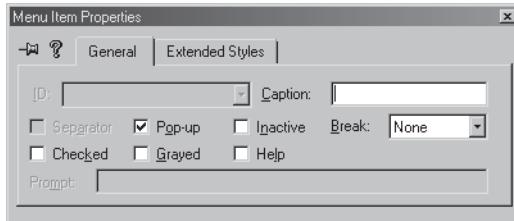


Рис. 6.5. Диалоговое окно **Menu Item Properties**

Предназначение наиболее часто используемых атрибутов показано в табл. 6.1.

Таблица 6.1. Атрибуты пункта меню

Атрибут	Описание
ID	Идентификатор пункта
Caption	Имя пункта (если в имени встречается символ «&», то следующий за ним символ является мнемоническим)
Separator	Пункт представляет собой горизонтальную разделительную линию
Checked	При выводе на экран пункт помечается слева галочкой
Pop-up	Пункт определяет подменю, если флагок отмечен. В противном случае пункт является обычной командой
Grayed	Пункт недоступен в исходном состоянии (несовместим с атрибутом Inactive)
Inactive	Пункт запрещен в исходном состоянии (несовместим с атрибутом Grayed)
Break	Этот атрибут может принимать одно из трех значений: None — обычный пункт меню; Column — для меню верхнего уровня пункт выводится с новой строки, а для подменю — в новом столбце; Bar — дополнительный столбец подменю отделяется вертикальной линией

Как показано на рис. 6.5, изначально окно атрибутов пункта настроено на определение *пункта-подменю* (флагок *Pop-up* установлен).

Если вам нужно определить данный пункт как *пункт-команду*, то флагок *Pop-up* необходимо сбросить. В этом случае текстовое поле ID становится доступным для ввода идентификатора (рис. 6.6).

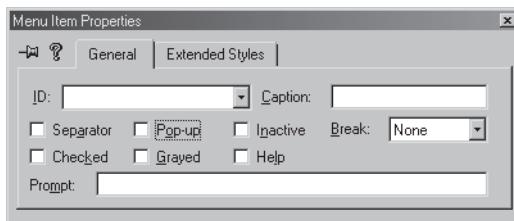


Рис. 6.6. Диалоговое окно **Menu Item Properties** со сброшенным флагком *Pop-up*

Обычно идентификаторы меню начинаются с префикса IDM_ (ID for a menu). Например, определяя пункт-команду для открытия файла, вы вводите в поле Caption текстовую строку &Open..., а в поле ID — текстовую строку IDM_OPEN.

ПРИМЕЧАНИЕ

Иногда мы будем отступать от этого правила и назначать для отдельных пунктов-команд идентификаторы, начинающиеся с префикса ID_. Это удобно, если демонстрируется решение одной и той же задачи с применением разных средств. Например, в рассматриваемой ниже программе MenuDemo1 выбор вида рисуемой фигуры осуществляется при помощи команд меню ID_RECTANGLE, ID_RHOMB, ID_ELLIPSE. В главе 7 решение аналогичной задачи в приложении DlgDemo2 реализуется через вызов диалогового окна с элементами управления типа «переключатель», которые имеют такие же идентификаторы. Применяя для сходных опций одни и те же идентификаторы в разных проектах, мы облегчаем сравнительный анализ этих решений.

Уровни меню

Меню верхнего уровня обычно относится к *нулевому уровню*. Если пункт этого меню определяет некоторое подменю, то такое подменю будет называться *меню первого уровня*. Если пункт меню первого уровня, в свою очередь, определяет некоторое подменю, то такое подменю будет называться *меню второго уровня*.

Прежде чем излагать процедуру создания меню при помощи редактора меню, опишем *рекурсивную процедуру* определения произвольного пункта для меню i -го уровня.

Процедура определения пункта для меню i -го уровня

1. Начало.
2. Двойным щелчком мыши на прямоугольнике пункта меню вызовите диалоговое окно *Menu Item Properties*.
3. Если пункт определяет подменю, то есть меню $(i + 1)$ -го уровня, выполните следующую последовательность действий:
 - установите флажок *Pop-up*;
 - введите имя пункта. Если оно содержит мнемонический символ, то перед таким символом нужно вставить знак &;
 - при необходимости задайте другие опции пункта;
 - закройте окно *Menu Item Properties*;
 - для каждого пункта меню $(i + 1)$ -го уровня выполните настоящую процедуру, подставив на месте параметра i значение $i + 1$.
4. Если пункт определяет команду, то выполните следующую последовательность действий:
 - сбросьте флажок *Pop-up*;
 - введите имя пункта, предваряя мнемонический символ знаком &;
 - введите идентификатор пункта;

- при необходимости задайте другие опции пункта;
 - закройте окно **Menu Item Properties**.
5. Если пункт определяет горизонтальную разделительную линию, то выполните следующую последовательность действий:
- сбросьте флажок **Pop-up**;
 - установите флажок **Separator**;
 - закройте окно **Menu Item Properties**.
6. Конец.

Обратите внимание на то, что на шаге 3 алгоритма процедура вызывает саму себя. Именно поэтому она является рекурсивной.

При кажущейся простоте алгоритма на самом деле он может описывать довольно сложный процесс. Вообразите себе виртуальный стек, на вершине которого запоминается очередной рекурсивный вызов процедуры из шага 3 алгоритма. Когда разработчик, создающий меню, достигает шага 6 алгоритма, это в общем случае вовсе не конец процесса. Если виртуальный стек не пуст, то нужно вернуться к прерванной процедуре, из которой произошел вызов выполняемой процедуры. И, конечно, удалить запомненный вызов с вершины стека. И только когда воображаемый стек окажется пуст, создание рассматриваемого пункта меню *i*-го уровня будет завершено.

Теперь можно описать процедуру создания меню верхнего уровня при помощи редактора меню.

Процедура определения меню нулевого уровня

1. Для каждого пункта меню выполните описанную выше процедуру определения пункта для меню *i*-го уровня, полагая значение *i* равным нулю.
2. Если возникнет необходимость перегруппировать пункты меню в пределах конкретного уровня, то перетащите пункты в нужную позицию при помощи мыши. Сохраните ресурс меню аналогично сохранению ресурса пиктограммы¹.

Добавление меню к окну приложения

После определения меню в файле описания ресурсов оно еще не появится в составе окна приложения. Чтобы это случилось, меню нужно присоединить к окну. Для этого можно использовать несколько способов.

Наиболее традиционным способом является присваивание полю `lpszMenuName` структуры `WNDCLASSEX` значения указателя на имя меню, что надо сделать еще перед регистрацией класса окна. Если имя меню определено как целочисленный идентификатор (например, `IDR_MENU1`), то применяйте макрос `MAKEINTRESOURCE` для получения значения, присваиваемого полю `lpszMenuName`.

¹ См. главу 5, раздел «Создание пиктограммы с помощью графического редактора».

Так как во всех примерах программного кода в данной книге главное окно приложения создается с использованием объекта класса `KWnd`, то рассматриваемый способ присоединения меню реализуется передачей значения `MAKEINTRESOURCE(IDR_MENU1)` параметру `menuName` конструктора класса `KWnd` (чуть позже вы увидите, как это делается, в листинге 6.1).

Напомним, что оконный класс¹, регистрируемый с использованием структуры `WC` типа `WNDCLASSEX`, определяет меню, используемое по умолчанию всеми окнами этого класса. А что значит «используемое по умолчанию»? Только то, что при создании окна с помощью функции `CreateWindow` параметру `hMenu` этой функции передается значение `NULL`. Это видно в коде конструктора класса `KWnd`.

Вы можете связать с окном при его создании другое меню, отличающееся от того, которое применяется по умолчанию. В этом способе требуемое меню сначала загружается при помощи функции `LoadMenu`, имеющей следующий прототип:

```
HMENU LoadMenu(HINSTANCE hInstance, LPCTSTR lpメニュー名);
```

Затем дескриптор меню `hMenu`, возвращаемый функцией `LoadMenu`, передается параметру `hMenu` функции `CreateWindow`.

Есть и еще один способ назначения меню, когда для этой цели используется функция `SetMenu`:

```
BOOL SetMenu(HWND hWnd, HMENU hMenu);
```

В качестве параметров данная функция принимает дескриптор окна и дескриптор меню, возвращенный функцией `LoadMenu`. Новое меню заменяет старое меню, если оно уже было.

Внесение изменений в меню

Обычно в сложных приложениях возникает необходимость изменять меню в ходе выполнения программы. Win32 API предоставляет соответствующие функции для модификации меню и его отдельных пунктов.

Состав меню можно изменять с помощью следующих функций:

- `AppendMenu` — добавляет новый элемент в конец меню;
- `DeleteMenu` — удаляет существующий пункт меню и уничтожает его;
- `InsertMenu` — вставляет в меню новый пункт;
- `RemoveMenu` — удаляет существующий пункт меню.

Необходимо хорошо понимать, как различаются функции `DeleteMenu` и `RemoveMenu`, когда они применяются к пунктам-подменю. Функция `DeleteMenu` уничтожает это подменю, а функция `RemoveMenu` — нет.

Для пунктов меню наиболее часто применяются такие изменения, как смена символьной строки имени пункта, установка или снятие отметки, изменение статуса пункта. Функция `SetMenuItemInfo` позволяет выполнять сразу несколько таких операций в одном вызове функции. Можно также использовать более конк-

¹Не забываете, что *оконный класс* — это термин Windows, не имеющий никакого отношения к классам C++.

ретные функции для выполнения отдельных операций, которые приведены в следующем списке:

- ❑ CheckMenuItem — управляет отметкой пунктов-флажков;
- ❑ CheckMenuRadioItem — управляет отметкой пунктов-переключателей;
- ❑ EnableMenuItem — изменяет статус пункта меню;
- ❑ ModifyMenu — изменяет имя пункта меню.

Некоторые из функций стоит рассмотреть более подробно.

Функция CheckMenuItem

Функция ставит или снимает отметку на пункте меню, который интерпретируется как флагок:

```
DWORD CheckMenuItem (
    HMENU hmenu,           // дескриптор меню
    UINT uIDCheckItem,     // идентификатор или позиция пункта меню
    UINT uCheck            // интерпретация второго параметра и выполняемое действие
);
```

Пункт меню, для которого применяется функция, задается вторым параметром. Этому параметру передается либо идентификатор пункта меню в файле описания ресурсов, либо позиция пункта меню. Выбор варианта интерпретации задается в третьем параметре.

Третий параметр, *uCheck*, задается как побитовая операция объединения двух флагов. Первый флаг может содержать одно из значений: **MF_BYCOMMAND** или **MF_BYPOSITION**. Второй флаг может принимать одно из значений: **MF_CHECKED** или **MF_UNCHECKED**. Интерпретация указанных значений приведена в табл. 6.2.

Таблица 6.2. Значения флагов для параметра *uCheck*

Значение	Описание
MF_BYCOMMAND	Значение <i>uIDCheckItem</i> содержит идентификатор пункта меню
MF_BYPOSITION	Значение <i>uIDCheckItem</i> содержит относительную позицию пункта меню с отсчетом от нуля
MF_CHECKED	Поместить отметку слева от имени пункта меню
MF_UNCHECKED	Снять отметку слева от имени пункта меню

Функцию **CheckMenuItem** так же, как и функцию **CheckMenuRadioItem**, можно применять только для пунктов подменю. Пункты главного меню не могут быть помечены.

ВНИМАНИЕ

Используйте всегда флаг **MF_BYCOMMAND**, передавая параметру *uIDCheckItem* идентификатор пункта меню. Это предотвратит возможные проблемы в процессе сопровождения программы, если потребуется модификация меню, изменяющая относительные позиции пунктов меню.

Данная рекомендация относится и к последующим функциям, интерфейс которых предоставляет два варианта передачи ссылок на пункты меню.

Функция CheckMenuRadioItem

Эта функция ставит отметку на выбранном пункте-переключателе и снимает отметку со всех остальных пунктов-переключателей в указанной группе пунктов:

```
BOOL CheckMenuItem (
    HMENU hmenu, // дескриптор меню
    UINT idFirst, // идентификатор или позиция первого пункта в группе
    UINT idLast, // идентификатор или позиция последнего пункта в группе
    UINT idCheck, // идентификатор или позиция выбранного пункта
    UINT uFlags // интерпретация параметров idFirst, idLast и idCheck
);
```

Если параметр `uFlags` имеет значение `MF_BYCOMMAND`, то параметры со второго по четвертый указывают идентификаторы пунктов меню. Если параметр `uFlags` равен `MF_BYPOSITION`, то эти параметры указывают позиции пунктов меню.

ВНИМАНИЕ

Задавая в редакторе пункты меню для группы переключателей, следует убедиться, что их идентификаторы в файле `resource.h` имеют сквозную нумерацию и упорядочены в соответствии с позициями этих пунктов на полосе меню. При нарушении этого условия функция `CheckMenuItem` может работать некорректно.

Функция EnableMenuItem

Функция позволяет изменять статус пункта меню. Она имеет следующий прототип:

```
BOOL EnableMenuItem (
    HMENU hMenu, // дескриптор меню
    UINT uIDEnableItem, // идентификатор или позиция пункта
    UINT uEnable // интерпретация второго параметра и выполняемое действие
);
```

Как и при работе с функцией `CheckMenuItem`, параметр `uEnable` должен содержать флаг `MF_BYCOMMAND` или `MF_BYPOSITION`, а также один из трех флагов: `MF_ENABLED` (пункт разрешен), `MF_DISABLED` (пункт запрещен) или `MF_GRAYED` (пункт недоступен).

Если в результате применения функции изменяется статус пункта *главного меню*, то следует обязательно вызвать функцию `DrawMenuBar` для повторного отображения изменившейся полосы меню.

Функция ModifyMenu

Функция изменяет существующий пункт меню:

```
BOOL ModifyMenu (
    HMENU hMnu, // дескриптор меню
    UINT uPosition, // идентификатор или позиция пункта
    UINT uFlags, // флаги
    UINT_PTR uIDNewItem, // новый идентификатор пункта
    LPCTSTR lpNewItem // новое содержание пункта
);
```

Параметр `uFlags` должен содержать один из флагов: `MF_BYCOMMAND` или `MF_BYPOSITION`, — значение которых рассматривалось ранее. Кроме того, параметр `uFlags` может содержать другие флаги, в частности, влияющие на интерпретацию параметра `lpNewItem`. Значения некоторых дополнительных флагов приведены в табл. 6.3.

В этой книге мы рассматриваем меню только с традиционным текстовым представлением его пунктов. Применяя функцию `ModifyMenu` для меню с текстовым представлением пунктов, используйте флаг по умолчанию `MF_STRING`.

Таблица 6.3. Флаги, влияющие на интерпретацию параметра lpNewItem

Флаг	Значение параметра lpNewItem
MF_STRING	Содержит указатель на С-строку. Этот флаг используется по умолчанию
MF_BITMAP	Параметр содержит дескриптор раstra. В этом случае пункт меню будет отображен не в виде текстовой строки, а в виде рисунка
MF_OWNERDRAW	Содержит данные, используемые для рисования пункта меню способом OwnerDraw

Отметим, что во всех рассмотренных функциях в качестве первого параметра указывается дескриптор меню. К счастью, Win32 API предоставляет функции, позволяющие определять значения дескрипторов как для главного меню, так и для любого подменю.

Функции для получения дескриптора меню

Дескриптор меню верхнего уровня, связанного с окном hWnd, можно получить при помощи функции GetMenu:

```
HMENU GetMenu(HWND hWnd);
```

Функция возвращает значение NULL, если окно hWnd не имеет меню. Эту функцию нельзя применять для дочерних окон.

Дескриптор подменю определяется вызовом функции GetSubMenu:

```
HMENU GetSubMenu (
    HMENU hMenu, // дескриптор родительского меню
    int nPos     // позиция пункта-подменю в родительском меню
);
```

Относительная позиция nPos для пункта родительского меню отсчитывается от нуля. Если пункт с позицией nPos не активизирует всплывающее меню, а является пунктом-командой, то функция возвращает значение NULL.

Сообщения меню

Система Windows посылает сообщение WM_COMMAND при каждом выборе пункта меню, определяющего команду. Обычно это единственное сообщение, обрабатываемое приложением, которое может поступить от меню. При выборе пунктов системного меню вместо указанного сообщения отправляется сообщение WM_SYSCOMMAND.

Иногда в программе может потребоваться обработка сообщений WM_INITMENU и WM_INITMENUPOPUP. Они отправляются непосредственно перед активизацией главного меню или всплывающего меню. Эти сообщения позволяют приложению изменить меню перед тем, как оно будет отображено на экране.

Сообщение WM_MENUCHAR отправляется, если пользователь пытается использовать клавиатурную «горячую» клавишу, которая не соответствует ни одному из мнемонических символов меню. Это позволяет обрабатывать несколько «горячих» клавиш для одного пункта меню или отображать сообщение об ошибке.

При навигации по меню система отправляет также сообщение WM_MOUSESELECT. Оно более универсально по сравнению с WM_COMMAND, так как инициируется даже тогда, когда выделен недоступный или запрещенный пункт. Это сообщение мо-

жет использоваться для формирования контекстной справки меню, которая отображается в строке состояния приложения.

В большинстве программ все сообщения от меню, кроме WM_COMMAND, передаются на обработку в функцию DefWindowProc.

Окно получает и обрабатывает сообщение WM_COMMAND при помощи своей оконной процедуры:

```
HRESULT CALLBACK WndProc (
    HWND hWnd,           // дескриптор окна
    UINT uMsg,           // WM_COMMAND
    WPARAM wParam,       // идентификатор пункта меню
    LPARAM lParam // 0
);
```

На самом деле источником сообщений WM_COMMAND могут быть не только пункты меню, но и быстрые клавиши, а также другие элементы управления (кнопки, списки, текстовые поля и т. д.). Сообщения от меню или быстрых клавиш всегда имеют нулевое значение параметра lParam. Между собой эти два вида сообщений различаются значением старшего слова wParam (табл. 6.4).

Таблица 6.4. Интерпретация параметров wParam и lParam для сообщений от разных источников

Источник	Младшее слово wParam	Старшее слово wParam	lParam
Меню	Идентификатор пункта меню	0	0
Быстрая клавиша	Идентификатор быстрой клавиши	1	0
Элемент управления	Идентификатор элемента управления	Код уведомления (нотификационного сообщения)	Дескриптор элемента управления (HWND)

Обработка сообщения WM_COMMAND в оконной процедуре осуществляется в блоке оператора switch. Например, в программе MenuDemo1, рассматриваемой ниже, для обработки команд подменю File используется следующий фрагмент кода:

```
case WM_COMMAND:
    switch (LOWORD(wParam))
    {
        case IDM_OPEN:
            MessageBox(hWnd, "Выбран пункт 'Open'", "Меню File", MB_OK);
            break;
        case IDM_CLOSE:
            MessageBox(hWnd, "Выбран пункт 'Close'", "Меню File", MB_OK);
            break;
        case IDM_SAVE:
            MessageBox(hWnd, "Выбран пункт 'Save'", "Меню File", MB_OK);
            break;
        case IDM_EXIT:
            SendMessage(hWnd, WM_DESTROY, 0, 0);
            break;
    }
    // ...
```

Поскольку этот пример является учебным, то вся обработка для команд IDM_OPEN, IDM_CLOSE и IDM_SAVE сводится к выводу окна сообщений при помощи функции MessageBox. Но обработка команды IDM_EXIT содержит вполне реальный код, выполнение которого вызывает завершение работы приложения.

Приложение MenuDemo1

Для демонстрации техники использования меню приведем пример разработки специального учебного приложения MenuDemo1.

Окно программы MenuDemo1 показано на рис. 6.3. Эта программа должна рисовать в центре окна прямоугольник, ромб или эллипс по выбору пользователя. Фигура закрашивается однородной кистью. Цвет кисти задается пользователем как комбинация цветовых составляющих Red, Green, Blue с учетом интенсивности, выбираемой из трех вариантов: Dark (темный цвет), Medium (средний цвет), Light (светлый цвет). Также по команде пользователя программа должна увеличивать размеры фигур до полного заполнения клиентской области окна, а также возвращать их к исходному размеру. Программа должна позволить пользователю скрыть изображение, а потом снова показать его. Также в приложении должно функционировать подменю File, пункты которого вызывают окна сообщений с информацией о выбранном пункте.

Приступим к разработке.

Создайте новый проект с именем MenuDemo1. Скопируйте в папку проекта файлы KWnd.h и KWnd.cpp из листинга 1.2 и добавьте их в состав проекта. Определите ресурс меню по описанной выше процедуре с сохранением в файле MenuDemo1.rc.

Главное меню должно содержать пункты, которые приведены в табл. 6.5.

Таблица 6.5. Пункты главного меню (0-й уровень)

Имя пункта	Тип пункта	Идентификатор	Позиция
&File	Подменю	—	0
&View	Подменю	—	1
&Draw shape	Подменю	—	2
&Increase!	Команда	IDM_RESIZE	3
&About...	Команда	IDM_ABOUT	4

Имя пункта Increase завершается восклицательным знаком. Это подсказка для пользователя, информирующая, что данный пункт главного меню является командой, а не заголовком подменю.

Подменю File должно содержать пункты, перечисленные в табл. 6.6.

Таблица 6.6. Пункты подменю File (1-й уровень)

Имя пункта	Тип пункта	Идентификатор
&Open...	Команда	IDM_OPEN
&Close...	Команда	IDM_CLOSE
&Save...	Команда	IDM_SAVE
—	SEPARATOR	—
E&xit	Команда	IDM_EXIT

Подменю View должно содержать пункты, перечисленные в табл. 6.7.

Таблица 6.7. Пункты подменю View (1-й уровень)

Имя пункта	Тип пункта	Идентификатор
&Show shape	Команда	IDM_SHOW_SHAPE
&Hide shape	Команда	IDM_HIDE_SHAPE

Подменю Draw shape должно содержать пункты, перечисленные в табл. 6.8.

Таблица 6.8. Пункты подменю Draw shape (1-й уровень)

Имя пункта	Тип пункта	Идентификатор
&Shape	Подменю	—
&Color	Подменю	—

Подменю Shape должно содержать пункты, перечисленные в табл. 6.9.

Таблица 6.9. Пункты подменю Shape (2-й уровень)

Имя пункта	Тип пункта	Идентификатор
&Rectangle	Команда	ID_RECTANGLE
Rhom&b	Команда	ID_RHOMB
&Ellipse	Команда	ID_ELLIPSE

Подменю Color должно содержать пункты, перечисленные в табл. 6.10.

Таблица 6.10. Пункты подменю Color (2-й уровень)

Имя пункта	Тип пункта	Идентификатор
&Red	Команда	ID_RED
&Green	Команда	ID_GREEN
&Blue	Команда	ID_BLUE
—	SEPARATOR	—
&Dark	Команда	ID_DARK
&Medium	Команда	ID_MEDIUM
&Light	Команда	ID_LIGHT

Теперь добавьте к проекту новый файл с именем MenuDemo1.cpp и введите в файл исходный код, приведенный в листинге 6.1¹.

Листинг 6.1. Проект MenuDemo1

```
///////////////////////////////
// MenuDemo1.cpp
#include <windows.h>
#include <stdio.h>
#include "KWnd.h"
#include "resource.h"

#define W 200 // ширина фигуры
#define H 140 // высота фигуры
```

продолжение ↗

¹ Исходные тексты программ, приведенных в книге, можно скачать с сайта издательства <http://www.piter.com>.

Листинг 6.1 (продолжение)

```

enum ShapeSize { MAX, MIN };

typedef struct {
    int id_shape; // идентификатор фигуры
    BOOL fRed; // компонент красного цвета
    BOOL fGreen; // компонент зеленого цвета
    BOOL fBlue; // компонент синего цвета
    int id_bright; // идентификатор яркости цвета
} ShapeData;

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    hWnd mainWnd("MenuDemo1", hInstance, nCmdShow, WndProc,
        MAKEINTRESOURCE(IDR_MENU1), 100, 100, 400, 300);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    RECT rect;
    static HMENU hMenu; // дескриптор главного меню
    int x0, y0, x1, y1, x2, y2;
    POINT pt[4];
    static ShapeSize shapeSize = MIN;
    static BOOL bShow = TRUE;
    static HBRUSH hBrush, hOldBrush;
    char* itemResizeName[2] = { "Decrease!", "Increase!" };
    int intensity[3] = { 85, 170, 255 }; // интенсивность RGB-компонентов цвета
    int brightness;
    static ShapeData shapeData;

    switch (uMsg)
    {
    case WM_CREATE:
        hMenu = GetMenu(hWnd);
        // IDM_OPEN - пункт меню, применяемый по умолчанию
        SetMenuItemDefaultItem(GetSubMenu(hMenu, 0), IDM_OPEN, FALSE);

        // Исходные отметки пунктов меню
        CheckMenuRadioItem(GetSubMenu(hMenu, 1), IDM_SHOW_SHAPE,
            IDM_HIDE_SHAPE, IDM_SHOW_SHAPE, MF_BYCOMMAND);
        CheckMenuRadioItem(GetSubMenu(hMenu, 2), ID_RECTANGLE,
            ID_ELLIPSE, ID_RECTANGLE, MF_BYCOMMAND);
        CheckMenuRadioItem(GetSubMenu(hMenu, 2), ID_DARK,
            ID_LIGHT, ID_DARK, MF_BYCOMMAND);

        shapeData.id_shape = ID_RECTANGLE;
    }
}

```

```
shapeData.id_bright = ID_DARK;  
  
break;  
  
case WM_COMMAND:  
    switch (LOWORD(wParam))  
    {  
        case IDM_OPEN:  
            MessageBox(hWnd, "Выбран пункт 'Open'", "Меню File", MB_OK);  
            break;  
        case IDM_CLOSE:  
            MessageBox(hWnd, "Выбран пункт 'Close'", "Меню File", MB_OK);  
            break;  
        case IDM_SAVE:  
            MessageBox(hWnd, "Выбран пункт 'Save'", "Меню File", MB_OK);  
            break;  
        case IDM_EXIT:  
            SendMessage(hWnd, WM_DESTROY, 0, 0);  
            break;  
  
        case IDM_SHOW_SHAPE:  
            bShow = TRUE;  
            CheckMenuItem(GetSubMenu(hMenu, 1), IDM_SHOW_SHAPE,  
                          IDM_HIDE_SHAPE, LOWORD(wParam), MF_BYCOMMAND);  
            EnableMenuItem(hMenu, IDM_RESIZE, MF_BYCOMMAND | MFS_ENABLED);  
            DrawMenuBar(hWnd);  
            break;  
        case IDM_HIDE_SHAPE:  
            bShow = FALSE;  
            CheckMenuItem(GetSubMenu(hMenu, 1), IDM_SHOW_SHAPE,  
                          IDM_HIDE_SHAPE, LOWORD(wParam), MF_BYCOMMAND);  
            EnableMenuItem(hMenu, IDM_RESIZE, MF_BYCOMMAND | MFS_GRAYED);  
            DrawMenuBar(hWnd);  
            break;  
  
        case ID_RECTANGLE:  
            shapeData.id_shape = ID_RECTANGLE;  
            CheckMenuItem(GetSubMenu(hMenu, 2), ID_RECTANGLE,  
                          ID_ELLIPSE, LOWORD(wParam), MF_BYCOMMAND);  
            break;  
        case ID_RHOMB:  
            shapeData.id_shape = ID_RHOMB;  
            CheckMenuItem(GetSubMenu(hMenu, 2), ID_RECTANGLE,  
                          ID_ELLIPSE, LOWORD(wParam), MF_BYCOMMAND);  
            break;  
        case ID_ELLIPSE:  
            shapeData.id_shape = ID_ELLIPSE;  
            CheckMenuItem(GetSubMenu(hMenu, 2), ID_RECTANGLE,  
                          ID_ELLIPSE, LOWORD(wParam), MF_BYCOMMAND);  
            break;  
  
        case ID_RED:  
            shapeData.fRed = ~shapeData.fRed;  
            CheckMenuItem(GetSubMenu(hMenu, 2), ID_RED,  
                          MF_BYCOMMAND | shapeData.fRed? MF_CHECKED : MF_UNCHECKED);  
            break;  
        case ID_GREEN:  
            shapeData.fGreen = ~shapeData.fGreen;
```

продолжение ⤵

Листинг 6.1 (продолжение)

```

CheckMenuItem(GetSubMenu(hMenu, 2), ID_GREEN,
    MF_BYCOMMAND | shapeData.fGreen? MF_CHECKED : MF_UNCHECKED);
break;
case ID_BLUE:
    shapeData.fBlue = ~shapeData.fBlue;
    CheckMenuItem(GetSubMenu(hMenu, 2), ID_BLUE,
        MF_BYCOMMAND | shapeData.fBlue? MF_CHECKED : MF_UNCHECKED);
break;

case ID_DARK:
    shapeData.id_bright = ID_DARK;
    CheckMenuItemRadioItem(GetSubMenu(hMenu, 2), ID_DARK,
        ID_LIGHT, LOWORD(wParam), MF_BYCOMMAND);
break;
case ID_MEDIUM:
    shapeData.id_bright = ID_MEDIUM;
    CheckMenuItemRadioItem(GetSubMenu(hMenu, 2), ID_DARK,
        ID_LIGHT, LOWORD(wParam), MF_BYCOMMAND);
break;
case ID_LIGHT:
    shapeData.id_bright = ID_LIGHT;
    CheckMenuItemRadioItem(GetSubMenu(hMenu, 2), ID_DARK,
        ID_LIGHT, LOWORD(wParam), MF_BYCOMMAND);
break;

case IDM_RESIZE:
    shapeSize = (shapeSize == MIN)? MAX : MIN;
    ModifyMenu(hMenu, IDM_RESIZE, MF_BYCOMMAND, IDM_RESIZE,
        itemResizedName[shapeSize]);
    DrawMenuBar(hWnd);
break;

case IDM_ABOUT:
    MessageBox(hWnd,
        "MenuDemo1\nVersion 1.0\nCopyright: Finesoft Corporation, 2005.",
        "About MenuDemo1", MB_OK);
break;

default:
    break;
}
InvalidateRect(hWnd, NULL, TRUE);
break;

case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);
    brightness = intensity[shapeData.id_bright - ID_DARK];

    if (bShow) {
        hBrush = CreateSolidBrush(RGB(
            shapeData.fRed? brightness : 0,
            shapeData.fGreen? brightness : 0,
            shapeData.fBlue? brightness : 0));
        hOldBrush = (HBRUSH)SelectObject(hDC, hBrush);

        // Определение центра фигуры (x0, y0)
        GetClientRect(hWnd, &rect);
    }
}

```

```
x0 = rect.right / 2;
y0 = rect.bottom / 2;
// Координаты прямоугольника и эллипса
if (shapeSize == MIN) {
    x1 = x0 - W/2;    y1 = y0 - H/2;
    x2 = x0 + W/2;    y2 = y0 + H/2;
}
else {
    x1 = y1 = 0;
    x2 = rect.right;  y2 = rect.bottom;
}
// Координаты ромба
pt[0].x = (x1 + x2) / 2;  pt[0].y = y1;
pt[1].x = x2;              pt[1].y = (y1 + y2) / 2;
pt[2].x = (x1 + x2) / 2;  pt[2].y = y2;
pt[3].x = x1;              pt[3].y = (y1 + y2) / 2;

switch (shapeData.id_shape) {
case ID_RECTANGLE:
    Rectangle(hdc, x1, y1, x2, y2);
    break;
case ID_RHOMB:
    Polygon(hdc, pt, 4);
    break;
case ID_ELLIPSE:
    Ellipse(hdc, x1, y1, x2, y2);
    break;
}

DeleteObject(SelectObject(hdc, hOldBrush));
}

EndPaint(hWnd, &ps);
break;

case WM_DESTROY:
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
////////////////////////////////////////////////////////////////
```

Обратите внимание на некоторые детали реализации в программном коде. Так, меню, определенное в файле ресурсов `MenuDemo1.rc` с идентификатором `IDR_MENU1`, подключается к окну приложения при объявлении объекта `mainWnd` класса `KWnd`:

```
KWnd mainWnd("MenuDemo1", hInstance, nCmdShow, WndProc, MAKEINTRESOURCE(IDR_MENU1),
100, 100, 400, 300);
```

Текущие параметры рисуемой фигуры хранятся в переменной `shapeData` структурного типа `ShapeData`.

В блоке обработки сообщения `WM_CREATE` сначала определяется значение дескриптора `hMenu` главного меню, а затем вызывается функция `SetMenuItemDefault`

для установки атрибута «применяемый по умолчанию» пункту Open подменю File. Дескриптор подменю File, передаваемый первому параметру данной функции, получен вызовом функции GetSubMenu(hMenu, 0).

Далее трижды вызывается функция CheckMenuItem для установки исходных отметок в группах пунктов, интерпретируемых как переключатели:

Группа	Первый пункт в группе	Последний пункт в группе
Подменю View	IDM_SHOW_SHAPE	IDM_HIDE_SHAPE
Подменю Shape	ID_RECTANGLE	ID_ELLIPSE
Подменю Color	ID_DARK	ID_LIGHT

Отметки ставятся на пунктах, заданных четвертым параметром функции CheckMenuItem. Следует отметить, что эти пункты соответствуют начальным значениям переменных bShow, shapeData.id_shape и shapeData.id_bright. В дальнейшем при обработке соответствующих сообщений отметки переходят на другие пункты внутри каждой группы.

Пункты другой группы подменю, Color, интерпретируемые как флаги, получают свои отметки при помощи функции CheckMenuItem, когда обрабатывается сообщение WM_COMMAND с параметром wParam, младшее слово которого равно IDM_RED, IDM_GREEN или IDM_BLUE.

Пункт главного меню Increase! с идентификатором IDM_RESIZE используется как флагок, переводящий переменную shapeSize из значения MIN в значение MAX и обратно.

Но пункты главного меню не могут получать отметку введенного флагка. Поэтому, обрабатывая сообщение WM_COMMAND с параметром wParam, равным IDM_RESIZE, программа изменяет имя пункта от Increase к Decrease и обратно. Эта смена имени обеспечивается вызовом функции ModifyMenu. Символьная строка, назначаемая пункту, извлекается из массива itemResizeName. После возврата из ModifyMenu вызывается функция DrawMenuBar, чтобы немедленно отобразить изменения в полосе главного меню.

Когда изображение скрыто после выполнения команды Hide shape из подменю View, действия по изменению размеров фигуры теряют смысл. Интерфейс программы только выиграет, если в этой ситуации пункт меню IDM_RESIZE сделать недоступным. При обработке сообщения WM_COMMAND с параметром wParam, равным IDM_HIDE_SHAPE, вызывается функция EnableMenuItem, которой передается флаг MFS_GRAYED. В результате этого указанный пункт меню становится недоступным. При обработке сообщения WM_COMMAND с параметром wParam, равным IDM_SHOW_SHAPE, вызывается функция EnableMenuItem с флагом MFS_ENABLED, и пункт меню IDM_RESIZE снова становится доступным для пользователя.

При обработке сообщения WM_COMMAND с параметром wParam, равным IDM_ABOUT, вызывается функция MessageBox, которая выводит окно сообщений с информацией о программе и ее разработчиках.

В финальной части обработки сообщения WM_COMMAND вызывается функция InvalidateRect, чтобы перерисовать клиентскую область окна с учетом изменившегося состояния приложения.

Рекомендуем вам откомпилировать эту программу и посмотреть, как она работает.

Работа с контекстным меню

Контекстное меню, называемое иногда меню быстрого вызова (*shortcut menu*), появляется в любой части окна приложения при щелчке правой кнопкой мыши. Оно выглядит как всплывающее меню, но без привязки к меню верхнего уровня. Обычно содержание контекстного меню изменяется в зависимости от «контекста» — места, где произведен щелчок правой кнопкой мыши. В сложных приложениях контекстные меню часто содержат пункты, дублирующие команды основного меню, но сгруппированные иначе, чтобы максимально облегчить пользователю работу с приложением. После выбора пункта контекстного меню оно исчезает с экрана.

Конечно же, для программиста создание и обработка контекстного меню имеют свою специфику.

Определение шаблона контекстного меню

Создавая шаблон контекстного меню с помощью редактора меню, определите нулевой пункт меню нулевого уровня как подменю, имеющее какое-нибудь условное имя. Этот пункт нигде не будет отображаться. Он необходим только для получения дескриптора подменю с помощью функции `GetSubMenu`.

Затем определите подменю (меню 1-го уровня) по процедуре, описанной выше.

Сохраните созданный шаблон в файле описания ресурсов, назначив ему подходящий идентификатор, например `IDR_MENU_MYCONTEXT`.

Загрузка меню

Загрузка контекстного меню осуществляется в блоке обработки сообщения `WM_CREATE` при помощи функций `LoadMenu` и `GetSubMenu`:

```
HMENU hMenuMyContext;
hMenuMyContext = LoadMenu(hInstance, MAKEINTRESOURCE(IDR_MENU_MYCONTEXT));
hMenuMyContext = GetSubMenu(hMenuMyContext, 0);
```

Функция `LoadMenu` возвращает дескриптор меню, определенного в файле ресурсов с идентификатором `IDR_MENU_MYCONTEXT`. Этот дескриптор относится к фиктивному меню нулевого уровня, которое не должно отображаться на экране. Содержанием контекстного меню является нулевой пункт указанного меню, поэтому окончательное значение дескриптора `hMenuMyContext` определяется вызовом функции `GetSubMenu`.

Полученный дескриптор `hMenuMyContext` потом передается функции `TrackPopupMenuEx`, которая выводит всплывающее контекстное меню на экран.

Вызов меню

Когда пользователь делает щелчок правой кнопкой мыши, Windows отправляет окну приложения сообщение `WM_RBUTTONDOWN`, содержащее клиентские координаты¹ курсора мыши в момент щелчка. Кроме того, Windows отправляет сообщение `WM_CONTEXTMENU`, содержащее экранные координаты (*screen coordinates*) курсора

¹ Координаты указываются относительно левого верхнего угла клиентской области окна.

мыши. Вы можете организовать вызов контекстного меню, обрабатывая любое из этих сообщений, при помощи функции `TrackPopupMenuEx`:

```
BOOL TrackPopupMenuEx (
    HMENU hmenu,          // дескриптор контекстного меню
    UINT fuFlags,          // флаги
    int x,                 // горизонтальная позиция
    int y,                 // вертикальная позиция
    HWND hwnd,             // дескриптор окна
    LPTPMPARAMS lptpm // область экрана, которую меню не должно перекрывать
);
```

Второй параметр функции может содержать флаги, управляющие размещением полосы меню относительно позиции курсора мыши (*x*, *y*), и флаги, задающие некоторые другие характеристики. Обычно можно воспользоваться флагами по умолчанию, передавая этому параметру нулевое значение. По умолчанию левый верхний угол всплывающего меню привязывается к точке (*x*, *y*).

Параметрам *x* и *y* необходимо передать экranные координаты курсора мыши в момент щелчка правой кнопкой. Координаты курсора мыши можно извлечь из параметра lParam сообщения `WM_RBUTTONDOWN` или сообщения `WM_CONTEXTMENU`. Нужно только учесть, что сообщение `WM_RBUTTONDOWN` содержит клиентские координаты, поэтому при работе с этим сообщением необходимо преобразовать клиентские координаты в экranные координаты при помощи функции `ClientToScreen`.

Параметр *hwnd* содержит дескриптор окна, владеющего контекстным меню. Все сообщения от меню будут направляться этому окну. При помощи параметра *lptpm* можно определить область экрана, которую меню не должно перекрывать. Чаще всего этому параметру передается значение `NULL`.

Функция не возвращает управление до тех пор, пока работа пользователя с меню не будет завершена выбором пункта или отказом от выбора.

Обработка сообщений, направляемых окну-владельцу от контекстного меню, не отличается от обработки сообщений, источником которых является основное меню.

Приемы работы с контекстным меню продемонстрируем на примере разработки конкретного приложения.

Приложение MenuDemo2

Приложение `MenuDemo2` решает те же задачи, что и приложение `MenuDemo1`, отличаясь от него только организацией меню. Главное меню приложения `MenuDemo2` содержит те же пункты, что и приложение `MenuDemo1`, за вычетом пункта `Draw shape`. Подменю второго уровня `Shape` и `Color` для отсутствующего пункта `Draw shape` должны быть реализованы в виде контекстных меню.

Первое контекстное меню, всплывающее при щелчке мышью в левой половине окна, позволяет выбрать вид фигуры. Второе контекстное меню, всплывающее при щелчке мышью в правой половине окна, позволяет выбрать цвет заливки фигуры.

Создайте новый проект с именем `MenuDemo2`. Скопируйте из папки проекта `MenuDemo1` в папку проекта `MenuDemo2` файлы с расширениями `.cpp`, `.h` и `.rc`, скорректировав их имена заменой `MenuDemo1` на `MenuDemo2`. Добавьте эти файлы в состав проекта.

Откройте в окне редактирования файл resource.h и найдите строку комментария

```
// Used by MenuDemo1.rc
```

Замените ее следующей строкой:

```
// Used by MenuDemo2.rc
```

В окне **Workspace** перейдите на вкладку **ResourceView**, откройте в иерархическом списке папку **Menu** и двойным щелчком мыши на элементе **IDR_MENU1** вызовите редактор меню.

Из меню нужно удалить пункт **Draw shape**, выделив его мышью и нажав клавишу **Delete**. Среда разработки Visual Studio не позволит сделать это сразу, выведя предупреждение в виде диалогового окна (рис. 6.7).

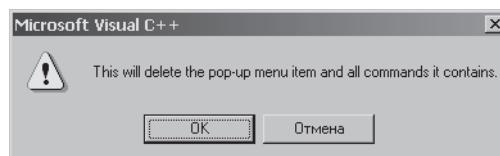


Рис. 6.7. Предупреждение об уничтожении всех пунктов подменю Draw shape

Нажмите клавишу **OK** для подтверждения удаления пункта меню.

Теперь нужно определить два новых шаблона меню, которые будут использоваться для контекстных меню. Эти шаблоны должны получить идентификаторы **IDR_MENU_SHAPE** и **IDR_MENU_COLOR** соответственно.

Меню **IDR_MENU_SHAPE** должно иметь фиктивный пункт нулевого уровня с условным именем **shape**, а подменю **shape** должно содержать пункты, указанные в табл. 6.9. Меню **IDR_MENU_COLOR** должно иметь фиктивный пункт нулевого уровня с условным именем **color**. Подменю **color** должно содержать пункты, указанные в табл. 6.10.

Код модуля MenuDemo2.cpp приведен в листинге 6.2.

Листинг 6.2. Проект MenuDemo2

```
//////////  
// MenuDemo2.cpp  
#include <windows.h>  
#include <stdio.h>  
#include "Klhd.h"  
#include "resource.h"  
  
#define W 200 // ширина фигуры  
#define H 140 // высота фигуры  
enum ShapeSize { MAX, MIN };  
  
typedef struct {  
    int id_shape; // идентификатор фигуры  
    BOOL fRed; // компонент красного цвета  
    BOOL fGreen; // компонент зеленого цвета  
    BOOL fBlue; // компонент синего цвета  
    int id_bright; // идентификатор яркости цвета  
} ShapeData;  
  
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
```

продолжение ↗

Листинг 6.2 (продолжение)

```

//=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("MenuDemo2", hInstance, nCmdShow, WndProc,
        MAKEINTRESOURCE(IDR_MENU1), 100, 100, 400, 300);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    RECT rect;
    static HMENU hMenu;           // дескриптор главного меню
    static HMENU hMenuShape;      // дескриптор 1-го контекстного меню
    static HMENU hMenuColor;      // дескриптор 2-го контекстного меню
    POINT point, ptInClient;
    int x0, y0, x1, y1, x2, y2;
    POINT pt[4];
    static ShapeSize shapeSize = MIN;
    static BOOL bShow = TRUE;
    static HBRUSH hBrush, hOldBrush;
    char* itemResizedName[2] = { "Decrease!", "Increase!" };
    int intensity[3] = { 85, 170, 255 }; // интенсивность RGB-компонентов цвета
    int brightness;
    static ShapeData shapeData;

    switch (uMsg)
    {
    case WM_CREATE:
        hMenu = GetMenu(hWnd);
        hMenuShape = LoadMenu(GetModuleHandle(NULL),
            MAKEINTRESOURCE(IDR_MENU_SHAPE));
        hMenuShape = GetSubMenu(hMenuShape, 0);

        hMenuColor = LoadMenu(GetModuleHandle(NULL),
            MAKEINTRESOURCE(IDR_MENU_COLOR));
        hMenuColor = GetSubMenu(hMenuColor, 0);

        // IDM_OPEN - пункт меню, применяемый по умолчанию
        SetMenuItemDefaultItem(GetSubMenu(hMenu, 0), IDM_OPEN, FALSE);

        // Исходные отметки пунктов меню
        CheckMenuItem(GetSubMenu(hMenu, 1), IDM_SHOW_SHAPE,
            IDM_HIDE_SHAPE, IDM_SHOW_SHAPE, MF_BYCOMMAND);
        CheckMenuItem(hMenuShape, ID_RECTANGLE,
            ID_ELLIPSE, ID_RECTANGLE, MF_BYCOMMAND);
        CheckMenuItem(hMenuColor, ID_DARK,
            ID_LIGHT, ID_DARK, MF_BYCOMMAND);

        shapeData.id_shape = ID_RECTANGLE;
    }
}

```

```
shapeData.id_bright = ID_DARK;
break;

case WM_CONTEXTMENU:
    point.x = LOWORD(lParam);
    point.y = HIWORD(lParam);
    ptInClient = point;
    ScreenToClient(hWnd, &ptInClient);
    GetClientRect(hWnd, &rect);

    if (ptInClient.x < rect.right / 2)
        TrackPopupMenuEx(hMenuShape, 0, point.x, point.y, hWnd, NULL);
    else
        TrackPopupMenuEx(hMenuColor, 0, point.x, point.y, hWnd, NULL);
    break;

case WM_COMMAND:
    switch (LOWORD(wParam))
    {
        /* код обработки для значений параметра: IDM_OPEN, IDM_CLOSE,
        IDM_SAVE, IDM_EXIT, IDM_SHOW_SHAPE, IDM_HIDE_SHAPE - из листинга 6.1 */
    }

    case ID_RECTANGLE:
        shapeData.id_shape = ID_RECTANGLE;
        CheckMenuItem(hMenuShape, ID_RECTANGLE,
                      ID_ELLIPSE, LOWORD(wParam), MF_BYCOMMAND);
        break;
    case ID_RHOMB:
        shapeData.id_shape = ID_RHOMB;
        CheckMenuItem(hMenuShape, ID_RECTANGLE,
                      ID_ELLIPSE, LOWORD(wParam), MF_BYCOMMAND);
        break;
    case ID_ELLIPSE:
        shapeData.id_shape = ID_ELLIPSE;
        CheckMenuItem(hMenuShape, ID_RECTANGLE,
                      ID_ELLIPSE, LOWORD(wParam), MF_BYCOMMAND);
        break;

    case ID_RED:
        shapeData.fRed = ~shapeData.fRed;
        CheckMenuItem(hMenuColor, ID_RED,
                      MF_BYCOMMAND | shapeData.fRed? MF_CHECKED : MF_UNCHECKED);
        break;
    case ID_GREEN:
        shapeData.fGreen = ~shapeData.fGreen;
        CheckMenuItem(hMenuColor, ID_GREEN,
                      MF_BYCOMMAND | shapeData.fGreen? MF_CHECKED : MF_UNCHECKED);
        break;
    case ID_BLUE:
        shapeData.fBlue = ~shapeData.fBlue;
        CheckMenuItem(hMenuColor, ID_BLUE,
                      MF_BYCOMMAND | shapeData.fBlue? MF_CHECKED : MF_UNCHECKED);
        break;

    case ID_DARK:
        shapeData.id_bright = ID_DARK;
```

продолжение ↗

Листинг 6.2 (продолжение)

```

        CheckMenuItem(hMenuColor, ID_DARK,
                      ID_LIGHT, LOWORD(wParam), MF_BYCOMMAND);
        break;
    case ID_MEDIUM:
        shapeData.id_bright = ID_MEDIUM;
        CheckMenuItem(hMenuColor, ID_DARK,
                      ID_LIGHT, LOWORD(wParam), MF_BYCOMMAND);
        break;
    case ID_LIGHT:
        shapeData.id_bright = ID_LIGHT;
        CheckMenuItem(hMenuColor, ID_DARK,
                      ID_LIGHT, LOWORD(wParam), MF_BYCOMMAND);
        break;

/* код обработки для case IDM_RESIZE - из листинга 6.1 */

case IDM_ABOUT:
    MessageBox(hWnd,
              "MenuDemo2\nVersion 1.0\nCopyright: Finesoft Corporation, 2005.",
              "About MenuDemo2", MB_OK);
    break;
default:
    break;
}
InvalidateRect(hWnd, NULL, TRUE);
break;

case WM_PAINT:
/* код обработки сообщения WM_PAINT - из листинга 6.1 */
break;

case WM_DESTROY:
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
/////////////////////////////////////////////////////////////////

```

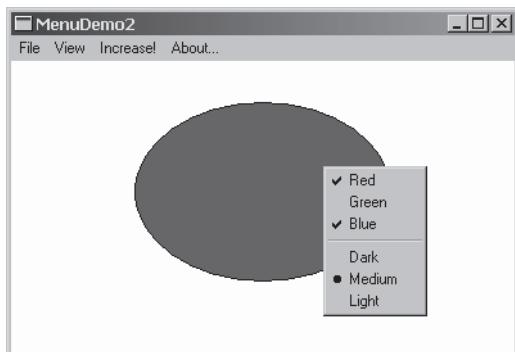


Рис. 6.8. Окно приложения MenuDemo2 с контекстным меню в правой части окна

При обработке сообщения WM_CONTEXTMENU необходимо выяснить, в какой части окна произведен щелчок правой кнопкой мыши. Для этого экранные координаты курсора мыши, сохраненные в переменной `point`, преобразуются к клиентским координатам, представленным переменной `ptInClient`. Размеры клиентской области окна определяются в переменной `rect` после вызова функции `GetClientRect`.

Если щелчок произошел в левой части окна, то функции `TrackPopupMenuEx` передается дескриптор меню `hMenuShape`. Если щелчок был выполнен в правой части окна, то функции `TrackPopupMenuEx` передается дескриптор меню `hMenuColor`.

На рис. 6.8 показан вид окна приложения с контекстным меню `hMenuColor`.

Быстрые клавиши

Быстрая клавиша (keyboard accelerator) — это клавиша или комбинация клавиш, которые при нажатии генерируют сообщение `WM_COMMAND` или `WM_SYSCOMMAND`.

Обычно быстрые клавиши дублируют пункты меню, предоставляемые пользователю альтернативный способ вызова команд. На рис. 6.2 мы уже видели обозначения быстрых клавиш на полосе подменю `File` в окне приложения `Visual Studio`. Например, команду `New` пользователь может вызвать, либо выбрав пункт `File ▶ New`, либо нажав сочетание клавиш `Ctrl+N`. Очевидно, что второй способ быстрее.

Хотя традиционно быстрые клавиши являются эквивалентом пунктов-команд меню, они могут генерировать и такие команды, которых нет в меню.

Для многооконных приложений с множеством оконных процедур быстрые клавиши очень важны. Как известно, Windows посыпает сообщения клавиатуры оконной процедуре того окна, которое в данный момент имеет фокус ввода. Однако при работе с быстрыми клавишами Windows посыпает сообщение `WM_COMMAND` той оконной процедуре, чей дескриптор был передан функции `TranslateAccelerator`. Как правило, это будет оконная процедура главного окна приложения, то есть именно того окна, в котором расположено меню. Следовательно, нет необходимости дублировать логику обработки быстрых клавиш в каждой оконной процедуре.

Чтобы добавить в приложение обработку быстрых клавиш, нужно выполнить последовательность действий:

1. Модифицировать определение ресурса меню, добавив к имени каждого дублируемого пункта информацию о быстрой клавише.
2. Определить таблицу быстрых клавиш в файле описания ресурсов.
3. Обеспечить загрузку таблицы быстрых клавиш в память приложения.
4. Модифицировать цикл обработки сообщений в функции `WinMain`.

Модификация определения ресурса меню

Говоря о модификации, мы подразумеваем, что ресурс меню уже был определен ранее, без учета использования быстрых клавиш. В принципе, ничто не мешает вам заранее продумать вопрос дублирования меню быстрыми клавишами и сделать те добавления, о которых говорится здесь, на этапе создания меню.

Для модификации определения пункта вызовите диалоговое окно `Menu Item Properties` (см. рис. 6.5) и в конце имени пункта, которое указано в поле `Caption`, добавьте текстовую строку `\t<обозначение быстрой клавиши>`.

Например, если добавляется быстрая клавиша Ctrl+0 для дублирования пункта &Open... из подменю File, то следует добавить к прежнему имени текстовую строку \tCtrl+0. После модификации поле Caption будет содержать строку &Open...\tCtrl+0.

Таблица быстрых клавиш

Эта таблица обычно создается при помощи *редактора таблиц быстрых клавиш*. Для его запуска нужно выполнить команду меню Insert ▶ Resource в главном меню приложения Visual Studio. В появившемся диалоговом окне Insert Resource следует указать тип ресурса Accelerator, после чего нажать кнопку New. В результате будет открыто окно редактора таблицы быстрых клавиш (рис. 6.9).



Рис. 6.9. Окно редактора таблицы быстрых клавиш

По умолчанию редактор присваивает таблице быстрых клавиш имя IDR_ACCELERATOR1. В файле resource.h этот идентификатор определяется как целочисленная константа. В большинстве случаев можно согласиться с этим именем по умолчанию. После двойного щелчка на пустой строке таблицы открывается диалоговое окно Accel Properties (рис. 6.10).

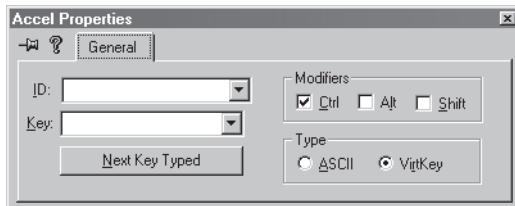


Рис. 6.10. Диалоговое окно Accel Properties

В поле ID введите идентификатор быстрой клавиши. Если быстрая клавиша дублирует пункт меню, то этот идентификатор должен совпадать с идентификатором пункта меню. Например, если определяется быстрая клавиша для пункта &Open... подменю File для приложения MenuDemo1, следует ввести идентификатор IDM_OPEN.

В группе переключателей Type оставьте включенным переключатель VirtKey, так как в поле Key будет указываться виртуальная клавиша. Коды виртуальных клавиш можно посмотреть в табл. 4.3 (глава 4).

Введите в поле Key идентификатор нужной клавиши. Для цифр и латинских букв в качестве идентификатора вводится сам символ.

В группе Modifiers установите те флашки, которые соответствуют дополнительным клавишам, входящим в определяемую комбинацию клавиш.

Например, если определяется быстрая клавиша Ctrl+0 для пункта &Open..., то следует оставить включенным флашок Ctrl, а в поле Key ввести символ 0 (строчная буква «О» английского алфавита).

COBET

Теоретически, можно определить быструю клавишу почти для каждой виртуальной или символьной клавиши в сочетании с клавишами Shift, Ctrl или Alt. Однако надо избегать использования тех быстрых клавиш, которые имеют традиционное применение во многих приложениях Windows. Например, F1 обычно используется для вызова контекстной справки, а клавиши F4, F5 и F6 — для специальных функций многооконного интерфейса приложений. Нежелательно назначать быстрыми клавишами клавиши Tab, Enter, Esc и Spacebar, поскольку они часто используются для системных функций. Например, Alt+F4 закрывает приложение или окно, Alt+Tab переключает на следующее окно, а Ctrl+Esc вызывает меню Старт.

Описанная процедура определения строки в таблице быстрых клавиш повторяется для задания информации о каждой быстрой клавише в обрабатываемом меню. После этого определенный ресурс сохраняется в файле описания ресурсов.

Загрузка таблицы быстрых клавиш

Во время работы программы для загрузки таблицы быстрых клавиш в память и получения ее дескриптора используется функция LoadAccelerators:

```
HACCEL LoadAccelerators (
    HINSTANCE hInstance, // дескриптор экземпляра приложения
    LPCTSTR lpTableName // имя таблицы быстрых клавиш
);
```

Для передачи второму параметру имени таблицы IDR_ACCELERATOR1, назначенного редактором ресурсов, необходимо использовать макрос MAKEINTRESOURCE.

В случае успешного выполнения функция возвращает дескриптор таблицы быстрых клавиш. В случае неудачного завершения функция возвращает значение NULL.

Модификация цикла обработки сообщений

Для обработки быстрых клавиш приложение должно перехватывать сообщения клавиатуры, анализировать их коды и в случае совпадения с кодом, определенным в таблице быстрых клавиш, направлять соответствующее сообщение в оконную процедуру главного окна. Все это может выполнить функция TranslateAccelerator:

```
int TranslateAccelerator (
    HWND hWnd, // дескриптор окна - получателя сообщения
    HACCEL hAccTable, // дескриптор таблицы быстрых клавиш
    LPMMSG lpMsg // указатель на структуру MSG
);
```

Функция преобразует сообщение WM_KEYDOWN или WM_SYSKEYDOWN¹ в сообщение WM_COMMAND или WM_SYSCOMMAND, если таблица акселераторов содержит соответствующий код виртуальной клавиши (с учетом состояния клавиш Ctrl, Alt и Shift).

Сформированное сообщение, содержащее идентификатор быстрой клавиши в младшем слове параметра wParam, отправляется непосредственно в оконную процедуру, минуя очередь сообщений. Возврат из функции TranslateAccelerator происходит только после того, как оконная процедура обработает посланное сообщение.

Если функция TranslateAccelerator возвращает ненулевое значение, это значит, что преобразование комбинации клавиш и обработка отправленного сообщения завершились успешно. В этом случае приложение не должно повторно

¹ Сообщения клавиатуры рассматривались в главе 4.

обрабатывать эту комбинацию клавиш при помощи функций `TranslateMessage` и `DispatchMessage`.

Данное требование можно выполнить, если применяемый до сих пор традиционный цикл обработки сообщений

```
while (GetMessage(&msg, NULL, 0, 0)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

преобразовать к следующему виду:

```
while (GetMessage(&msg, NULL, 0, 0)) {
    if (!TranslateAccelerator(hWnd, hAccel, &msg)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

где `hMainWnd` — дескриптор главного окна приложения.

Приложение MenuDemo3

Приложение `MenuDemo3` является улучшенной версией приложения `MenuDemo1`. Оно выполняет те же функции и имеет точно такое же меню, но теперь программа позволяет использовать быстрые клавиши, дублирующие некоторые команды меню. Дублируемые пункты меню и соответствующие им быстрые клавиши перечислены в табл. 6.11.

Таблица 6.11. Спецификации быстрых клавиш для программы `MenuDemo3`

Подменю	Имя пункта	Идентификатор	Быстрая клавиша
File	&Open...	IDM_OPEN	Ctrl+O
	Sa&ve...	IDM_SAVE	Ctrl+S
Shape	&Rectangle	ID_RECTANGLE	F2
	Rhom&b	ID_RHOMB	F3
	&Ellipse	ID_ELLIPSE	F4
Color	&Red	ID_RED	Ctrl+R
	&Green	ID_GREEN	Ctrl+G
	&Blue	ID_BLUE	Ctrl+B
	&Dark	ID_DARK	F10
	&Medium	ID_MEDIUM	F11
	&Light	ID_LIGHT	F12

Создайте новый проект с именем `MenuDemo3`. Скопируйте из папки проекта `MenuDemo1` в папку проекта `MenuDemo3` файлы с расширениями `.cpp`, `.h` и `.rc`, скорректировав их имена заменой `MenuDemo1` на `MenuDemo3`. Добавьте скопированные файлы в состав проекта. В окне `Workspace` перейдите на вкладку `ResourceView` и вызовите редактор меню, сделав двойной щелчок мышью на значке `IDR_MENU1`. Теперь следует отредактировать пункты меню, указанные в табл. 6.11, добавив к имени каждого пункта символьную строку `\t<обозначение быстрой клавиши>`, взяв обозначения быстрых клавиш из четвертого столбца табл. 6.11.

Вызовите редактор таблицы быстрых клавиш. Определите быстрые клавиши, используя информацию из третьего и четвертого столбцов табл. 6.11. Для функ-

циональных клавиш F2, F3 и им подобных в поле Key следует указывать идентификаторы VK_F2, VK_F3¹ и т. д. Также следует сбросить флагок Ctrl.

Исходный код файла MenuDemo3.cpp приведен в листинге 6.3.

Листинг 6.3. Проект MenuDemo3

```
//////////  
// MenuDemo3.cpp  
#include <windows.h>  
#include <stdio.h>  
#include "KWnd.h"  
#include "resource.h"  
#define W 200 // ширина фигуры  
#define H 140 // высота фигуры  
enum ShapeSize { MAX, MIN };  
  
typedef struct {  
    int id_shape; // идентификатор фигуры  
    BOOL fRed; // компонент красного цвета  
    BOOL fGreen; // компонент зеленого цвета  
    BOOL fBlue; // компонент синего цвета  
    int id_bright; // идентификатор яркости цвета  
} ShapeData;  
  
HRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);  
=====  
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,  
    LPSTR lpCmdLine, int nCmdShow)  
{  
    HACCEL hAccel;  
    MSG msg;  
    KWnd mainWnd("MenuDemo3", hInstance, nCmdShow, WndProc,  
        MAKEINTRESOURCE(IDR_MENU1), 100, 100, 400, 300);  
  
    // Загрузка таблицы быстрых клавиш  
    hAccel = LoadAccelerators(hInstance, MAKEINTRESOURCE(IDR_ACCELERATOR1));  
  
    while (GetMessage(&msg, NULL, 0, 0)) {  
        // Если сообщение не от быстрой клавиши,  
        // то выполнить стандартную обработку  
        if (!TranslateAccelerator(mainWnd.GetHWnd(), hAccel, &msg)) {  
            TranslateMessage(&msg);  
            DispatchMessage(&msg);  
        }  
    }  
    return msg.wParam;  
}  
=====  
HRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)  
{  
    /* Текст функции из листинга 6.1 */  
}  
//////////
```

Откомпилируйте проект и протестируйте программу MenuDemo3.exe, вызывая команды меню с помощью быстрых клавиш.

¹ Коды виртуальных клавиш приведены в табл. 4.3 (глава 4).

7

Диалоговые окна

Диалоговые окна, или окна диалога (*dialog box*), являются одним из важнейших элементов графического интерфейса пользователя Windows-приложений. Обычно диалоговые окна используются для ввода пользователем дополнительной информации, а также для вывода результатов работы приложения.

Диалоговое окно имеет вид всплывающего окна с одним или несколькими элементами управления (*controls*), которые являются для него дочерними окнами. Используя элементы управления, пользователь вводит текст, выбирает указанные опции (флажки, переключатели, элементы списка) и нажимает кнопки, вызывающие различные действия приложения.

От обычных окон диалоговые окна отличаются тем, что они создаются на базе предопределенного в Windows класса диалоговых окон. Окненная процедура этого класса, спрятанная в недрах Windows, обеспечивает обработку сообщений, поступающих в диалоговое окно, а также задает специфическое поведение элементов управления диалогового окна. Например, она управляет передачей фокуса ввода от одного элемента другому или переносит фокус ввода между группами элементов при нажатии клавиши *Tab*. Эту невидимую для программиста оконную процедуру иногда называют менеджером диалогового окна (*dialog box manager*).

Менеджер диалогового окна передает многие сообщения в функцию, определенную в вашем приложении, которая называется процедурой диалогового окна, или просто диалоговой процедурой (*dialog procedure*). Эта процедура похожа на обычную оконную процедуру, но все же имеет некоторые важные особенности, которые будут рассмотрены ниже.

Второе отличие диалоговых окон от обычных окон состоит в том, что они всегда связаны с шаблоном диалога, содержащим размеры окна, состав и расположение элементов управления. Шаблон диалогового окна можно определить двумя способами: а) в файле описания ресурсов, используя редактор диалоговых окон, б) создавая шаблон в памяти в процессе работы приложения. В книге рассматривается только первый способ, благо он наиболее простой и распространенный.

Типы диалоговых окон

Диалоговые окна бывают модальные (*modal*) и немодальные (*modeless*). Они различаются по влиянию на дальнейшее выполнение программы.

Когда в программе вызывается модальное диалоговое окно, оно ожидает выполнения некоторого действия со стороны пользователя, прежде чем программа сможет продолжить свое выполнение. Пользователь не может переключиться между диалоговым окном и другими окнами программы. Он должен явно закрыть диалоговое окно, что обычно делается щелчком на кнопке *OK* или *Cancel*. Однако пользователь может переключаться в другие программы, не закрыв диалоговое окно.

Существует также специальный вид модальных диалоговых окон — *системные модальные (system modal)* окна, которые не позволяют переключаться даже в другие программы. Они сообщают о серьезных проблемах, и пользователь должен закрыть системное модальное окно, чтобы продолжить работу в Windows.

Немодальное диалоговое окно не приостанавливает выполнение программы. Оно может получать и терять фокус ввода. Это значит, что пользователь может свободно переключаться между диалоговым окном и другими окнами программы. Окна этого типа предпочтительней использовать в тех случаях, когда они содержат элементы управления, которые должны быть в любой момент доступны пользователю.

Элементы управления в диалоговом окне

Основную функциональную нагрузку в диалоговом окне выполняют элементы управления. Все версии Windows поддерживают так называемые *базовые элементы управления*, перечисленные в табл. 7.1.

Таблица 7.1. Базовые элементы управления

Элемент управления	Описание	Предопределенный оконный класс
Рисунок (Picture control)	Элемент управления, отображающий пустую прямоугольную рамку, закрашенную прямоугольную область или растровый образ	STATIC
Надпись (Static text)	Текстовая строка. Обычно используется как метка (поясняющая надпись) рядом с полем ввода или элементом управления другого типа. Может применяться как самостоятельная информационная надпись	STATIC
Рамка (Group box)	Прямоугольная рамка с надписью, используемая для группирования набора связанных элементов управления	BUTTON
Кнопка (Button)	Элемент, который пользователь «нажимает», чтобы выполнить какое-либо действие	BUTTON
Флажок (Check box)	Элемент, который может быть либо установлен, либо сброшен для выбора или отмены опции, которая не связана с другими опциями	BUTTON
Переключатель (Radio button)	Элемент, используемый для выбора одной из группы взаимоисключающих опций. В группе переключателей может быть выбран только один из них	BUTTON
Список (List box)	Прямоугольное окно со списком элементов (строк), из которого пользователь может выбирать любой элемент	LISTBOX

Таблица 7.1. (продолжение)

Элемент управления	Описание	Предопределенный оконный класс
Окно редактирования или текстовое поле (Edit box)	Прямоугольное окно для ввода текста с клавиатуры. Элемент предоставляет определенные средства редактирования текста	EDIT
Комбинированный список (Combo box)	Элемент, объединяющий список с окном редактирования	COMBOBOX
Полоса прокрутки (Scroll bar)	Элемент управления линейкой прокрутки	SCROLLBAR

Начиная с Windows 95, в системе используется библиотека элементов управления общего пользования (*common control library*). Новые элементы управления, включенные в эту библиотеку, дополняют базовые элементы управления и позволяют придать приложениям более совершенный вид. Работа с элементами управления общего пользования рассматривается в следующей главе.

Обычно элементы управления определяются в шаблоне диалогового окна на языке описания шаблона диалога. В случае создания шаблона с помощью редактора диалоговых окон эти определения элементов генерируются автоматически. Одним из атрибутов описания элемента управления в шаблоне диалога является *идентификатор элемента управления*.

Каждый элемент управления, описанный в шаблоне диалога, реализуется Windows в виде окна соответствующего класса. Это окно является дочерним окном по отношению к диалоговому окну. Как всякое окно, оно идентифицируется своим дескриптором типа **HWND**. Вместо термина «дескриптор окна элемента управления» в документации (MSDN) обычно используется более короткий термин «дескриптор элемента управления».

Если элемент управления определен в шаблоне диалога, то программисту известен только его идентификатор. В то же время многие функции, работающие с элементом управления, принимают в качестве параметра его дескриптор. Для получения дескриптора элемента управления по его идентификатору используется функция **GetDlgItem**:

```
HWND GetDlgItem (
    HWND hDlg,           // дескриптор диалогового окна
    int nIDDlgItem        // идентификатор элемента управления
);
```

В случае успешного завершения функция возвращает дескриптор элемента управления, в случае ошибки — значение **NULL**.

Иногда возникает необходимость обратного преобразования, чтобы по дескриптору элемента управления определить его идентификатор. Такую проблему решает вызов функции **GetDlgCtrlID**:

```
int GetDlgCtrlID (
    HWND hwndCtl   // дескриптор элемента управления
);
```

Ранее было сказано, что элементы управления обычно определяются в шаблоне диалогового окна. Существует, однако, и альтернативный способ создания и размещения элемента управления при помощи функции **CreateWindow**, первому параметру которой передается имя предопределенного оконного класса (см. табл. 1.7 в главе 1). Но такой подход используется гораздо реже.

Элементы управления могут быть *разрешенными* (*enabled*) или *запрещенными* (*disabled*). По умолчанию все элементы управления имеют статус разрешенных элементов. Запрещенные элементы выводятся на экран серым цветом и не воспринимают пользовательский ввод с клавиатуры или от мыши. Изменение статуса элементов управления осуществляется при помощи функции `EnableWindow`.

Создание и обработка диалогового окна

Создание диалогового окна и работа с ним требуют выполнения следующей последовательности действий:

1. Определение шаблона диалогового окна.
2. Определение диалоговой процедуры.
3. Вызов функции создания диалогового окна.
4. Обмен данными между диалоговой процедурой и вызывающей функцией окна.

Реализация некоторых шагов различается для модальных и немодальных диалоговых окон.

Шаблон диалогового окна

Определение шаблона диалогового окна в файле описания ресурсов имеет следующий вид:

```
имя_диалога DIALOG DISCARDABLE x, y, width, height  
[ опции_диалога ]  
BEGIN  
    Определение элементов диалогового окна  
END
```

Имя_диалога интерпретируется Windows либо как С-строка с завершающим нулевым символом, либо как целочисленный идентификатор. Правила интерпретации такие же, как и для имени пиктограммы в файле описания ресурсов (см. главу 5).

Параметры *x*, *y*, *width*, *height* задают позицию и размеры диалогового окна. Координаты *x*, *y* измеряются относительно левого верхнего угла окна, из которого вызвано диалоговое окно. Все координаты и размеры задаются не в пикселях, а в специальных *шаблонных единицах* диалогового окна.

Опции диалога — это необязательные инструкции, в которых можно указывать значения `STYLE`, `CAPTION`, `FONT` и многие другие параметры.

Инструкция `STYLE` напоминает параметр `dwStyle` функции `CreateWindow`. Для модальных окон диалога обычно используются флаги `WS_POPUP` и `WS_DLGFRA ME`.

Инструкция `CAPTION` применяется для объявления диалоговых окон, которые имеют стиль `WS_CAPTION`. Наличие заголовка позволяет пользователям перемещать окно по экрану. Заголовок может также служить напоминанием о том, какое действие выполняется в этом окне.

Синтаксис описания элементов диалогового окна зависит от типа элементов. Нет нужды более подробно рассматривать язык описания шаблона диалогового окна, так как это описание создается автоматически при работе с редактором диалоговых окон.

Шаблонная система единиц

В описании шаблона диалогового окна применяется специальная шаблонная система единиц. Размеры и расположение диалогового окна, задаваемые в параметрах `x`, `y`, `width`, `height`, а также всех элементов управления должны быть указаны в единицах этой системы, называемых также *шаблонными единицами* (*dialog template units*).

Шаблонные единицы определяются отдельно по горизонтали и по вертикали через ширину и высоту используемого в диалоговом окне шрифта. Одна шаблонная единица по горизонтали составляет четверть средней ширины символов используемого шрифта. Одна шаблонная единица по вертикали составляет восьмую часть высоты символов.

По умолчанию в диалоговом окне используется системный шрифт. Поскольку высота символов системного шрифта примерно вдвое больше его ширины, то размеры делений по осям `X` и `Y` примерно одинаковы.

Среднюю ширину и высоту символов для системного шрифта можно определить с помощью функции `GetDialogBaseUnits`. Она возвращает 32-разрядное значение, младшее слово которого равно средней ширине символов в пикселях (`baseunitX`), а старшее слово — высоте символов в пикселях (`baseunitY`). Таким образом, для пересчета шаблонных единиц `templateunitX`, `templateunitY` в пиксели можно воспользоваться следующими выражениями:

```
pixelX = templateunitX * baseunitX / 4;  
pixelY = templateunitY * baseunitY / 8;
```

Подобная система единиц измерения позволяет операционной системе правильно масштабировать диалоговое окно независимо от размера шрифта, установленного в настройках экрана.

По умолчанию Windows использует «мелкий шрифт» (96 точек на дюйм). В этом режиме функция `GetDialogBaseUnits` возвращает значения `baseunitX = 8`, `baseunitY = 16`.

Но пользователь может выбрать более крупный шрифт. Для смены шрифта нужно щелкнуть правой кнопкой мыши на поверхности рабочего стола и в появившемся диалоговом окне **Свойства: Экран** нажать кнопку **Дополнительно...**, а затем в окне **Свойства** перейти на вкладку **Общие**. Комбинированный список **Размер шрифта** позволяет выбрать другой шрифт.

Если пользователь выберет «крупный шрифт» (120 точек на дюйм) и перезагрузит систему, чтобы новые параметры вступили в силу, то после этого функция `GetDialogBaseUnits` будет возвращать значения `baseunitX = 10`, `baseunitY = 20`. Windows автоматически увеличит размеры диалоговых окон, и поэтому все текстовые надписи по-прежнему будут размещены правильно.

Применяемая шаблонная система единиц дает еще одну возможность управления видом и размерами диалогового окна на стадии описания шаблона. Для этого достаточно в параметрах диалога выбрать другой шрифт при помощи инструкции `FONT`.

Модальный диалог

Техника добавления к приложению простейшего модального диалогового окна будет рассмотрена на примере разработки приложения `DlgDemo1`.

Приложение DlgDemo1 является модификацией программы MenuDemo1, которая была рассмотрена в предыдущей главе. Напомним, что в этой программе при выборе пункта меню *About...* вызывается функция *MessageBox* для вывода окна сообщений, содержащего информационный текст. Окно сообщений тоже является модальным диалоговым окном, но всю заботу о его создании и обработке берет на себя Windows. Но есть и обратная сторона подобной простоты создания информационного окна. Возможности программиста повлиять на внешний вид и содержимое подобного окна очень ограничены.

В приложении DlgDemo1 при выборе пункта меню *About...* будет вызываться настоящее модальное диалоговое окно, вид и поведение которого полностью определяются программистом. В остальном приложение DlgDemo1 должно работать так же, как и приложение MenuDemo1.

Как обычно, сначала следует создать новый проект с именем DlgDemo1. Скопируйте из папки проекта MenuDemo1 (см. листинг 6.1) в папку проекта DlgDemo1 файлы с расширениями .cpp, .h и .rc, скорректирував их имена заменой MenuDemo1 на DlgDemo1. Добавьте эти файлы в состав проекта.

Откройте в окне редактирования файл resource.h и замените в его тексте строку комментария

```
// Used by MenuDemo1.rc
```

следующей строкой:

```
// Used by DlgDemo1.rc
```

Модификацию программы начнем с определения шаблона диалогового окна.

Вызов и использование редактора диалоговых окон

В главном меню Visual Studio выполните команду *Insert ▶ Resource*. В появившемся окне *Insert Resource* выберите тип ресурса *Dialog* и нажмите кнопку *New*. В результате будет открыто окно редактора диалоговых окон с формой нового окна (рис. 7.1). Изначально в нем есть только кнопки *OK* и *Cancel*.

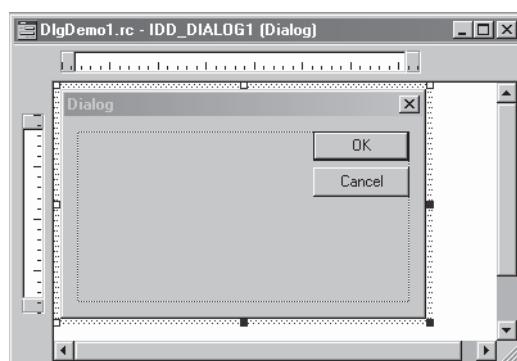


Рис. 7.1. Заготовка диалогового окна в окне редактора диалоговых окон

Также в окне приложения Visual Studio справа должна появиться панель инструментов *Controls* (рис. 7.2). Эта панель содержит кнопки для создания элементов управления всех типов, которые можно добавлять в диалоговое окно.

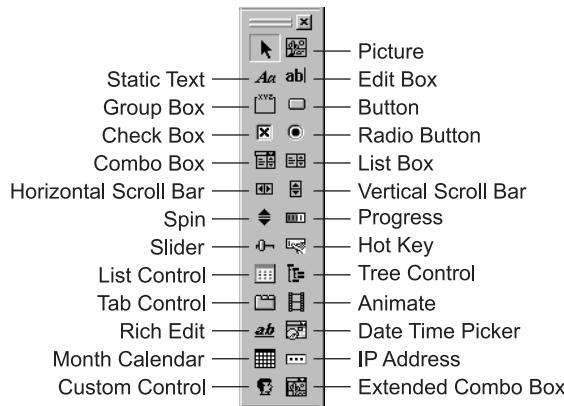


Рис. 7.2. Панель инструментов Controls

Если по каким-то причинам панель инструментов Controls не появится, то нужно щелкнуть правой кнопкой мыши на пустом месте окна Visual Studio и в открывшемся контекстном меню выбрать пункт Controls.

Проектирование шаблона диалогового окна начинается с установки его свойств.

Чтобы вызвать диалоговое окно Dialog Properties (рис. 7.3), нужно сделать двойной щелчок левой кнопкой мыши на форме диалога. Это окно можно вызывать и другим способом, щелкнув на форме правой кнопкой мыши и выбрав во всплывающем контекстном меню пункт Properties.

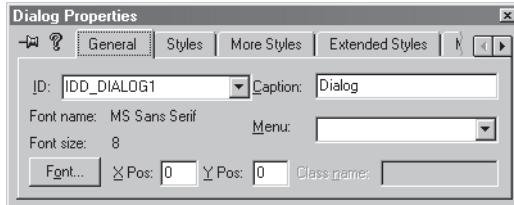


Рис. 7.3. Окно Dialog Properties

Введите в поле ID идентификатор IDD_ABOUT, а в поле Caption — текст About DlgDemo1.

Кнопка Font... предназначена для вызова диалога, позволяющего выбрать шрифт, но сейчас можно оставить шрифт, используемый по умолчанию (MS Sans Serif, размер 8). Другие параметры диалогового окна тоже можно оставить в стандартном варианте. Если переключиться на вкладку Styles, то можно увидеть, что по умолчанию используются стиль PopUp и тип рамки Dialog Frame, а также включены отметки флагков Title bar и System menu. После установки необходимых параметров окно свойств диалога нужно закрыть.

Теперь займемся «начинкой» диалогового окна. Мы хотели бы поместить какой-нибудь рисунок слева на форме, затем отобразить необходимые информационные надписи, а в правом нижнем углу расположить кнопку OK.

С помощью мыши увеличьте размеры формы примерно в полтора раза. Удалите кнопку Cancel. Чтобы удалить элемент управления, достаточно выделить его мышью и нажать клавишу Delete.

Переместите кнопку **OK** в правый нижний угол окна. Элементы управления перемещаются на форме или мышью, или при помощи клавиш-стрелок. В случае использования клавиатуры элемент предварительно нужно выделить.

Добавление элемента управления Рисунок

Предположим, что рисунок, который нужно добавить на форму диалога, находится в файле *Butterfly.bmp* и содержит изображение размером 64 × 90 пикселов, показанное на рис. 7.4.



Рис. 7.4. Растворный образ в файле *Butterfly.bmp*

Вы можете взять готовый рисунок, если загрузите исходный код проекта *DlgDemo1* из файлов, размещенных на сайте издательства «Питер» (www.piter.com). Впрочем, можно подготовить при помощи любого графического редактора другой рисунок примерно таких же размеров. Файл с рисунком должен находиться в папке проекта.

Прежде чем поместить растровый образ на форму диалога, нужно сначала включить его в состав ресурсов проекта. Эта процедура рассматривалась в пятой главе. Добавьте к создаваемому приложению ресурс растрового образа в режиме **Импорт**, связав ресурс с файлом *Butterfly.bmp* и назначив ему идентификатор **IDB_BUTTERFLY**.

Подготовительная работа завершена, и теперь можно приступить к размещению рисунка на форме диалога. Щелкните мышью на кнопке **Picture**, которая находится на панели инструментов **Controls**. Повторите щелчок мышью в том месте формы диалога, куда вы хотите добавить этот элемент (рис. 7.5).

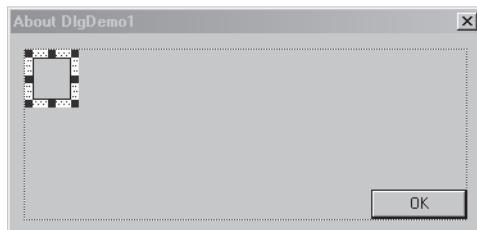


Рис. 7.5. Размещение элемента управления Picture

Во время дальнейшей разработки место размещения элемента можно скорректировать, используя мышь или клавиши-стрелки. Двойным щелчком мышью на элементе управления **Picture** вызовите диалоговое окно свойств этого элемента (рис. 7.6).

ПРИМЕЧАНИЕ

Обычно идентификаторы элементов управления начинаются с префикса **IDC_** (ID for a control). Впрочем, иногда мы будем отступать от этого правила и назначать для отдельных элементов управления идентификаторы, начинающиеся с префикса **ID_**.

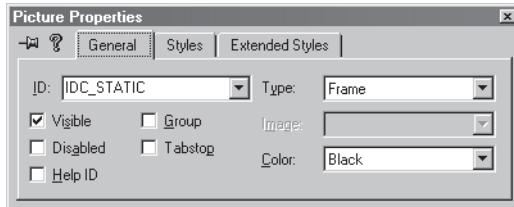


Рис. 7.6. Окно Picture Properties

Ведите в поле ID идентификатор IDC_STATIC_BFLY. В открывшемся списке Type выберите значение Bitmap. После этого станет доступным комбинированный список Image. В нем пока можно найти только один идентификатор IDB_BUTTERFLY, поскольку других растровых образов в файле описания ресурсов пока что нет. Именно этот идентификатор и нужно выбрать в списке. Значения остальных свойств на вкладке General оставьте такими, какими они применяются по умолчанию.

Стоит обратить внимание на включенную отметку флагка Visible. Она указывает, что элемент управления по умолчанию является *видимым*. В ходе выполнения приложения вы можете скрыть любой элемент управления при помощи функции ShowWindow, передав ей в первом параметре дескриптор элемента управления, а во втором параметре — константу SW_HIDE.

Теперь перейдите на вкладку Styles и включите отметку флагка Sunken, чтобы создать «вдавленную» рамку вокруг элемента управления. Значения остальных свойств на вкладке Styles следует оставить неизменными. Заметим, что если установить флагок Notify, то элемент управления будет посыпать родительскому окну нотификационные сообщения STN_CLICKED и STN_DBCLK, когда пользователь делает одинарный или двойной щелчок мышью на элементе управления. Закройте окно Picture Properties.

После этих действий элемент управления Picture будет отображать картинку с бабочкой, как показано на рис. 7.7. Но прямоугольной рамки элемента Static в этот момент еще нет.

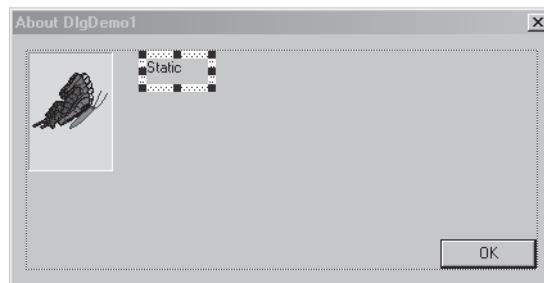


Рис. 7.7. Размещение на форме диалога элемента управления Static Text

Добавление элементов управления Надпись

Займемся информационными надписями. Щелкните мышью на кнопке Static Text и повторите щелчок на форме диалога правее от картинки. На форме появится прямоугольная рамка элемента Static, как показано на рис. 7.7.

Растяните мышью ограничивающий прямоугольник элемента **Static** так, чтобы его размеры позволяли ввести нужный текст. Двойным щелчком мыши вызовите окно свойств элемента управления (рис. 7.8).

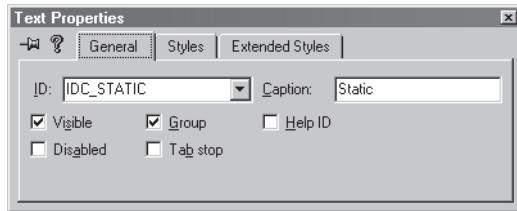


Рис. 7.8. Окно Text Properties

Редактор диалога присваивает по умолчанию всем элементам **Static Text** идентификатор **IDC_STATIC**. Если в ходе выполнения программы не требуется изменять атрибуты элемента управления (например, его местоположение или значение свойства **Caption**), то можно оставить значение идентификатора по умолчанию. В противном случае необходимо назначить элементу уникальный идентификатор.

Замените идентификатор в поле **ID** на значение **IDC_STATIC_1**, так как в ходе работы программы будут изменяться некоторые атрибуты этого элемента.

Поле **Caption** содержит текстовую строку, которая будет отображаться внутри ограничивающего прямоугольника элемента **Static Text**. В строке могут использоваться управляющие символы \t (табуляция) и \n (перевод строки).

Ведите в поле **Caption** текст **DlgDemo1**.

Теперь перейдите на вкладку **Styles** (рис. 7.9).

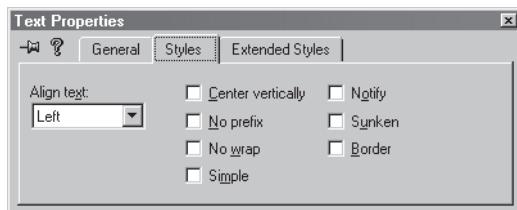


Рис. 7.9. Свойства элемента Static Text на вкладке Styles

Открывающийся список **Align text** позволяет задать выравнивание текста по горизонтали. Можно использовать значение **Left** (по умолчанию), **Center** или **Right**. Флажок **Center vertically** задает центрирование текста по вертикали. Способы выравнивания указываются по отношению к ограничивающему прямоугольнику.

Флажок **No prefix** влияет на интерпретацию символа &. Если этот флажок установлен, то символ & отображается. В противном случае символ & предшествует подчеркиваемому символу, что позволяет задавать «горячую» клавишу.

Флажок **Border** создает рамку вокруг элемента управления.

Выберите значение **Center** из списка **Align text** и установите флажки **Center vertically** и **Border**. Закройте окно **Text Properties**.

Теперь форма диалога примет вид, показанный на рис. 7.10.

Осталось добавить еще две информационные надписи. Для этого надо повторить описанные выше действия и поместить два элемента управления **Static Text**

ниже надписи DlgDemo1. Свойства этих элементов определите в соответствии со следующей таблицей:

ID	Caption	Align text	Установлены флажки
IDC_STATIC	Version 1.0	Center	Center vertically
IDC_STATIC	Copyright: Finesoft Corporation, 2005.	Center	Center vertically

После этого диалоговое окно примет примерно такой вид, какий показан на рис. 7.11.

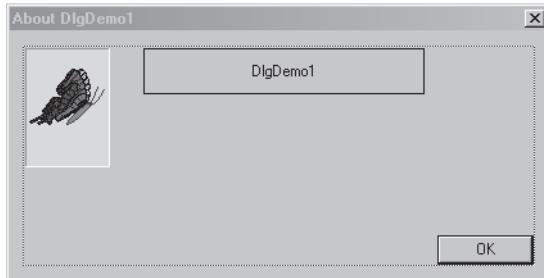


Рис. 7.10. Форма диалога после установки элемента IDC_STATIC_1

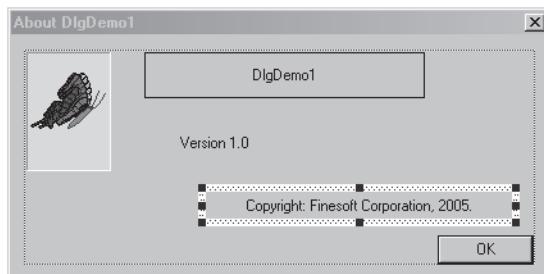


Рис. 7.11. Форма диалога после первичного размещения элементов управления

Но что-то в этой форме вызывает смутное раздражение. Попытаемся в этом разобраться...

Выравнивание элементов управления на форме диалога

В процессе размещения элементов управления часто возникает желание придать этому размещению эстетически приятный вид. Например, хорошо смотрится диалоговое окно, в котором однотипные элементы имеют одинаковые размеры, хотя бы по вертикали. Также окно радует глаз пользователя, если позиционирование некоторой группы элементов выровнено по положению одного из элементов.

Интегрированная среда Visual Studio содержит удобный инструмент для решения таких задач. Когда программист работает с редактором диалоговых окон, в главном меню Visual Studio появляется пункт *Layout*, выбор которого вызывает соответствующее подменю. Команды выравнивания из подменю *Layout* применяются к группе выделенных элементов.

Чтобы выделить группу элементов, нужно нажать клавишу **Ctrl** и, удерживая ее в нажатом состоянии, последовательно щелкать мышью на элементах, включаемых в группу. Следует помнить, что последний элемент, включаемый в группу, считается эталонным элементом. Именно по этому эталонному элементу будут подстраиваться характеристики остальных элементов в группе при выполнении некоторых команд из подменю **Layout**.

После выделения группы элементов вызывается нужная команда. Например, если необходимо сделать одинаковыми размеры элементов, то выбирается команда **Make Same Size**, а всплывающее каскадное меню предоставляет на выбор варианты **Width** (по ширине), **Height** (по высоте), **Both** (по ширине и высоте одновременно).

Для выравнивания позиционирования элементов выбирается команда **Align**. Чаще всего при работе с этой командой используются параметры **Left** (по левой границе) и **Right** (по правой границе).

Команда **Space Evenly** позволяет выровнять промежутки между элементами при размещении по горизонтали (опция **Across**) или при размещении по вертикали (опция **Down**). Команда **Center in Dialog** перемещает всю группу выделенных элементов, центрируя ее либо по горизонтали (опция **Horizontal**), либо по вертикали (опция **Vertical**).

Команда **Arrange Buttons** применяется только для кнопок и позволяет размещать группу кнопок либо по правому краю, либо по нижнему краю формы диалогового окна.

Воспользуемся инструментом **Layout** для улучшения внешнего вида созданного диалогового окна в приложении **DlgDemo1**.

Применим команду **Make Same Size**, чтобы сделать одинаковой ширину всех элементов **Static Text**, команду **Align** — для позиционирования всех элементов по левому краю надписи **DlgDemo1** и команду **Space Evenly** — для выравнивания промежутков между элементами по вертикали. На рис. 7.12 показан результат этих манипуляций.

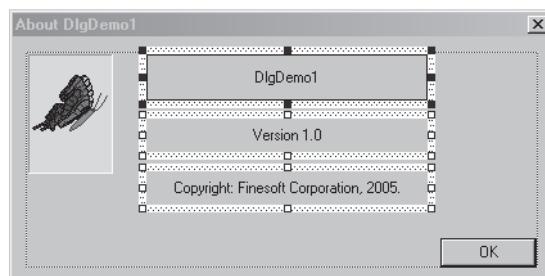


Рис. 7.12. Форма диалога после выравнивания элементов управления

Итак, мы завершили проектирование шаблона диалога.

Ради любопытства посмотрим на свойства кнопки **OK**, которую нам «подарил» редактор диалога. На рис. 7.13 показано окно свойств этого элемента управления, которое, как обычно, вызывается двойным щелчком мыши.

Обратите внимание на включенную отметку флагка **Tab stop**. Благодаря этому элемент управления получает стиль **WS_TABSTOP**, означающий, что кнопка **OK** может получать фокус ввода при работе с диалоговым окном через клавиатурный интерфейс. Этот способ общения пользователя с окном мы рассмотрим позже.



Рис. 7.13. Свойства кнопки OK

Если перейти на вкладку **Styles**, то можно увидеть, что на ней установлен флагок **Default button**. Он определяет данную кнопку как кнопку, применяемую по умолчанию. Это означает, что когда ни один из элементов управления не имеет фокуса ввода, то клавиатурный ввод переправляется именно этой кнопке.

Определение диалоговой процедуры и вызов диалога

Реализацию этих этапов рассмотрим на примере исходного кода приложения **DlgDemo1**.

Файл **DlgDemo1.cpp**, входящий в состав проекта, в настоящий момент является точной копией файла **MenuDemo1.cpp** из листинга 6.1. Текст данного файла нужно отредактировать так, чтобы он соответствовал листингу 7.1.

Листинг 7.1. Проект DlgDemo1

```
//////////  
// DlgDemo1.cpp  
#include <windows.h>  
#include <stdio.h>  
#include "KWnd.h"  
#include "resource.h"  
  
#define W 200 // ширина фигуры  
#define H 140 // высота фигуры  
enum ShapeSize { MAX, MIN }:  
  
typedef struct {  
    int id_shape; // идентификатор фигуры  
    BOOL fRed; // компонент красного цвета  
    BOOL fGreen; // компонент зеленого цвета  
    BOOL fBlue; // компонент синего цвета  
    int id_bright; // идентификатор яркости цвета  
} ShapeData;  
  
BOOL CALLBACK AboutDlgProc(HWND, UINT, WPARAM, LPARAM);  
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);  
=====  
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,  
    LPSTR lpCmdLine, int nCmdShow)  
{  
    MSG msg;  
    KWnd mainWnd("DlgDemo1", hInstance, nCmdShow, WndProc,  
        MAKEINTRESOURCE(IDR_MENU1), 100, 100, 400, 300);  
  
    while (GetMessage(&msg, NULL, 0, 0)) {  
        TranslateMessage(&msg);
```

```
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    static HINSTANCE hInst;
    /* Объявления остальных локальных переменных - из листинга 6.1 */

    switch (uMsg)
    {
    case WM_CREATE:
        hInst = GetModuleHandle(NULL);
        /* Остальной код обработки сообщения WM_CREATE - из листинга 6.1 */
        break;

    case WM_COMMAND:
        switch (LOWORD(wParam))
        {

        /*
         код обработки для команд IDM_OPEN, IDM_CLOSE, . . . , IDM_RESIZE -
         из листинга 6.1
        */

        case IDM_ABOUT:
            // Вызов диалога
            DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUT), hWnd, AboutDlgProc);

            break;

        default:
            break;
        }
        InvalidateRect(hWnd, NULL, TRUE);
        break;

    case WM_PAINT:
        /* Код обработки сообщения WM_PAINT из листинга 6.1 */
        break;

    case WM_DESTROY:
        PostQuitMessage(0);
        break;

    default:
        return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }
}

//=====
// Диалоговая процедура
```

Листинг 7.1 (продолжение)

```
BOOL CALLBACK AboutDlgProc(HWND hDlg, UINT uMsg, WPARAM wParam,
    LPARAM lParam)
{
    switch (uMsg) {

        case WM_INITDIALOG:
            return TRUE;

        case WM_COMMAND:
            switch (LOWORD(wParam)) {
                case IDOK:
                case IDCANCEL:
                    EndDialog(hDlg, 0);
                    return TRUE;
            }
            break;
    }
    return FALSE;
}
////////////////////////////////////////////////////////////////
```

Диалоговая процедура

Диалоговая процедура `AboutDlgProc` во многом напоминает оконную процедуру. Она должна иметь спецификатор `CALLBACK`, поскольку вызывается операционной системой. Имя функции может быть произвольным, однако сложилась традиция завершать это имя суффиксом `DlgProc`.

Функция `AboutDlgProc` принимает такой же набор параметров, что и обычная оконная процедура. Но некоторые различия между диалоговой процедурой и оконной процедурой все же есть:

- ❑ Оконная процедура возвращает значение типа `LRESULT`, а диалоговая процедура — значение типа `BOOL`.
- ❑ Если оконная процедура не обрабатывает какое-то сообщение, то она вызывает `DefWindowProc`. Если диалоговая процедура не обрабатывает какое-то сообщение, то она возвращает значение `FALSE`. Если же сообщение обрабатывается диалоговой процедурой, то она возвращает значение `TRUE`.
- ❑ Диалоговая процедура не обрабатывает сообщение `WM_CREATE`. Вместо этого она выполняет инициализацию при обработке специального сообщения `WM_INITDIALOG`.
- ❑ Диалоговая процедура обычно не обрабатывает сообщение `WM_PAINT`, так как все функции диалогового окна реализуются элементами управления.

Сообщение `WM_INITDIALOG` является первым сообщением, которое получает диалоговая процедура. Если после обработки этого сообщения процедура возвращает значение `TRUE`, то Windows помещает фокус ввода на первое дочернее окно элемента управления, которое имеет стиль `WS_TABSTOP`. В разрабатываемом диалоговом окне первым элементом управления со стилем `WS_TABSTOP` является кнопка `OK`. В то же время при обработке сообщения `WM_INITDIALOG` диалоговая процедура может использовать функцию `SetFocus` для того, чтобы установить фокус на какой-то другой элемент управления. Но тогда она должна вернуть значение `FALSE`.

Блок обработки сообщения `WM_INITDIALOG` является самым удобным местом для инициализации элементов управления, если в этом есть необходимость.

Основным сообщением, обрабатываемым в диалоговой процедуре, является **WM_COMMAND**. Напомним, что если источником сообщения **WM_COMMAND** является элемент управления, то младшее слово параметра **wParam** содержит идентификатор элемента управления, старшее слово **wParam** содержит код уведомления, а параметр **lParam** — дескриптор элемента управления.

В нашей программе диалоговая процедура должна обрабатывать только два сообщения **WM_COMMAND**. Источником первого сообщения является кнопка **OK** с идентификатором **IDOK**. Сообщение инициируется, когда пользователь щелкает на кнопке мышью или нажимает клавишу **Enter**.

Источником второго сообщения является кнопка закрытия диалогового окна, находящаяся в правой части его заголовка и имеющая идентификатор **IDCANCEL**. Это сообщение появляется, когда пользователь щелкает мышью на кнопке с крестиком или нажимает сочетание клавиш **Alt+F4**.

Обрабатывая оба эти сообщения, диалоговая процедура вызывает функцию **EndDialog**, после чего возвращает значение **TRUE**. Для всех остальных сообщений диалоговая процедура возвращает значение **FALSE**.

Функция **EndDialog** закрывает модальное диалоговое окно. Она имеет следующий прототип:

```
BOOL EndDialog (
    HWND hDlg,          // дескриптор диалогового окна
    INT_PTR nResult     // значение, возвращаемое из функции DialogBox
);
```

Второй параметр функции **EndDialog** задает значение, которое передается функции **DialogBox** (описываемой ниже) для использования в качестве кода возврата из функции **DialogBox**. Чаще всего функции **EndDialog** передается в параметре **nResult** значение **TRUE** при обработке команды **IDOK** и **FALSE** — при обработке команды **IDCANCEL**.

В программе **DlgDemo1** код возврата функции **DialogBox** игнорируется, но на самом деле он может быть использован для проверки выбора, сделанного пользователем.

Вызов диалога

Вызов диалогового окна в программе должен происходить после выбора пользователем пункта меню **About...**. Обработкой сообщений **WM_COMMAND**, которые инициируются после выбора команд меню, занимается оконная процедура. В случае выбора пункта меню **About...** младшее слово **wParam** сообщения **WM_COMMAND** содержит идентификатор **IDM_ABOUT**.

Оконная процедура программы **MenuDemo1**, обрабатывая это сообщение, вызывает функцию **MessageBox**. Теперь же оконная процедура вызывает функцию **DialogBox**:

```
DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUT), hWnd, AboutDlgProc);
```

Для вызова функции необходимо использовать дескриптор экземпляра приложения, сохраненный при обработке сообщения **WM_CREATE**, идентификатор шаблона диалога, дескриптор родительского окна для окна диалога и адрес диалоговой процедуры.

Функция **DialogBox** создает и выводит на экран модальное диалоговое окно, построенное по шаблону **IDD_ABOUT**. Кроме того, **DialogBox** сообщает Windows, что диалоговая процедура для обслуживания этого окна имеет адрес **AboutDlgProc**.

В своей реализации функция DialogBox использует вызов CreateWindowEx для создания диалогового окна. После этого она посыпает сообщение WM_INITDIALOG диалоговой процедуре, отображает диалоговое окно, блокирует (делает недоступным для ввода) родительское окно и запускает цикл обработки сообщений для менеджера диалогового окна.

Функция DialogBox не возвращает управление в WndProc до тех пор, пока окно диалога не будет закрыто.

Когда диалоговая процедура вызывает функцию EndDialog, функция DialogBox уничтожает диалоговое окно, завершает работу цикла обработки сообщений для менеджера диалогового окна, деблокирует родительское окно и возвращает значение параметра nResult, с которым вызывалась функция EndDialog.

Тестирование приложения

Откомпилируйте и запустите программу. Она должна функционировать так же, как и приложение MenuDemo1.

Выберите пункт меню About.... Если все было сделано правильно, то появится модальное диалоговое окно, внешний вид которого показан на рис. 7.14.

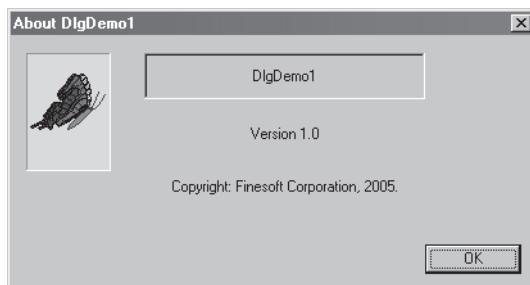


Рис. 7.14. Модальное диалоговое окно About DlgDemo1

Получилось неплохо. Во всяком случае, гораздо привлекательнее, чем обычное окно сообщений, которое отображается в программе MenuDemo1 при помощи функции MessageBox.

Но внутренний голос подсказывает, что дизайн диалогового окна можно сделать еще более симпатичным.... Вот если бы можно было написать имя приложения более крупным шрифтом? И не только более крупным шрифтом, но и цветными буквами?..

Для реализации столь смелой мечты продолжим наше путешествие по лабиринтам Win32 API.

Изменение атрибутов элемента управления

Во время работы приложения вы можете изменить некоторые из атрибутов элемента управления в диалоговом окне. Так, для изменения статуса элемента управления можно вызвать функцию EnableWindow, которая имеет следующий прототип:

```
BOOL EnableWindow(
    HWND hWnd, // дескриптор элемента управления
    BOOL bEnable // флаг разрешения или запрещения ввода от клавиатуры и мыши
);
```

Второй параметр функции задает новый статус элемента управления. Если он имеет значение **TRUE**, то элемент становится *разрешенным*. Если параметру передается значение **FALSE**, то элемент становится *запрещенным*.

Также вы можете изменить следующие атрибуты элемента управления:

- текст, связанный с элементом управления;
- используемый шрифт;
- цветовые атрибуты (только для элементов **Static Text** и **Edit Box**).

Текст (атрибут **Caption**)

Текст, связанный с элементом управления, можно изменить при помощи функции **SetWindowText**. Первому параметру функции передается дескриптор элемента управления, второму параметру — указатель на символьную строку или строковая константа.

Напомним, что дескриптор элемента управления **hWndControl** в диалоговом окне **hDlg** можно получить по его идентификатору **ID**, вызвав функцию **GetDlgItem**:

```
hWndControl = GetDlgItem(hDlg, ID);
```

Используемый шрифт

В элементах управления используется системный шрифт по умолчанию. Образец такого шрифта представлен на рис. 7.14.

Обычно надписи (**Static Text**) применяются в качестве текстовых меток для других элементов управления, и в этом случае системный шрифт хорошо сочетается с другими элементами диалогового окна. Но иногда все-таки желательно изменить шрифт элемента управления. Например, когда статический текст используется как информационная надпись.

Для изменения шрифта элемента управления необходимо послать сообщение **WM_SETFONT** окну этого элемента. Как известно, приложение может посыпать сообщения любому окну при помощи функции **SendMessage**. Интерпретация параметров функции **SendMessage** зависит от кода сообщения. В данном случае функция вызывается со следующими параметрами:

```
SendMessage ( // сообщение
    (HWND) hWnd, // дескриптор окна-получателя сообщения
    WM_SETFONT, // код сообщения
    (WPARAM) wParam; // дескриптор назначаемого шрифта
    (LPARAM) lParam; // опция перерисовки элемента управления
);
```

Если параметр **wParam** равен **NULL**, то элемент управления будет использовать системный шрифт по умолчанию. Значение **TRUE** для параметра **lParam** заставляет элемент управления немедленно перерисовать себя.

Цветовые атрибуты

Элементы управления **Static Text** и **Edit Box**, используемые в режиме «только для чтения» (**read-only**), посыпают своему родительскому окну сообщение **WM_CTLCOLORSTATIC** перед тем, как они должны быть перерисованы.

В этом сообщении параметр **wParam** содержит дескриптор контекста устройства элемента управления, а параметр **lParam** — дескриптор самого элемента управления. Для использования этих дескрипторов в программе значение **wParam** следует преобразовать к типу **HDC**, а значение **lParam** — к типу **HWND**.

Обрабатывая сообщение **WM_CTLCOLORSTATIC**, родительское окно может использовать полученный дескриптор контекста устройства, чтобы изменить цветовые атрибуты элемента управления. К таким атрибутам относятся цвет текста, цвет фона графических элементов и режим смещивания фона.

Если приложение обрабатывает сообщение **WM_CTLCOLORSTATIC**, то возвращаемым значением должен быть дескриптор кисти типа **HBRUSH**, который система использует для заполнения фона окна элемента управления.

Пример модификации атрибутов элемента управления

Применим новые знания на практике. Доработаем приложение **DlgDemo1**, чтобы улучшить дизайн модального диалогового окна. Для этого в листинге 7.1 нужно заменить определение функции **AboutDlgProc** следующим кодом:

```
//=====
// Диалоговая процедура
BOOL CALLBACK AboutDlgProc(HWND hDlg, UINT uMsg, WPARAM wParam,
    LPARAM lParam)
{
    static HWND hStatic1;
    static LOGFONT lf; // обязательно static для инициализации полей нулями!
    HFONT hFont1;

    switch (uMsg) {
        case WM_INITDIALOG:
            hStatic1 = GetDlgItem(hDlg, IDC_STATIC_1);

            // Создание логического шрифта Verdana
            lf.lfHeight = 28;
            lstrcpy( (LPSTR)&lf.lfFaceName, "Verdana" );
            hFont1 = CreateFontIndirect(&lf);

            // Модификация шрифта для элемента hStatic1
            SendMessage(hStatic1, WM_SETFONT, (WPARAM)hFont1, TRUE );
            return TRUE;

        case WM_CTLCOLORSTATIC:
            // Модификация цветовых атрибутов элемента hStatic1
            if ((HWND)lParam == hStatic1) {
                SetTextColor((HDC)wParam, RGB(160, 0, 0));
                SetBkMode((HDC)wParam, TRANSPARENT);
                return (DWORD)GetSysColorBrush(COLOR_3DFACE);
            }
            break;

        case WM_COMMAND:
            switch (LOWORD(wParam)) {
                case IDOK:
                case IDCANCEL:
                    DeleteObject(hFont1);
                    EndDialog(hDlg, 0);
                    return TRUE;
            }
            break;
    }
    return FALSE;
}
//////////
```

Модификация шрифта для элемента управления IDC_STATIC_1 осуществляется в блоке обработки сообщения WM_INITDIALOG, а модификация цветовых атрибутов — в блоке обработки сообщения WM_CTLCOLORSTATIC.

Если при обработке сообщения WM_CTLCOLORSTATIC выясняется, что оно поступило от элемента управления с дескриптором hStatic1, то программа устанавливает темно-красный цвет для вывода текста и возвращает дескриптор системной кисти, имеющей системный цвет COLOR_3DFACE. Возвращаемое значение позволяет сохранить обычный фон элемента управления Static Text, сливающийся с фоном диалогового окна. Если бы было необходимо сделать иным фон окна элемента управления, то нужно было бы создать кисть (HBRUSH) желаемого цвета и использовать для возврата ее дескриптор.

Модифицированное приложение DlgDemo1 будет выводить диалоговое окно About DlgDemo1, показанное на рис. 7.15.



Рис. 7.15. Модальное диалоговое окно приложения DlgDemo1

Использование других элементов управления

Приложение DlgDemo1 демонстрирует использование в диалоговом окне элементов управления Picture и Static Text. Эти элементы реализуются в системе как дочерние окна предопределенного оконного класса STATIC.

Следующая группа элементов управления, которая будет рассматриваться в этой главе, включает в себя кнопку (Button), флажок (Check box), переключатель (Radio button) и групповую рамку (Group box). Все эти элементы реализуются как дочерние окна предопределенного оконного класса BUTTON.

Кнопки

Кнопка (Button), называемая иногда «нажимаемой кнопкой» (*push button*), представляет собой прямоугольник, внутри которого обычно располагается некоторый текст. Щелчок мышью на кнопке заставляет ее перерисовать себя, используя стиль 3D с тенью, чтобы выглядеть «нажатой». Отпускание кнопки мыши восстанавливает начальный облик нажимаемой кнопки, а родительскому окну посыпается сообщение WM_COMMAND с кодом уведомления BN_CLICKED.

Windows заботится о надлежащем поведении и внешнем облике кнопок. Впрочем, то же самое можно сказать о флажках и переключателях, которые тоже являются особыми разновидностями кнопок. Если кнопка имеет фокус ввода, то текст

обводится штриховой линией, а нажатие и отпускание клавиши пробела имеет тот же эффект, что и щелчок мышью. Так утверждается в справочных материалах MSDN. К сожалению, *выделение текста штриховой линией* для кнопки с фокусом ввода четко срабатывает только при использовании клавиатурного интерфейса, когда перевод фокуса ввода осуществляется при помощи клавиши Tab. Когда же приложение отслеживает положение курсора мыши самостоятельно и вызывает функцию SetFocus для перевода фокуса ввода на соответствующий элемент, то никакого выделения текста почему-то не происходит. В приложении DlgDemo2, рассматриваемом ниже, показан вариант решения этой проблемы с помощью вызова функции DrawFocusRect.

Кнопки Button используются в основном для немедленного выполнения действия, без сохранения какой-либо индикации положения кнопки «включено/выключено».

В приложении DlgDemo1 уже использовалась кнопка OK, которая находилась в исходной форме диалога, предложенной редактором диалоговых окон. Но вы можете добавлять на форму диалога и другие кнопки, выполняющие те или иные функции пользовательского интерфейса.

Элемент управления «кнопка» размещается на форме диалога так же, как и другие элементы управления, — с помощью мыши, с предварительным выделением элемента Button на панели инструментов Controls. Затем надо вызвать окно свойств Push Button Properties и на вкладке General (см. рис. 7.13) в текстовом поле ID задать идентификатор кнопки, а в поле Caption указать текст, который будет отображаться на кнопке.

СОВЕТ

Редактор диалоговых окон предлагает в качестве идентификаторов для кнопок Button имена IDC_BUTTON1, IDC_BUTTON2 и им подобные. Рекомендуется заменять их именами, отражающими функциональное назначение кнопок. Этот совет относится и к определению идентификаторов других элементов управления.

На вкладке Styles (рис. 7.16) можно задать дополнительные свойства кнопки.

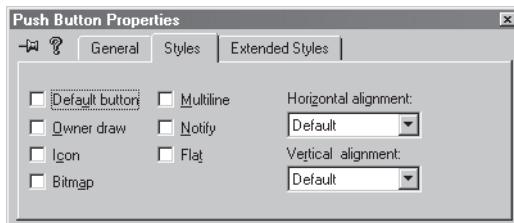


Рис. 7.16. Вкладка Styles окна свойств Push Button Properties

- Флажок Default button назначает кнопке атрибут «применяется по умолчанию».
- Флажок Owner draw применяется для создания кнопок, за прорисовку которых отвечает окно-владелец. Иногда кнопки этого стиля называют *кнопками, определяемыми программистом*.
- Флажки Icon или Bitmap позволяют указать, что вместо текста будет отображаться пиктограмма или растровый образ.

- ❑ Флажок **Multiline** используется, если текст слишком длинный, чтобы уместиться на кнопке в одну строку.
- ❑ Флажок **Flat** создает плоскую кнопку.
- ❑ Опции **Horizontal alignment** и **Vertical alignment** позволяют выбрать вариант выравнивания текста внутри ограничивающего прямоугольника.
- ❑ Чаще всего значения свойств на вкладке **Styles** оставляют по умолчанию, то есть в том состоянии, которое показано на рис. 7.16.

Наиболее сложны в реализации кнопки, определяемые программистом (**Owner draw**), поэтому этот вопрос стоит рассмотреть отдельно.

Кнопка **Owner draw**

Если при задании свойств кнопки был установлен флажок **Owner draw**, то в определении кнопки в файле описания ресурсов появится стиль **BS_OWNERDRAW**. Кнопка стиля **BS_OWNERDRAW** отличается от обычной кнопки тем, что при необходимости перерисовки она посылает своему родительскому окну сообщение **WM_DRAWITEM**. Это происходит при первоначальном создании кнопки, при ее нажатии или отпускании, при получении или потере фокуса ввода и во всех других случаях, когда требуется перерисовка элемента управления.

Когда приложение получает сообщение **WM_DRAWITEM**, параметр **wParam** этого сообщения содержит идентификатор элемента управления, а параметр **lParam** является указателем на структуру типа **DRAWITEMSTRUCT**. Сообщение **WM_DRAWITEM** может поступать не только от кнопок со стилем **Owner draw**, но и от элементов управления **Combo box**, **List box** и **List view**, если они имеют аналогичный стиль.

Структура **DRAWITEMSTRUCT** определена следующим образом:

```
typedef struct {
    UINT      CtlType;
    UINT      CtlID;
    UINT      itemID;
    UINT      itemAction;
    UINT      itemState;
    HWND      hwndItem;
    HDC       hDC;
    RECT     rcItem;
    ULONG_PTR itemData;
} DRAWITEMSTRUCT;
```

Поле **CtlType** позволяет указывать тип элемента управления. Возможные значения этого поля приведены в табл. 7.2.

Таблица 7.2. Возможные значения поля **CtlType**

Значение поля CtlType	Тип элемента управления
ODT_BUTTON	Кнопка со стилем Owner draw
ODT_COMBOBOX	Combo box со стилем Owner draw
ODT_LISTBOX	List box со стилем Owner draw
ODT_LISTVIEW	List control со стилем Owner draw
ODT_MENU	Пункт меню со стилем Owner draw
ODT_STATIC	Static control со стилем Owner draw
ODT_TAB	Tab control со стилем Owner draw

В поле `CtlID` передается идентификатор элемента управления. Оно не используется, если источником сообщения является пункт меню.

В поле `itemID` передается идентификатор пункта меню или индекс строки в элементе управления `Combo box` или `List box`.

Поле `itemAction` определяет тип действия, связанного с элементом управления. Его значением может быть одна из констант, указанных в табл. 7.3.

Таблица 7.3. Возможные значения поля `itemAction`

Значение <code>itemAction</code>	Интерпретация
<code>ODA_DRAWENTIRE</code>	Элемент управления требует перерисовки
<code>ODA_FOCUS</code>	Элемент управления получил или потерял фокус ввода. Текущее состояние можно определить, проверяя флаг <code>ODS_FOCUS</code> в поле <code>itemState</code>
<code>ODA_SELECT</code>	Элемент управления получил или потерял статус выбранного (нажатого) элемента. Текущее состояние можно определить, проверяя флаг <code>ODS_SELECTED</code> в поле <code>itemState</code>

Поле `itemState` определяет текущее визуальное состояние элемента управления в виде комбинации битовых флагов. Применение флагов `ODS_FOCUS` и `ODS_SELECTED` поясняется в табл. 7.3. Информацию о других флагах можно найти в справочных материалах MSDN.

Поле `hDC` содержит дескриптор контекста устройства для дочернего окна элемента управления. Этот дескриптор нужно использовать во всех функциях рисования.

Поле `rcItem` содержит размеры прямоугольника, ограничивающего элемент управления. Прямоугольник определен в контексте устройства с дескриптором `hDC`.

Обрабатывая сообщение `WM_DRAWITEM`, приложение должно вызывать функции рисования в контексте устройства `hDC`, чтобы обеспечить требуемый вид кнопки. Чаще всего для этого используются растровые образы или просто выводится необходимый текст.

Пример использования кнопки `Owner draw`

Технику применения кнопки со стилем `Owner draw` покажем на примере модификации приложения `DlgDemo1`. В новой версии программы, названной `DlgDemo2`, диалоговое окно `About...` должно стать еще более привлекательным за счет совмещения картинки и кнопки `OK`.

Создайте новый проект с именем `DlgDemo2`. Скопируйте из папки проекта `DlgDemo1` (см. листинг 7.1) в папку проекта `DlgDemo2` файлы с расширениями `.cpp`, `.h` и `.rc`, скорректировав их имена заменой `DlgDemo1` на `DlgDemo2`. Добавьте эти файлы в состав проекта. В файле `DlgDemo2.cpp` все вхождения подстроки `DlgDemo1` замените подстрокой `DlgDemo2`.

Скопируйте также в папку проекта файл `Butterfly.bmp`, заменив его имя на `BtnActive.bmp`. Этот растровый образ будет использоваться для изображения *активной кнопки*, то есть кнопки, имеющей фокус ввода. Создайте при помощи какого-либо графического редактора еще два варианта растрового изображения. Файл `BtnNoActive.bmp` должен содержать изображение *пассивной кнопки* в серой гамме цветов, а файл `BtnPress.bmp` будет хранить изображение *нажатой кнопки*.

с более темным фоном¹. Кнопка становится пассивной, когда она теряет фокус ввода. В нажатое состояние кнопка переходит либо при щелчке мышью, либо, если кнопка имеет фокус ввода, при нажатии клавиши пробела.

Доработку программы начнем с модификации шаблона диалога IDD_ABOUT. Для этого надо перейти на вкладку **Resource View** в окне **Workspace** и дважды щелкнуть мышью на элементе **IDD_ABOUT**. В результате будет вызван редактор диалоговых окон с шаблоном диалога, внешний вид которого показан на рис. 7.12.

Откройте окно свойств диалогового окна **IDD_ABOUT** и замените текст **About DlgDemo1** строкой **About DlgDemo2**. Затем откройте окно свойств элемента **IDC_STATIC_1** и замените текст **DlgDemo1** строкой **DlgDemo2**.

Удалите кнопку **OK** и рисунок с бабочкой. Выделите группу надписей, как показано на рис. 7.12, и, используя клавиши-стрелки, сместите эту группу немного правее. Увеличьте высоту элемента **DlgDemo2**, а остальные две надписи передвиньте к нижней границе окна.

На месте удаленного рисунка поместите элемент управления **Button** и задайте его размеры, как показано на рис. 7.17. Теперь вызовите окно свойств кнопки и укажите идентификатор **IDOK**, а в поле **Caption** удалите весь текст. Затем перейдите на вкладку **Styles** и установите флажок **Owner draw**.

Ниже кнопки **IDOK** поместите еще одну кнопку стандартного размера. В окне свойств кнопки задайте идентификатор **ID_ABOUT_HELP**, а в поле **Caption** введите строку **Help**. После всех указанных манипуляций шаблон диалога примет вид, показанный на рис. 7.17.

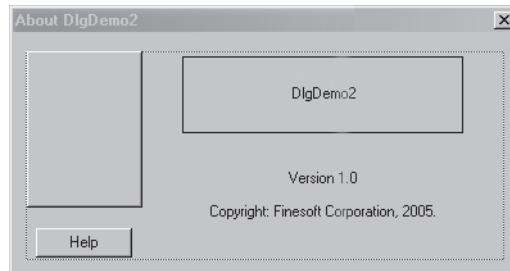


Рис. 7.17. Модифицированный шаблон диалога

Теперь обеспечим необходимые растревые ресурсы для программы. Перейдите опять на вкладку **Resource View** в окне **Workspace**. Откройте папку **Bitmap** и удалите элемент **IDB_BUTTERFLY**. Добавьте к проекту три растревых ресурса в режиме **Import**, назначив им следующие идентификаторы:

Идентификатор	Файл
IDB_PICT_0	BtnNoActive.bmp
IDB_PICT_1	BtnActive.bmp
IDB_PICT_2	BtnPress.bmp

¹ Вы можете взять готовые файлы с этими изображениями, если скачаете файлы к данной книге с сайта издательства <http://www.piter.com>.

Осталось внести необходимые изменения в код программы. Нет нужды приводить полный текст файла DlgDemo2.cpp, поскольку изменения касаются только функции AboutDlgProc. В листинге 7.2 содержится ее новая редакция.

Листинг 7.2. Диалоговая процедура AboutDlgProc в файле DlgDemo2.cpp

```
//=====
BOOL CALLBACK AboutDlgProc(HWND hDlg, UINT uMsg, WPARAM wParam,
    LPARAM lParam)
{
    HINSTANCE hInst;
    static HWND hStatic1;
    static LOGFONT lf; // обязательно static для инициализации полей нулями!
    static HFONT hFont1, hFont2;

    static HBITMAP hPict0;    // изображение пассивной кнопки
    static HBITMAP hPict1;    // изображение активной кнопки
    static HBITMAP hPict2;    // изображение нажатой кнопки
    static HBRUSH hBrush0;   // фон пассивной кнопки
    static HBRUSH hBrush1;   // фон активной кнопки
    static HBRUSH hBrush2;   // фон нажатой кнопки
    static HBRUSH hCurBrush; // текущий фон
    static HDC hMemDC;      // контекст устройства в памяти
    static BITMAP bm;        // параметры раstra

    LPDRAWITEMSTRUCT pdis;
    static HWND hBtnOk, hBtnHelp;
    static RECT rcBtnHelp, rect;
    int x0, y0 = 0;

    switch (uMsg) {
        case WM_INITDIALOG:
            hInst = GetModuleHandle(NULL);
            hStatic1 = GetDlgItem(hDlg, IDC_STATIC_1);
            hBtnOk = GetDlgItem(hDlg, IDOK);
            hBtnHelp = GetDlgItem(hDlg, ID_ABOUT_HELP);
            GetWindowRect(hBtnHelp, &rcBtnHelp);

            // Создание логического шрифта Verdana
            lf.lfHeight = 28;
            lstrcpy( (LPSTR)&lf.lfFaceName, "Verdana" );
            hFont1 = CreateFontIndirect(&lf);
            lf.lfHeight = 20;
            hFont2 = CreateFontIndirect(&lf);
            // Модификация шрифта для элемента hStatic1
            SendMessage(hStatic1, WM_SETFONT, (WPARAM)hFont1, TRUE );

            // Инициализация растровых изображений кнопки IDOK
            hPict0 = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_PICT_0));
            hPict1 = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_PICT_1));
            hPict2 = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_PICT_2));
            hBrush0 = CreateSolidBrush(RGB(214, 214, 214));
            hBrush1 = CreateSolidBrush(RGB(149, 242, 242));
            hBrush2 = CreateSolidBrush(RGB(83, 52, 254));

            GetObject(hPict0, sizeof(bm), (LPSTR)&bm);
            SetFocus(hBtnHelp);
```

```
return FALSE; // чтобы фокус ввода остался на hBtnHelp

case WM_CTLCOLORSTATIC:
// Модификация цветовых атрибутов элемента hStatic1
if ((HWND)lParam == hStatic1) {
    SetTextColor((HDC)wParam, RGB(160, 0, 0));
    SetBkMode((HDC)wParam, TRANSPARENT);
    return (DWORD)GetSysColorBrush(COLOR_3DFACE);
}
break;

case WM_DRAWITEM: // сообщение, используемое для
// перерисовки кнопки Owner draw
pdis = (LPDRAWITEMSTRUCT)lParam;
hMemDC = CreateCompatibleDC(pdis->hDC);
// Проверяем, какие события произошли
switch (pdis->itemAction) {
case ODA_FOCUS:
    if (pdis->itemState & ODS_FOCUS) {
        SelectObject(hMemDC, hPict1);
        hCurBrush = hBrush1;
    }
    else {
        SelectObject(hMemDC, hPict0);
        hCurBrush = hBrush0;
    }
break;

case ODA_SELECT:
    if (pdis->itemState & ODS_SELECTED) {
        SelectObject(hMemDC, hPict2);
        hCurBrush = hBrush2;
    }
break;

default:
    SelectObject(hMemDC, hPict0);
    hCurBrush = hBrush0;
} // end of switch

// Вывод изображения кнопки с учетом произошедших событий
FillRect(pdis->hDC, &pdis->rcItem, hCurBrush);
x0 = ((pdis->rcItem.right - pdis->rcItem.left) - bm.bmWidth) / 2;
// Картина ...
BitBlt(pdis->hDC, x0, y0, bm.bmWidth, bm.bmHeight,
       hMemDC, 0, 0, SRCCOPY);
// а теперь текст
SelectObject(pdis->hDC, hFont2);
SetBkMode(pdis->hDC, TRANSPARENT);
SetTextColor(pdis->hDC, RGB(255, 255, 255));
TextOut(pdis->hDC, 32, 74, "OK", 2);
SetTextColor(pdis->hDC, RGB(100, 100, 100));
TextOut(pdis->hDC, 30, 72, "OK", 2);
// рамка вокруг кнопки
FrameRect(pdis->hDC, &pdis->rcItem,
           (HBRUSH)GetStockObject(BLACK_BRUSH));
DeleteDC(hMemDC); // !!! не забудьте удалить hMemDC
```

продолжение ↗

Листинг 7.2 (продолжение)

```

    return TRUE;

case WM_SETCURSOR: // отслеживание фокуса ввода для курсора мыши
    if ((HWND)wParam == hBtnOk)
        if (GetFocus() != hBtnOk) SetFocus(hBtnOk);

    if ((HWND)wParam == hBtnHelp) {
        if (GetFocus() != hBtnHelp) {
            SetFocus(hBtnHelp);
            // Рисуем пунктирную рамку вызовом DrawFocusRect
            SetRect(&rect, 3, 3, rcBtnHelp.right - rcBtnHelp.left - 3,
                    rcBtnHelp.bottom - rcBtnHelp.top - 3);
            DrawFocusRect(GetDC(hBtnHelp), &rect);
        }
    }
    return FALSE;

case WM_COMMAND:
    switch (LOWORD(wParam)) {
        case ID_ABOUT_HELP:
            MessageBox(hDlg, "В этом диалоговом окне работает кнопка
                \"Owner draw\", \"HELP\", MB_OK");
            return TRUE;

        case IDOK:
        case IDCANCEL:
            DeleteObject(hFont1);
            DeleteObject(hFont2);
            EndDialog(hDlg, 0);
            return TRUE;
    }
    break;
}

return FALSE;
}
/////////////////////////////////////////////////////////////////

```

Обратите внимание на обработку сообщений WM_DRAWITEM и WM_SETCURSOR.

При обработке сообщения WM_DRAWITEM на поверхности кнопки IDOK отображается растровое изображение, соответствующее текущему состоянию кнопки. Для этого в контекст устройства в памяти hMemDC выбирается один из дескрипторов растров, hPict0, hPict1 или hPict2. Отображение осуществляется позже вызовом функции BitBlt. Но сначала поверхность кнопки заполняется цветом фона при помощи функции FillRect с использованием текущей кисти hCurBrush. Цвет кисти должен совпадать с цветом фона используемой картинки. В этом случае картинка плавно вписывается в поверхность кнопки, и пользователь не заметит, что рисунок меньше, чем кнопка. После отображения растра выводится текст OK с применением более красивого шрифта, чем системный шрифт.

В приложении приходится решать еще одну проблему. Дело в том, что кнопка IDOK должна изменять свой облик в зависимости от того, имеет она фокус ввода или нет. Если пользователь общается с приложением через клавиатурный интерфейс, то никаких проблем с управлением фокусом ввода не будет. При нажатии клавиши Tab фокус автоматически переходит от одного элемента управления со стилем WS_TABSTOP к другому элементу, имеющему такой же стиль.

Хотелось бы сделать так, чтобы при использовании мыши фокус ввода также переходил автоматически с одной кнопки на другую, как только курсор мыши наводится на эту кнопку. К сожалению, Windows не обеспечивает такого мышного интерфейса, поэтому в программе эта проблема решается при помощи обработки сообщения WM_SETCURSOR.

Сообщение WM_SETCURSOR посыпается активному окну приложения при любом перемещении курсора мыши. В нашей программе положение курсора мыши отслеживается только при появлении на экране диалогового окна IDD_ABOUT. Когда функция AboutDlgProc получает сообщение WM_SETCURSOR, его параметр wParam содержит дескриптор того окна, в котором в данный момент находится курсор. Так как дескрипторы кнопок hBtnOk и hBtnHelp известны, то не составляет труда зафиксировать момент попадания курсора на ту или иную кнопку и вызвать функцию SetFocus, если кнопка в этот момент не имеет фокуса ввода.

Для обычных кнопок, не имеющих стиля BS_OWNERDRAW, при получении фокуса ввода вызывается функция DrawFocusRect, чтобы нарисовать штриховую рамку вокруг текста кнопки.

На рис. 7.18 показан вид диалогового окна About DlgDemo2 с пассивным состоянием кнопки OK.

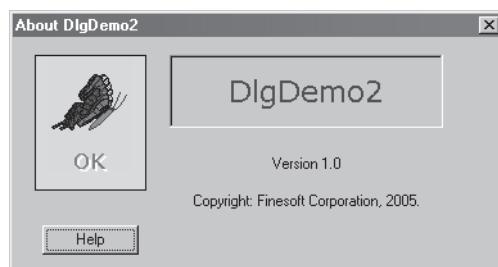


Рис. 7.18. Диалоговое окно About DlgDemo2

Если вы хотите увидеть кнопку OK во всех трех состояниях, да еще и в цветном исполнении, вам придется откомпилировать проект DlgDemo2 и запустить программу на вашем компьютере.

Флажки

Флажок (Check box) представляет собой маленькое квадратное окно с сопроводительным текстом, который обычно размещается справа от этого окна. Флажок действует как двухпозиционный переключатель. Один щелчок вызывает появление контрольной отметки (галочки), а другой щелчок приводит к ее исчезновению. Соответствующие состояния элемента управления определяют также с помощью терминов «флажок установлен» и «флажок сброшен».

Этот элемент управления уже не раз использовался при работе со средой Visual Studio. Например, свойства элементов управления в соответствующих окнах задавались как раз с помощью флажков.

Сам флажок тоже обладает набором свойств, и для их определения вызывается окно свойств Check Box Properties. На рис. 7.19 показана вкладка Styles в этом окне.

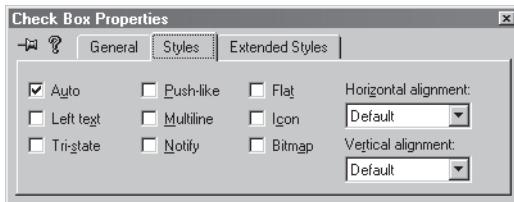


Рис. 7.19. Вкладка Styles в окне свойств флажка

Флажок Auto установлен по умолчанию. Это очень важное свойство. Оно заставляет элемент управления отслеживать все щелчки мышью, и при этом элемент управления сам включает или выключает контрольную отметку. Поэтому вы можете об этом не беспокоиться. Если же выключить свойство Auto, то программа должна при обработке сообщения WM_COMMAND, поступающего от флажка, вызывать функцию CheckDlgButton для установки или снятия отметки.

Флажок Left text ставится, если вы хотите поместить сопровождающий текст слева от квадратного окошка.

Свойство Tri-state используется для создания флажка, имеющего три состояния. Кроме состояний «установлен» и «сброшен» добавляется «неопределенное состояние», в котором флажок отображен в серой гамме. Серый цвет показывает пользователю, что выбор флажка не определен или не имеет отношения к текущей операции.

Свойство Push-like изменяет внешний вид флажка так, что он выглядит как нажимаемая кнопка. Вместо установки галочки эта кнопка переходит в нажатое состояние и остается в нем до следующего щелчка мышью.

Флажок Flat применяется, если нужно использовать плоское изображение квадратного окошка.

Флажок Icon или Bitmap используется, чтобы вместо текста рядом с окошком поместить какой-нибудь рисунок. Но так делают очень редко, поэтому здесь эти параметры не будут рассматриваться детально.

Несмотря на то что с включенным свойством Auto флажок сам управляет установкой и снятием отметки, все же при первом появлении диалогового окна на экране иногда требуется программная инициализация состояния элемента управления. Для этого используется функция CheckDlgButton:

```
BOOL CheckDlgButton (
    HWND hDlg,           // дескриптор диалогового окна
    int nIDButton,        // идентификатор кнопки
    UINT uCheck          // состояние отметки
);
```

Параметр uCheck может принимать одно из значений, указанных в табл. 7.4.

Таблица 7.4. Возможные значения параметра uCheck

Константа	Шестнадцатеричное значение	Интерпретация
BST_UNCHECKED	0x0000	Снять отметку
BST_CHECKED	0x0001	Установить отметку
BST_INDETERMINATE	0x0002	Установить неопределенное состояние (state to grayed). Используется только для кнопок, имеющих свойство Tri-state

В окнах диалога элементы управления «флажок» используются обычно для выбора независимых параметров.

Переключатели

Переключатель (Radio button) похож на флажок, но имеет круглую форму, а не квадратную. Жирная точка внутри кружка показывает, что переключатель включен.

В диалоговых окнах группы переключателей, как правило, используются для выбора взаимоисключающих опций. В отличие от флажков, если повторно щелкнуть на переключателе, его состояние не изменится.

Устанавливая свойства переключателя в окне свойств Radio Button Properties, проследите, чтобы остался установленным флажок Auto, обеспечивающий автоматическую установку и снятие отметки для переключателя. Если флажок Auto снят, то программа должна при обработке сообщения WM_COMMAND, поступающего от переключателя, вызвать функцию CheckRadioButton для установки или снятия отметки.

Кроме того, для первого переключателя в группе связанных взаимоисключающих переключателей нужно обязательно установить флажок Group на вкладке General. Все последующие переключатели (в файле описания ресурсов) со сброшенным флажком Group считаются принадлежащими к этой группе. Если в последовательности описаний элементов управления встречается переключатель с установленным флажком Group, считается, что он начинает новую группу элементов Radio button.

Для группировки флажков или переключателей важен порядок их описания в файле описания ресурсов. Не забывайте, что определения элементов заносятся в файл описания ресурсов по мере размещения их на форме диалога. Впрочем, среда Visual Studio позволяет легко изменить этот порядок (этот вопрос рассматривается ниже в разделе «Клавиатурный интерфейс и порядок обхода элементов управления»).

Хотя с установленным флажком Auto переключатель сам управляет установкой и снятием отметки, при первом появлении диалогового окна может потребоваться программная инициализация состояния переключателя с помощью функции CheckRadioButton:

```
BOOL CheckRadioButton (
    HWND hDlg,           // дескриптор диалогового окна
    int nIDFirstButton,  // идентификатор первой кнопки в группе
    int nIDLastButton,   // идентификатор последней кнопки в группе
    int nIDCheckButton  // идентификатор выбранной кнопки
);
```

Групповая рамка

Групповая рамка (Group box) — это прямоугольная рамка с надписью, внутри которой помещается некоторый набор элементов управления.

Групповая рамка является исключением среди элементов класса BUTTON (см. табл. 7.1). Она не обрабатывает ни сообщения от клавиатуры, ни сообщения от мыши. Она не посыпает своему родительскому окну сообщение WM_COMMAND. В основном групповая рамка применяется как элемент оформления интерфейса, благодаря которому он становится более удобным и дружественным для пользователя.

Часто групповая рамка используется для объединения в одну группу нескольких флажков или переключателей. Причем в первом случае состояния флажков в группе не зависят друг от друга. Состояния переключателей, объединенных в группу, взаимозависимы, и только один из них может иметь включенное состояние.

Обращаем ваше внимание на то, что *одно лишь размещение* нескольких переключателей в групповой рамке не обеспечит их надлежащее поведение. Рамка является сугубо визуальным элементом. Чтобы переключатели составляли группу, первый из них должен иметь установленными флашки *Group* и *Tab stop* (на вкладке *General* окна свойств элемента управления), а у всех остальных переключателей в группе эти опции должны быть сброшены.

Пример использования групповой рамки, флажков и переключателей

Рассмотрим применение групповой рамки, флажков и переключателей на примере разработки приложения *DlgDemo3*, являющегося модификацией приложения *DlgDemo1*.

Цель модификации — сделать более удобным интерфейс пользователя в этой программе. Дело в том, что для выбора вида фигуры и задания ее цвета пользователь приложения *DlgDemo1* должен трижды выбирать пункт меню *DrawShape*. Первый раз — с командой *Shape* для выбора вида фигуры, второй раз — с командой *Color* для установки флашек *Red*, *Green*, *Blue* и третий раз — с командой *Color* для выбора одного из переключателей, *Dark*, *Medium*, *Light*. Было бы, наверное, удобнее, если бы по команде *DrawShape* вызывалось диалоговое окно, содержащее группы опций для установки всех атрибутов рисования.

Сказано — сделано. Создайте новый проект с именем *DlgDemo3*. Скопируйте из папки проекта *DlgDemo1* в папку проекта *DlgDemo3* файлы с расширениями *.cpp*, *.h* и *.rc*, скорректировав их имена заменой *DlgDemo1* на *DlgDemo3*. Включите эти файлы в состав проекта. Скопируйте также файл *Butterfly.bmp*.

Модификация меню в приложении *DlgDemo3*

В окне *Workspace* перейдите на вкладку *ResourceView*, откройте в иерархическом списке папку *Menu* и щелчком на элементе *IDR_MENU1* вызовите редактор меню. Откройте окно свойств *Menu Item Properties* для пункта *Draw shape*. В этом окне нужно сбросить флашок *Pop-up*. После щелчка мышью на нем будет отображено окно с предупреждением, что это действие уничтожит все пункты подменю *Draw shape* (рис. 7.20).

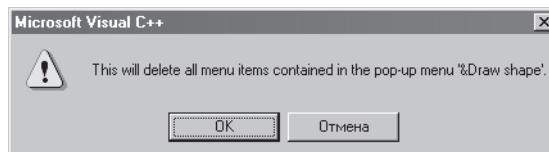


Рис. 7.20. Предупреждение редактора меню при попытке снять флашок *Pop-up*

Подтвердите серьезность ваших намерений, нажав кнопку *OK*. Окно исчезнет, а текстовое поле *ID* станет доступным для ввода информации. Введите в него идентификатор *IDM_DRAW_SHAPE*. После этого закройте окно свойств пункта *Draw Shape*.

Разработка шаблона диалога

Вы уже умеете делать это. Создайте заготовку шаблона нового диалогового окна. Для этого выполните команду **Insert ▶ Resource**, укажите тип ресурса **Dialog** и нажмите кнопку **New**. Затем вызовите окно свойств диалога, дважды щелкнув мышью на форме диалога. В текстовом поле **ID** укажите идентификатор **IDD_SHAPE_PARAM**. Переийдите на вкладку **Styles** и отключите отметку флажка **Title bar**, после чего закройте окно свойств диалога.

Увеличьте примерно вдвое размер формы по вертикали. Выделите группу из двух элементов, щелкнув на кнопках **OK** и **Cancel**. Применив команду **Layout ▶ Arrange Buttons ▶ Bottom**, переместите эти кнопки на нижнюю границу формы.

Теперь разместите на форме три групповые рамки. Внутри первой рамки нужно расположить три переключателя, внутри второй рамки — три флажка, внутри третьей рамки — снова три переключателя, как показано на рис. 7.21.

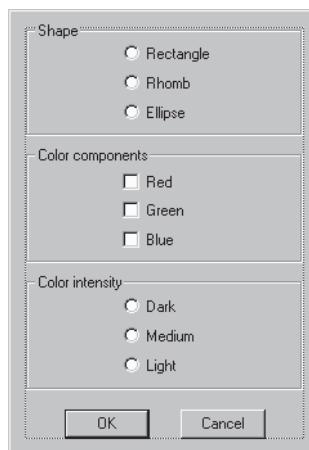


Рис. 7.21. Размещение элементов управления на форме диалога IDD_SHAPE_PARAM

В процессе размещения задайте свойства элементов управления в соответствии с табл. 7.5. Рекомендации по определению состояния флажков **Tab stop** и **Group** приводятся ниже в разделе «Клавиатурный интерфейс и порядок обхода элементов управления».

Таблица 7.5. Свойства элементов управления на форме диалога IDD_SHAPE_PARAM

Тип элемента	ID	Caption	Флажок Tab stop	Флажок Group
Group box	IDC_STATIC	Shape	Сброшен	Сброшен
Radio button	ID_RECTANGLE	Rectangle	Установлен	Установлен
Radio button	ID_RHOMB	Rhomb	Сброшен	Сброшен
Radio button	ID_ELLIPSE	Ellipse	Сброшен	Сброшен
Group box	IDC_STATIC	Color components	Сброшен	Сброшен
Check box	ID_RED	Red	Установлен	Сброшен
Check box	ID_GREEN	Green	Сброшен	Сброшен
Check box	ID_BLUE	Blue	Сброшен	Сброшен
Group box	IDC_STATIC	Color intensity	Сброшен	Сброшен

продолжение ➔

Таблица 7.5 (продолжение)

Тип элемента	ID	Caption	Флажок Tab stop	Флажок Group
Radio button	ID_DARK	Dark	Установлен	Установлен
Radio button	ID_MEDIUM	Medium	Сброшен	Сброшен
Radio button	ID_LIGHT	Light	Сброшен	Сброшен

Сохраните подготовленный шаблон в файле описания ресурсов при помощи кнопки Save на панели инструментов Visual Studio.

Коррекция шаблона диалога IDD_ABOUT

В окне Workspace перейдите на вкладку ResourceView и откройте шаблон диалога IDD_ABOUT. После этого откройте диалоговое окно Dialog Properties и в поле Caption введите текст About DlgDemo3. Затем нужно открыть окно свойств Text Properties для элемента управления IDC_STATIC_1 и заменить текст DlgDemo1 на строку DlgDemo3.

Модификация исходного кода

Отредактируйте код файла DlgDemo3.cpp так, чтобы он соответствовал листингу 7.3.

Листинг 7.3. Проект DlgDemo3

```
//////////  
// DlgDemo3.cpp  
#include <windows.h>  
#include <stdio.h>  
#include "KWnd.h"  
#include "resource.h"  
  
#define W 200 // ширина фигуры  
#define H 140 // высота фигуры  
  
enum ShapeSize { MAX, MIN };  
  
typedef struct {  
    int id_shape; // идентификатор фигуры  
    BOOL fRed; // компонент красного цвета  
    BOOL fGreen; // компонент зеленого цвета  
    BOOL fBlue; // компонент синего цвета  
    int id_bright; // идентификатор яркости цвета  
} ShapeData;  
  
ShapeData shapeData; // Глобальная структура для обмена данными  
// между ShapeParamDlgProc и WndProc  
  
BOOL CALLBACK ShapeParamDlgProc(HWND, UINT, WPARAM, LPARAM);  
BOOL CALLBACK AboutDlgProc(HWND, UINT, WPARAM, LPARAM);  
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);  
=====  
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,  
    LPSTR lpCmdLine, int nCmdShow)  
{  
    MSG msg;  
    KWnd mainWnd("DlgDemo3", hInstance, nCmdShow, WndProc,  
        MAKEINTRESOURCE(IDR_MENU1), 100, 100, 400, 300);  
  
    while (GetMessage(&msg, NULL, 0, 0)) {
```

```
TranslateMessage(&msg);
DispatchMessage(&msg);
}
return msg.wParam;
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    static HINSTANCE hInst;
    HDC hDC;
    PAINTSTRUCT ps;
    RECT rect;
    static HMENU hMenu; // дескриптор главного меню
    int x0, y0, x1, y1, x2, y2;
    POINT pt[4];
    static ShapeSize shapeSize = MIN;
    static BOOL bShow = TRUE;
    static HBRUSH hBrush, hOldBrush;
    char* itemResizeName[2] = { "Decrease!", "Increase!" };
    int intensity[3] = { 85, 170, 255 }; // интенсивность RGB-компонентов цвета
    int brightness;

    switch (uMsg)
    {
        case WM_CREATE:
            hInst = GetModuleHandle(NULL);
            hMenu = GetMenu(hWnd);
            SetMenuItemDefaultItem(GetSubMenu(hMenu, 0), IDM_OPEN, FALSE);
            CheckMenuItemRadioItem(GetSubMenu(hMenu, 1), IDM_SHOW_SHAPE,
                IDM_HIDE_SHAPE, IDM_SHOW_SHAPE, MF_BYCOMMAND);
            shapeData.id_shape = ID_RECTANGLE;
            shapeData.id_bright = ID_DARK;
            break;

        case WM_COMMAND:
            switch (LOWORD(wParam))
            {
/*                
    код обработки для команд IDM_OPEN, IDM_CLOSE, . . . , IDM_RESIZE -
    из листинга 6.1
*/
            }

        case IDM_ABOUT:
            DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUT), hWnd, AboutDlgProc);
            break;

        case IDM_DRAW_SHAPE:
            DialogBox(hInst, MAKEINTRESOURCE(IDD_SHAPE_PARAM),
                hWnd, ShapeParamDlgProc);
            break;

        default:
            break;
    }
}
```

продолжение ↗

Листинг 7.3 (продолжение)

```
InvalidateRect(hWnd, NULL, TRUE);
break;

case WM_PAINT:
    /* Код обработки сообщения WM_PAINT из листинга 6.1 */
    break;

case WM_DESTROY:
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}

return 0;
}

//=====
BOOL CALLBACK AboutDlgProc(HWND hDlg, UINT uMsg, WPARAM wParam,
    LPARAM lParam)
{
    /* Текст функции - из листинга 7.1 (модифицированный вариант) */
}

//=====
BOOL CALLBACK ShapeParamDlgProc(HWND hDlg, UINT uMsg, WPARAM wParam,
    LPARAM lParam)
{
    static ShapeData shapeDataNew;

    switch (uMsg) {
    case WM_INITDIALOG:
        CheckRadioButton(hDlg, ID_RECTANGLE, ID_ELLIPSE, shapeData.id_shape);
        CheckRadioButton(hDlg, ID_DARK, ID_LIGHT, shapeData.id_bright);
        CheckDlgButton(hDlg, ID_RED, shapeData.fRed);
        CheckDlgButton(hDlg, ID_GREEN, shapeData.fGreen);
        CheckDlgButton(hDlg, ID_BLUE, shapeData.fBlue);
        shapeDataNew = shapeData;
        return TRUE;

    case WM_COMMAND:
        switch (LOWORD(wParam)) {
        case ID_RECTANGLE:
        case ID_RHOMB:
        case ID_ELLIPSE:
            shapeDataNew.id_shape = LOWORD(wParam);
            return TRUE;

        case ID_RED:
            shapeDataNew.fRed = ~shapeDataNew.fRed;
            return TRUE;
        case ID_GREEN:
            shapeDataNew.fGreen = ~shapeDataNew.fGreen;
        }
    }
}
```

```
    return TRUE;
case ID_BLUE:
    shapeDataNew.fBlue = ~shapeDataNew.fBlue;
    return TRUE;

case ID_DARK:
case ID_MEDIUM:
case ID_LIGHT:
    shapeDataNew.id_bright = LOWORD(wParam);
    return TRUE;

case IDOK:
    shapeData = shapeDataNew;
    EndDialog(hDlg, TRUE);
    return TRUE;

case IDCANCEL:
    EndDialog(hDlg, FALSE);
    return TRUE;
}
break;
}
return FALSE;
}
////////////////////////////////////////////////////////////////
```

Конечно, следует обратить внимание на изменения, появившиеся в коде программы. Так, структурная переменная `shapeData` теперь стала глобальной, хотя раньше она объявлялась в теле функции `WndProc`. Это необходимо для обмена данными между `WndProc` и диалоговой процедурой `ShapeParamDlgProc`. Поля переменной `shapeData` хранят текущие параметры рисуемой фигуры. В момент запуска программы поля глобальной структуры инициализируются нулями.

Также появилась новая функция `ShapeParamDlgProc`, которая является диалоговой процедурой для окна `IDD_SHAPE_PARAM`. Обрабатывая сообщение `WM_INITDIALOG`, эта функция инициализирует элементы управления диалогового окна. Ранее указывалось, что если для флажка или переключателя установлено свойство `Auto`, то эти элементы сами управляют установкой и снятием своих меток, реагируя на щелчки пользователя. Но при первом появлении диалогового окна на экране программа должна позаботиться об установке меток в соответствии с текущими параметрами фигуры, сохраняемыми в переменной `shapeData`. Для этого вызываются функции `CheckRadioButton` и `CheckDlgButton`.

Завершив инициализацию, функция `ShapeParamDlgProc` копирует значение переменной `shapeData` в локальную переменную `shapeDataNew`. В полях переменной `shapeDataNew` записываются все изменения состояний флажков и переключателей в блоке обработки сообщения `WM_COMMAND`. Если диалоговое окно закрывается щелчком на кнопке `OK` с идентификатором `IDOK`, то перед его закрытием переменной `shapeData` присваивается значение `shapeDataNew`. Если диалоговое окно закрывается щелчком на кнопке `Cancel` с идентификатором `IDCANCEL`, то переменная `shapeData` сохраняет то значение, которое она имела до вызова диалогового окна.

В оконной процедуре `WndProc` обработка сообщения `WM_COMMAND` расширена новой ветвью:

```
case IDM_DRAW_SHAPE:
    DialogBox(hInst, MAKEINTRESOURCE(IDD_SHAPE_PARAM),
              hWnd, ShapeParamDlgProc);
break;
```

В этом блоке кода при помощи функции `DialogBox` вызывается диалоговое окно `IDD_SHAPE_PARAM`.

Откомпилировав этот проект, проверьте работу приложения `DlgDemo3`.

Клавиатурный интерфейс и порядок обхода элементов управления

Система Windows обеспечивает удобный клавиатурный интерфейс для доступа к элементам управления в диалоговом окне. Перемещение между элементами управления осуществляется при помощи клавиши `Tab` либо при помощи клавиш-стрелок. Нажатие клавиши `Tab` перемещает фокус ввода на *следующий* элемент управления, который имеет стиль `WS_TABSTOP` (или для которого установлено свойство `Tab stop`), *в соответствии с установленным порядком обхода*. Порядок обхода элементов управления соответствует последовательности определений этих элементов в файле описания ресурсов.

При большом количестве элементов управления в диалоговом окне они обычно разбиваются на группы элементов по какому-нибудь функциональному признаку. Каждая группа элементов помещается в свою групповую рамку. Пользователю удобно переходить от одной группы элементов к другой при помощи клавиши `Tab`, а внутри группы переходить от элемента к элементу с помощью клавиш-стрелок. Если элемент управления имеет фокус ввода, то нажатие клавиши пробела производит тот же эффект, что и щелчок мышью.

Чтобы обеспечить привычный для пользователя клавиатурный интерфейс, следует придерживаться несложных правил. Первому элементу управления в группе нужно назначать свойство `Tab stop`, а если это группа переключателей, то еще и свойство `Group`. У остальных элементов в группе оба свойства, `Tab stop` и `Group`, должны быть выключенными.

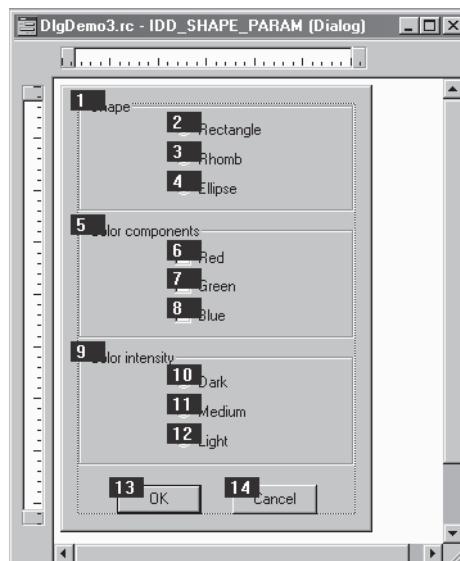


Рис. 7.22. Порядок обхода элементов управления в окне диалога `IDD_SHAPE_PARAM`

После окончания проектирования шаблона диалога порядок обхода элементов управления не всегда получается удобным для пользователя. Среда Visual Studio позволяет легко изменить эту последовательность.

Работая с редактором диалоговых окон, вызовите подменю **Layout**, а в нем выберите команду **Tab order**. Выполняя эту команду, редактор диалоговых окон размещает номера на элементах управления, показывая порядок обхода, первоначально соответствующий порядку добавления элементов управления.

Для изменения порядка обхода последовательно щелкните на каждом элементе управления в задаваемом вами порядке. Например, вид шаблона диалога **IDD_SHAPE_PARAM** из приложения **DlgDemo3** после выполнения указанной процедуры показан на рис. 7.22.

Перейдем теперь к рассмотрению элементов управления **Edit box** и **List box**.

Окно редактирования

Окно редактирования (**Edit box**), которое также часто называют *текстовым полем ввода*, представляет собой прямоугольное окно, в котором пользователь может вводить текст с клавиатуры. Элемент управления **Edit box** реализован на базе предопределенного оконного класса **EDIT**.

На практике используются различные окна редактирования в самом широком спектре: от небольшого односторочного поля ввода до многострочного элемента управления с автоматическим переносом строк, как в программе Microsoft Notepad.

Когда окно редактирования имеет фокус ввода, пользователь может набирать текст, перемещать курсор, удалять символы, выбирать группы символов, используя либо мышь, либо клавишу **Shift** и клавиши-стрелки, удалять или копировать выбранный текст в буфер обмена Windows, вставлять текст из буфера обмена.

Элемент управления **Edit box** размещается на форме диалога так же, как и другие элементы управления. Затем нужно вызвать окно свойств **Edit Properties** и на вкладке **General** в поле **ID** указать идентификатор элемента управления. На вкладке **Styles**, показанной на рис. 7.23, можно задать дополнительные свойства окна редактирования.

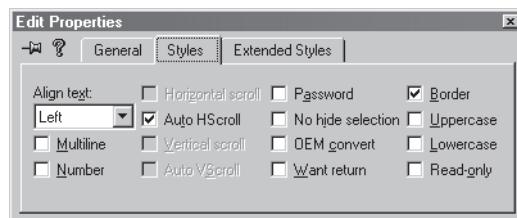


Рис. 7.23. Вкладка **Styles** в окне свойств окна редактирования

По умолчанию окно редактирования является односторочным, с автоматической горизонтальной прокруткой (свойство **Auto HScroll**), с объемной рамкой (свойство **Border**) и выравниванием текста по левой границе окна (**Left**). Открыв комбинированный список **Align text**, можно выбрать другой режим выравнивания текста (**Centered** или **Right**).

Если установить флажок **Multiline**, то окно редактирования будет работать в многострочном режиме. При этом станут доступными для использования опции **Horizontal scroll**, **Vertical scroll**, **Auto VScroll**.

Установленный флажок **Number** разрешает вводить в окно редактирования только цифры. Если же создаваемое окно редактирования предназначено для ввода пароля, то надо установить флажок **Password**, и тогда вместо вводимых символов в поле будут отображаться символы звездочки.

Флажок **No hide selection** заставляет элемент управления всегда показывать выделенную подстроку, даже если окно редактирования теряет фокус. Флажок **OEM convert** задает режим работы, при котором вводимый текст преобразуется из набора символов Windows в набор символов OEM, а затем — обратно. Это обеспечивает правильное преобразование символов при использовании функции **CharToOem**.

Флажок **Want return** используется для многострочного окна редактирования. Если он установлен, то нажатие клавиши **Enter** приводит к вводу символа возврата каретки. Если флажок сброшен, то нажатие клавиши **Enter** воспринимается как щелчок на кнопке, которая отмечена в диалоговом окне в качестве выбранной по умолчанию. В этом случае для ввода символа возврата каретки применяется комбинация клавиш **Ctrl+Enter**.

Взаимоисключающие флажки **Uppercase** и **Lowercase** преобразуют вводимые символы. В первом случае происходит конвертирование в символы верхнего регистра, а во втором — в символы нижнего регистра.

Флажок **Read-only** создает окно редактирования, которое предназначено только для чтения и не дает возможности пользователю вводить или редактировать текст.

Для каждого элемента управления **Edit box** система резервирует в области памяти приложения буфер размером 32 Кбайт для хранения введенного текста.

Посылка сообщений окну редактирования

Для выполнения различных операций по редактированию текста приложение может отправлять сообщения элементу **hwndEdit** с помощью вызова следующей функции:

```
SendMessage(hwndEdit, код_сообщения, wParam, lParam);
```

В параметре **hwndEdit** указывается дескриптор элемента управления.

Win32 API содержит специальную функцию **SendDlgItemMessage** для отправки сообщений элементам управления, обращение к которой выглядит так:

```
SendDlgItemMessage(hDlg, nIDD, код_сообщения, wParam, lParam);
```

где **hDlg** — дескриптор родительского диалогового окна, **nIDD** — идентификатор элемента управления. Иногда проще будет использовать эту функцию, а не **SendMessage**, так как в данном случае не требуется получать дескриптор элемента управления. По результатам выполнения обе функции равнозначны.

В табл. 7.6 описаны некоторые коды сообщений для работы с окном редактирования.

Таблица 7.6. Операции с окном редактирования через коды сообщений

Код сообщения	wParam	lParam	Описание
EM_SETSEL	iStart	iEnd	Выделить текст, начиная с позиции iStart и заканчивая позицией iEnd
EM_GETSEL	&iStart	&iEnd	Получить начальное и конечное положения текущего выделения
WM_CLEAR	0	0	Удалить выделенный текст

Код сообщения	wParam	lParam	Описание
WM_CUT	0	0	Удалить выделенный текст и поместить его в буфер обмена Windows
WM_COPY	0	0	Скопировать выделенный текст в буфер обмена Windows
WM_PASTE	0	0	Вставить текст из буфера обмена в место, соответствующем позиции курсора
WM_GETTEXT	nMax	szBuffer	Скопировать текст (не более nMax символов) из элемента управления в буфер szBuffer
EM_GETLINECOUNT	0	0	Функция возвращает число строк для многострочного окна редактирования
EM_LINELENGTH	iLine	0	Функция возвращает длину строки iLine
EM_GETLINE	iLine	szBuffer	Скопировать содержание строки iLine в буфер szBuffer

Получение сообщений от окна редактирования

Элементы управления Edit box посылают оконной процедуре родительского окна сообщения WM_COMMAND. Эти сообщения содержат в младшем слове параметра wParam идентификатор элемента управления, в старшем слове параметра wParam — код уведомления, а в параметре lParam — дескриптор элемента управления.

Некоторые из кодов уведомления от окна редактирования приведены в табл. 7.7.

Таблица 7.7. Коды уведомления от элемента управления типа Edit box

Код уведомления	Интерпретация
EN_SETFOCUS	Окно получило фокус ввода
EN_KILLFOCUS	Окно потеряло фокус ввода
EN_UPDATE	Содержимое окна будет меняться
EN_CHANGE	Содержимое окна изменилось
EN_ERRSPACE	Произошло переполнение буфера редактирования

Кроме того, перед своей перерисовкой обычное окно редактирования посыпает своему родительскому окну сообщение WM_CTLCOLOREDIT, а окно редактирования с атрибутом «только для чтения» посыпает сообщение WM_CTLCOLORSTATIC. Обрабатывая эти сообщения, родительское окно может изменить цветовые атрибуты окна редактирования аналогично тому, как это было реализовано в приложении DlgDemo1 для элемента управления типа Static Text.

Функции чтения/записи текста для элементов управления

Win32 API содержит функции GetDlgItemText, SetDlgItemText и GetDlgItemInt, которые используются для работы с текстовой информацией, ассоциированной с некоторым элементом управления.

Эти функции могут применяться не только к окну редактирования, но и к надписям, кнопкам, флажкам, переключателям и рамкам.

Функция GetDlgItemText имеет следующий прототип:

```
UINT GetDlgItemText(HWND hDlg, int ctrlID, LPTSTR lpString, int nMaxCount);
```

Она позволяет извлечь текст, ассоциированный с элементом управления, идентификатор которого передается в параметре ctrlID, и записать его в буфер с адрес-

сом `lpString`. Параметр `nMaxCount` задает максимальное количество символов, копируемых в буфер `lpString`. При успешном завершении функция возвращает число символов, прочитанных в буфер, без учета завершающего нуль-символа. При возникновении ошибки функция возвращает нулевое значение.

Функция `SetDlgItemText`, имеющая такие же параметры, осуществляет обратную операцию записи текстовой строки, на которую указывает `lpString`, в элемент управления `ctrlID`.

Функция `GetDlgItemInt` с прототипом

```
UINT GetDlgItemInt(HWND hDlg, int ctrlID, BOOL* lpTranslated, BOOL bSigned);
```

предназначена для извлечения текстового изображения десятичного числа из элемента управления `ctrlID` и преобразования его к целочисленному значению. Если изображению числа предшествуют пробелы, то они отбрасываются. В процессе преобразования функция сканирует текстовую строку символ за символом и, если очередной символ оказывается не цифрой, то прекращает преобразование.

Параметру `lpTranslated` обычно передается адрес переменной типа `BOOL`, в которую записывается значение, свидетельствующее об успешности завершения функции. Значение `TRUE` используется в случае успешного преобразования. Если вас эта информация не интересует, то можно передать данному параметру значение `NULL`.

Если параметр `bSigned` равен `TRUE`, то функция будет учитывать наличие знака «минус» перед числом и возвращать значение типа `int`, несмотря на определение в прототипе функции возвращаемого типа `UINT`. Пробел между минусом и первой цифрой числа не допускается.

Функция `GetDlgItemInt` возвращает нулевое значение, если транслируемое значение превышает `INT_MAX` (2147483647) для знакового числа или `UINT_MAX` (4294967295) для беззнакового числа.

Пример использования функций `GetDlgItemText` и `SetDlgItemText` приведен в листинге 7.4.

Список

Список (*List box*) представляет собой прямоугольное окно, в котором отображается набор элементов, из которых пользователь может делать выбор.

Элементы списка могут быть представлены строками, растровыми образами или комбинацией текста и изображения. Если размеры окна не позволяют показать все элементы списка, то *List box* создает полосу прокрутки. Типичный пример использования списка — список файлов в диалоговом окне *Open*, вызываемом по команде *File ▶ Open* в среде Visual Studio.

Различают списки с единичным выбором, в которых пользователь может выбрать только один пункт списка, и списки с множественным выбором, допускающие выбор более одного пункта списка. Windows показывает выбранный элемент в списке, инвертируя цвет текста и цвет фона для символов.

В списке с единичным выбором пользователь может выбрать пункт списка щелчком мыши или нажатием клавиши пробела после получения списком фокуса ввода. Клавиши управления курсором перемещают как курсор, так и текущую выборку. В списке с множественным выбором первый щелчок на пункте выбирает его, а следующий щелчок отменяет предыдущую операцию.

Программа может добавлять или удалять элементы в списке, отсылая сообщения оконной процедуре списка. Когда пользователь выбирает или отменяет вы-

бор пункта в списке, система посыпает нотификационное сообщение родительскому окну. Родительское окно может определить, какой пункт списка был выбран пользователем.

Элемент управления List box помещается на форму диалога так же, как и другие элементы управления. Затем необходимо вызвать окно свойств List Box Properties и на вкладке General в поле ID указать идентификатор элемента управления. На вкладке Styles, показанной на рис. 7.24, можно задать дополнительные свойства списка.

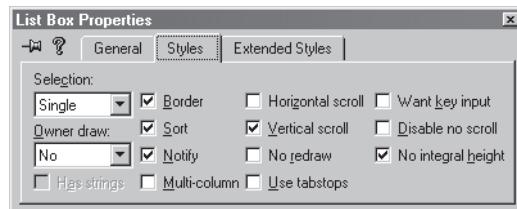


Рис. 7.24. Вкладка Styles в окне свойств списка

По умолчанию создается список с единичным выбором (Single), имеющий рамку (Border), выполняющий автоматическую сортировку элементов списка (Sort) и обеспечивающий появление вертикальной полосы прокрутки в случае необходимости (Vertical scroll).

Флажок No integral height указывает системе, что высота окна списка List box должна быть в точности такой, какой она указана на форме диалога. Если этот флажок сбросить, то Windows может уменьшить высоту окна, чтобы исключить неполное изображение элемента списка.

В окне Selection можно выбрать другой стиль списка. Кроме значения Single здесь доступны следующие значения:

Стиль списка	Описание
Multiple	Список с множественным выбором. Разрешено выбирать несколько элементов, щелкнув на каждом элементе для выбора или отмены выбора
Extended	Список с множественным выбором. Разрешено выбирать несколько элементов только при нажатой клавише Shift
None	Список содержит элементы, которые можно просматривать, но не выбирать

В окне Owner Draw по умолчанию используется опция No. Если необходимо создать список, изображаемый окном-владельцем, то нужно выбрать соответствующий параметр:

Опция	Описание
Fixed	Создать список со стилем Owner draw. Родительское окно получает сообщение WM_MEASUREITEM при создании списка и сообщение WM_DRAWITEM, когда требуется отобразить элемент
Variable	То же, что и Fixed, за исключением того, что размер каждого элемента в списке можно определять отдельно. Для любого элемента в списке перед вызовом сообщения WM_DRAWITEM вызывается сообщение WM_MEASUREITEM

Флажок Has string можно использовать только при выборе стиля Owner draw. Если он установлен, то это означает, что в список должны быть введены строки.

Затем для получения конкретного элемента списка в приложении можно использовать сообщение `LB_GETTEXT`.

Флажок `Multi-column` позволяет создавать многостолбцовый список, который может прокручиваться по горизонтали. Но для этого должен быть установлен флажок `Horizontal scroll`. Для установки ширины столбцов используется сообщение `LB_SETCOLUMNWIDTH`.

Флажок `No redraw` определяет список, окно которого не получает сообщений `WM_PAINT` при внесении изменений. Этот стиль может быть в любой момент отменен посылкой сообщения `WM_SETREDRAW`.

Флажок `Disable no scroll` заставляет отображать в списке вертикальную полосу прокрутки, даже если список не содержит необходимого количества элементов. При этом полоса прокрутки недоступна до тех пор, пока не появится достаточное количество элементов. По умолчанию полоса прокрутки скрыта, если список содержит столько элементов, что все они отображаются в окне.

Посылка сообщений списку

Для выполнения различных операций со списком приложение может отправлять сообщения элементу `hwndList`, вызывая функцию

```
SendMessage(hwndList, код_сообщения, wParam, lParam);
```

или равнозначную ей функцию

```
SendDlgItemMessage(hDlg, IDC_LIST, код_сообщения, wParam, lParam);
```

где `IDC_LIST` — идентификатор элемента управления.

Результаты выполнения этих функций для некоторых кодов сообщений описываются в табл. 7.8.

Таблица 7.8. Операции со списком через коды сообщений

Код сообщения	wParam	lParam	Описание
<code>LB_ADDSTRING</code>	0	<code>szString</code>	Добавляет в список строку <code>szString</code> . Если свойство <code>Sort</code> у списка выключено, то строка будет добавлена в конец списка. Если это свойство включено, то после добавления строки производится сортировка списка
<code>LB_DELETESTRING</code>	<code>iIndex</code>	0	Удаляет строку с индексом <code>iIndex</code>
<code>LB_FINDSTRING</code>	<code>iStart</code>	<code>szString</code>	Ищет строку, начинающуюся префиксом <code>szString</code> . Поиск начинается с элемента с индексом <code>iStart + 1</code> . Если <code>wParam</code> равен <code>-1</code> , то поиск начинается с начала списка. Функция возвращает индекс найденной строки или <code>LB_ERR</code> , если поиск завершился неуспешно
<code>LB_GETCOUNT</code>	0	0	Возвращает количество элементов в списке
<code>LB_GETCURSEL</code>	0	0	Возвращает индекс текущего выбранного элемента или <code>LB_ERR</code> , если такого элемента нет. Код можно использовать только для списков с единичным выбором
<code>LB_GETSELCOUNT</code>	0	0	Возвращает количество выбранных элементов (для списка с множественным выбором)
<code>LB_GETSELITEMS</code>	<code>nMax</code>	<code>pBuf</code>	Заполняет буфер с адресом <code>pBuf</code> массивом индексов выделенных элементов для списка с множественным выбором. <code>nMax</code> задает максимальное количество таких элементов

Код сообщения	wParam	lParam	Описание
LB_GETTEXT	iIndex	szString	Копирует строку с индексом iIndex в буфер szString. Размер буфера должен быть достаточным для размещения строки. В случае выделения динамической памяти требуемый размер можно узнать, послав сообщение LB_GETTEXTLEN.
LB_GETTEXTLEN	iIndex	0	Возвращает длину строки с индексом iIndex
LB_INSERTSTRING	iIndex	szString	Вставляет строку szString после строки с индексом iIndex. Применяется для списка с выключенным свойством Sort
LB_RESETCONTENT	0	0	Удаляет все элементы из списка
LB_SELECTSTRING	iStart	szString	Выполняется так же, как для сообщения LB_FINDSTRING, но дополнительно выделяет найденную строку
LB_SETCURSEL	iIndex	0	Выбирает элемент с индексом iIndex (в списке с единичным выбором)
LB_SETSEL	wParam	iIndex	Выбирает элемент с индексом iIndex (в списке с множественным выбором). Если wParam равен TRUE, элемент выбирается и выделяется, если FALSE — выбор отменяется. Если lParam равен -1, то операция применяется ко всем элементам списка

Ссылка на строки текста в списке обычно осуществляется через индекс iIndex, который начинается с нуля, что соответствует самому верхнему элементу списка.

Функция SendMessage после отправки некоторых сообщений, не использующих код возврата для передачи результата, может вернуть один из специфичных кодов. Код LB_OK (0) возвращается при нормальном завершении функции, код LB_ERRSPACE (-2) — в случае нехватки памяти для сохранения содержимого списка, LB_ERR (-1) — при возникновении любой другой ошибки.

Получение сообщений от списка

Когда пользователь щелкает мышью в окне списка, оно получает фокус ввода и посыпает оконной процедуре родительского окна сообщение WM_COMMAND, в котором старшее слово параметра wParam содержит код уведомления.

Наиболее распространенные коды уведомления приведены в табл. 7.9.

Таблица 7.9. Коды уведомления от элемента управления типа List box

Код уведомления	Интерпретация
LBN_SETFOCUS	Окно получило фокус ввода
LBN_KILLFOCUS	Окно потеряло фокус ввода
LBN_SELCHANGE	Текущий выбор был изменен (список должен иметь свойство Notify)
LBN_DBLCLK	На данном пункте списка был двойной щелчок мыши (список должен иметь свойство Notify)
LBN_ERRSPACE	Превышен размер памяти, отведенный для списка

Пример использования элементов управления Edit box и List box

Различные вариации на тему проекта DlgDemo..., вам, наверное, уже надоели. Давайте создадим что-нибудь новенькое. Например, приложение «Электронная

записная книжка» для хранения сведений о ваших друзьях и знакомых. Эти сведения могут включать:

- имя (фамилия и.о.), содержащее не более 20 символов;
- телефон (максимум 20 символов);
- дата рождения (максимум 20 символов);
- адрес (максимум 80 символов).

Реальное хранение информации будет обеспечиваться с помощью «базы данных», реализованной в файле notebook.dat. Каждая запись в базе данных, или строка файла, имеет формат, показанный в табл. 7.10.

Таблица 7.10. Формат записи для базы данных и для списка List box

Поле:	Имя	Телефон	День рождения	Адрес
Байты:	0 1 ... 20	21 22 ... 41	42 43 ... 62	63 64 ... 143

Заметим, что каждое поле имеет длину на единицу большую, чем того требует спецификация программы. Дополнительный байт в конце каждого поля учитывает размещение нуля при записи значения в виде С-строки.

Для простоты реализации приложение будет иметь меню с единственным пунктом Просмотр, при выборе которого будет вызываться диалоговое окно, содержащее окно списка List box. Диалоговое окно должно содержать кнопки, при помощи которых пользователь может добавить новую запись в список или удалить выбранную запись из списка.

Примем решение, что для удобства пользователя окно списка должно отображать только первые два поля каждой записи (мы считаем их основными). Дополнительная информация, показывающая день рождения и адрес, должна отображаться в отдельных окнах редактирования (Edit box) только после выбора какого-либо элемента в окне списка.

Создайте новый проект типа Win32 Application с именем MyNotebook1. Добавьте к приложению ресурс меню IDR_MENU1 с одним пунктом. Пункт меню должен иметь имя Просмотр и идентификатор IDM_VIEW.

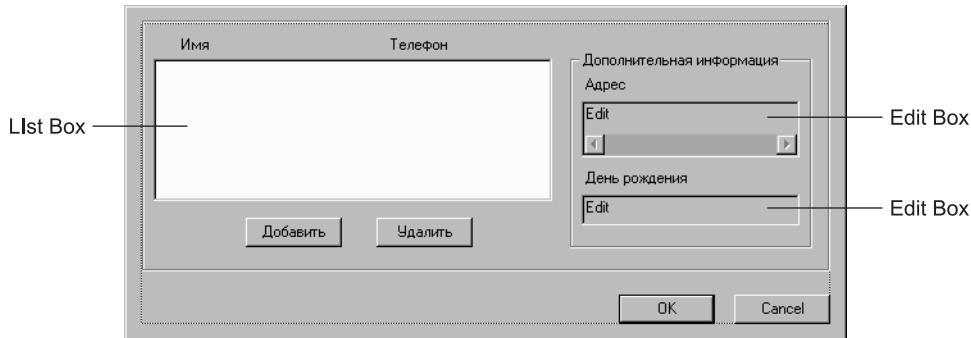


Рис. 7.25. Форма диалога IDD_VIEW

Включите в приложение ресурс диалогового окна IDD_VIEW. В свойствах диалогового окна снимите флагшки Title bar и System menu. Теперь увеличьте ширину

первоначальной формы диалога примерно вдвое и разместите на ней элементы управления в соответствии с рис. 7.25.

Ширину окна списка, возможно, придется изменить. Это выяснится на этапе тестирования программы. В окне списка должны быть полностью видны первые два поля каждой записи, но не должно попадать начало третьего поля.

Позже, при разработке приложения MyNotebook2, мы покажем, как можно регулировать программным способом размеры и размещение элементов управления в зависимости от метрик используемого шрифта.

Для элементов управления должны быть установлены следующие атрибуты:

Элемент	ID	Caption	Свойства
Group box	IDC_GROUP_0		По умолчанию
Static text	IDC_STATIC	Имя	По умолчанию
Static text	IDC_STATIC_PHONE	Телефон	По умолчанию
List box	IDC_LIST1	—	По умолчанию
Button	IDC_REC_ADD	Добавить	По умолчанию
Button	IDC_REC_DELETE	Удалить	По умолчанию
Group box	IDC_GROUP_1	Дополнительная информация	По умолчанию
Static text	IDC_STATIC_ADDR	Адрес	По умолчанию
Edit box	IDC_INFO_ADDR	—	Установить дополнительно флажки Multiline, Horizontal scroll, Read-only
Static text	IDC_STATIC_BDAY	День рождения	По умолчанию
Edit box	IDC_INFO_BDAY	—	Установить дополнительно флажок Read-only

Включите в приложение ресурс диалогового окна IDD_ADD_REC с заголовком Ввод записи. Свойства диалогового окна оставьте по умолчанию. Поместите на форму диалога IDD_ADD_REC четыре надписи и четыре окна редактирования. Размещение элементов управления должно соответствовать рис. 7.26.

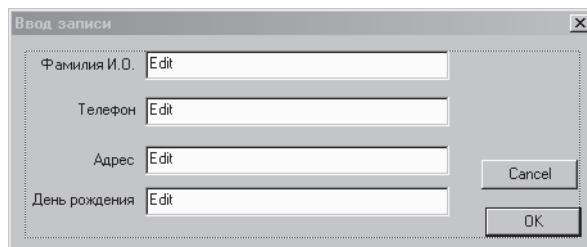


Рис. 7.26. Форма диалога IDD_ADD_REC

Для элементов управления установите следующие атрибуты:

Элемент	ID	Caption	Свойства
Static text	IDC_STATIC	Имя	По умолчанию
Static text	IDC_STATIC	Телефон	По умолчанию
Static text	IDC_STATIC	Адрес	По умолчанию

продолжение ↗

Элемент	ID	Caption	Свойства
Static text	IDC_STATIC_BDAY	День рождения	По умолчанию
Edit box	IDC_INPUT_NAME	—	По умолчанию
Edit box	IDC_INPUT_PHONE	—	По умолчанию
Edit box	IDC_INPUT_ADDR	—	По умолчанию
Edit box	IDC_INPUT_BDAY	—	По умолчанию

Осталось написать код. Добавьте к проекту файл MyNotebook1.cpp, текст которого приведен в листинге 7.4.

Листинг 7.4. Проект MyNotebook1

```
//////////  
// MyNotebook1.cpp  
#include <windows.h>  
#include <fstream>  
using namespace std;  
  
#include "KWnd.h"  
#include "resource.h"  
  
#define N1 21  
#define N2 81  
#define TOTAL_SIZE (3*N1 + N2)  
  
fstream fdat;  
  
struct SubStr {  
    char name[N1];  
    char phone[N1];  
    char birthday[N1];  
    char address[N2];  
};  
union Data {  
    SubStr ss;  
    char line[TOTAL_SIZE];  
};  
  
class ListItem {  
    char buf[TOTAL_SIZE];  
public:  
    ListItem() { Clear(); }  
  
    // Очистка данных (заполнение строки line пробелами)  
    void Clear() { memset(dat.line, ' ', TOTAL_SIZE); }  
  
    // Замена в строке line всех нулевых байтов, кроме последнего,  
    // символом 'пробел'  
    void DoOneString() {  
        int count = 0;  
        for (int i = TOTAL_SIZE - 1; i > 0; --i)  
            if (!dat.line[i])  
                if (!count) count++;  
                else dat.line[i] = ' ';  
    }  
  
    const char* GetBirthday() {  
        memcpuy(buf, dat.ss.birthday, N1);  
    }
```

```
buf[N1-1] = 0;
return buf;
}

const char* GetAddress() {
    memcpy(buf, dat.ss.address, N2);
    buf[N2-1] = 0;
    return buf;
}

Data dat;
};

ListItem item;

BOOL CALLBACK ViewDlgProc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK AddRecDiaProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("MyNotebook1", hInstance, nCmdShow, WndProc,
        MAKEINTRESOURCE(IDR_MENU1), 100, 100, 400, 300);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HINSTANCE hInst;

    switch (uMsg) {

        case WM_COMMAND:
            switch (LOWORD(wParam)) {
                case IDM_VIEW:
                    hInst = GetModuleHandle(NULL);
                    DialogBox(hInst, MAKEINTRESOURCE(IDD_VIEW), hWnd, ViewDlgProc);
                    break;
            }
            break;

        case WM_DESTROY:
            PostQuitMessage(0); break;

        default:
            return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }
    return 0;
}

=====
```

Листинг 7.4 (продолжение)

```
BOOL CALLBACK ViewDlgProc(HWND hDlg, UINT uMsg, WPARAM wParam,
    LPARAM lParam) {

    static HWND hListBox;
    static HWND hEditInfoAddr, hEditInfoBday;
    HWND hCtrl;
    static LOGFONT lf;
    HFONT hFont1;
    COLORREF editInfoColor;
    static HBRUSH hBrush;
    int nItems; // количество элементов в списке
    int iCurItem; // текущий выбранный элемент списка
    BOOL dlgYes;
    int i;

    switch (uMsg) {
        case WM_INITDIALOG:
            hListBox = GetDlgItem(hDlg, IDC_LIST1);
            hEditInfoAddr = GetDlgItem(hDlg, IDC_INFO_ADDR);
            hEditInfoBday = GetDlgItem(hDlg, IDC_INFO_BDAY);

            // Модификация шрифта для элемента hListBox
            lf.lfHeight = 16;
            lstrcpy( (LPSTR)&lf.lfFaceName, "Courier" );
            hFont1 = CreateFontIndirect(&lf);
            SendMessage(hListBox, WM_SETFONT, (WPARAM)hFont1, TRUE );
            // Создание кисти для фона элементов управления
            editInfoColor = RGB(190, 255, 255);
            hBrush = CreateSolidBrush(editInfoColor);

            // Чтение "базы данных" из файла и инициализация списка
            fdat.open("notebook.dat", ios::in);
            if (!fdat) { fdat.clear(); return TRUE; }
            else while (1) {
                fdat.getline(item.dat.line, sizeof(Data));
                if (fdat.eof()) { fdat.clear(); break; }
                SendMessage(hListBox, LB_ADDSTRING, 0,
                           (LPARAM)item.dat.line);
            }
            fdat.close();
            return TRUE;

        case WM_CTLCOLORSTATIC:
            // Модификация цветовых атрибутов окон редактирования
            hCtrl = (HWND)lParam;
            if ((hCtrl == hEditInfoAddr) || (hCtrl == hEditInfoBday)) {
                SetBkColor((HDC)wParam, editInfoColor);
                SetBkMode((HDC)wParam, TRANSPARENT);
                return (DWORD)hBrush;
            }
            break;

        case WM_COMMAND:
            switch (LOWORD(wParam))
            {
```

```
case IDC_REC_ADD:
    dlgYes = DialogBox((HINSTANCE)GetModuleHandle(NULL),
        MAKEINTRESOURCE(IDD_ADD_REC), hDlg, AddRecDiaProc);
    if (dlgYes)
        SendMessage(hListBox, LB_ADDSTRING, 0,
            (LPARAM)item.dat.line);
    break;

case IDC_REC_DELETE:
    iCurItem = SendMessage(hListBox, LB_GETCURSEL, 0, 0);
    if (iCurItem != LB_ERR) {
        SendMessage(hListBox, LB_DELETESTRING, iCurItem, 0);
        SetDlgItemText(hDlg, IDC_INFO_ADDR, "");
        SetDlgItemText(hDlg, IDC_INFO_BDAY, "");
    }
    else {
        MessageBox(hDlg, "Сначала надо выделить элемент списка",
            "Ошибка", MB_OK);
        break;
    }
    break;

case IDC_LIST1:
    switch (HIWORD(wParam)) {
    case LBN_SELCHANGE:
        // Показываем дополнительную информацию
        iCurItem = SendMessage(hListBox, LB_GETCURSEL, 0, 0);
        SendMessage(hListBox, LB_GETTEXT, iCurItem,
            (LPARAM)item.dat.line);
        SetDlgItemText(hDlg, IDC_INFO_ADDR, item.GetAddress());
        SetDlgItemText(hDlg, IDC_INFO_BDAY, item.GetBirthday());
        return TRUE;
    }
    break;

case IDOK:
    fdat.open("notebook.dat", ios::out | ios::trunc);
    if (!fdat) {
        MessageBox(hDlg,
            "Не могу открыть файл 'notebook.dat' для записи!",
            "Ошибка", MB_OK);
        return FALSE;
    }
    nItems = SendMessage(hListBox, LB_GETCOUNT, 0, 0);
    for (i = 0; i < nItems; ++i) {
        SendMessage(hListBox, LB_GETTEXT, i, (LPARAM)item.dat.line);
        fdat << item.dat.line << endl;
    }
    fdat.close();

    EndDialog(hDlg, TRUE);
    return TRUE;

case IDCANCEL:
    EndDialog(hDlg, FALSE);
```

Листинг 7.4 (продолжение)

```

        return TRUE;
    }
    break;
}
return FALSE;
}

//=====
BOOL CALLBACK AddRecDiaProc(HWND hDlg, UINT uMsg,
    WPARAM wParam, LPARAM lParam)
{
    switch (uMsg) {

case WM_INITDIALOG:
    SetFocus(GetDlgItem(hDlg, IDC_INPUT_NAME));
    return FALSE;

case WM_COMMAND:
    switch (LOWORD(wParam))
    {
    case IDOK:
        item.Clear();
        GetDlgItemText(hDlg, IDC_INPUT_NAME, item.dat.ss.name, N1);
        GetDlgItemText(hDlg, IDC_INPUT_PHONE, item.dat.ss.phone, N1);
        GetDlgItemText(hDlg, IDC_INPUT_BDAY, item.dat.ss.birthday, N1);
        GetDlgItemText(hDlg, IDC_INPUT_ADDR, item.dat.ss.address, N2);
        item.DoOneString();
        EndDialog(hDlg, TRUE);
        return TRUE;
    case IDCANCEL:
        EndDialog(hDlg, FALSE);
        return TRUE;
    }
    break;
}
return FALSE;
}
///////////

```

Информация, которую необходимо хранить в списке и записывать в файл для длительного хранения, представляет собой последовательность записей длиной 144 байта. В этих операциях запись должна интерпретироваться как одна С-строка с завершающим нулевым байтом. В то же время, когда формируется новая запись с помощью диалогового окна **Ввод записи**, отдельные ее поля получают значения С-строк в результате вызова функции `GetDlgItemText`. Поэтому появляется проблема «склеивания» полей для получения одной С-строки. В другой ситуации, когда требуется отобразить дополнительную информацию для выбранной в списке записи, возникает обратная проблема: извлечь С-подстроку из общей С-строки.

Чтобы упростить эти операции, в программе определен класс `ListItem`. Для хранения записи он содержит член `dat`, имеющий тип объединения (*union*) `Data`. С помощью объединения мы можем интерпретировать один и тот же участок памяти и как одну строку `line`, и как структуру `ss`, содержащую четыре поля, `name`, `phone`, `birthday` и `address`.

В классе также определен метод `DoOneString`, склеивающий четыре подстроки в одну строку. Эта подзадача решается путем сканирования массива `dat.line` от его конца к началу, с заменой всех нулевых байтов, кроме ближайшего к концу массива, символом пробела. В классе также определены метод `GetBirthday`, выполняющий извлечение в виде С-строки значения поля `dat.ss.birthday`, и метод `GetAddress`, делающий то же самое для поля `dat.ss.address`.

Переменная `item`, которая является объектом класса `ListItem`, объявлена в глобальной области видимости программы. Эта переменная используется и для внутренних нужд диалоговой процедуры `ViewDlgProc`, и для обмена данными между диалоговыми процедурами `AddRecDiaProc` и `ViewDlgProc`.

Диалоговая процедура `ViewDlgProc`, обслуживающая диалог `IDD_VIEW`, выполняет самую большую часть работы данного приложения.

Блок обработки сообщения `WM_INITDIALOG` содержит следующую функциональность:

- ❑ Создается специальный шрифт с дескриптором `hFont1` для использования в окне списка. В программе используется моноширинный шрифт `Courier`, чтобы обеспечить правильное расположение границы второго столбца, в котором выводится информация о телефоне. Модификация шрифта в окне `hListBox` реализуется отправкой сообщения `WM_SETFONT`.
- ❑ Создается кисть `hBrush` для изменения фона элементов управления `hEditInfo-Addr` и `hEditInfoBday`. Эта кисть применяется в блоке обработки сообщения `WM_CTLCOLORSTATIC`.
- ❑ Инициализируется список `hListBox`. Для этого из файла `notebook.dat` считывается информация, и каждая прочитанная строка добавляется в список с помощью посылки сообщения `LB_ADDSTRING`.

В блоке обработки сообщения `WM_COMMAND` решаются следующие подзадачи:

- ❑ Если сообщение поступило от кнопки `IDC_REC_ADD`, то при помощи функции `DialogBox` вызывается модальный диалог `IDD_ADD_REC`. Работая с этим диалогом, пользователь вводит информацию для очередной записи. Если ввод завершен нажатием кнопки `OK`, то функция `DialogBox` вернет значение `TRUE` и будет вызвана функция `SendMessage` для отправки сообщения `LB_ADDSTRING`.
- ❑ Если сообщение поступило от кнопки `IDC_REC_DELETE`, мы определяем, какой пункт списка выделен пользователем. Если такой пункт есть, то отправляем окну списка сообщение `LB_DELETESTRING`, а также очищаем окна редактирования `IDC_INFO_ADDR` и `IDC_INFO_BDAY` при помощи функции `SetDlgItemText` (в них может отображаться информация для выбранного пункта списка).
- ❑ Если сообщение поступило от окна списка `IDC_LIST1` и оно содержит код уведомления `LBN_SELCHANGE`, то надо получить данные для выделенного пункта `iCurItem`. Для этого отправляется сообщение `LB_GETTEXT`. Затем поля записи, сохраненной в объекте `item`, с информацией об адресе и дне рождения выводятся с помощью функции `SetDlgItemText` в окна редактирования `IDC_INFO_ADDR` и `IDC_INFO_BDAY`.
- ❑ Если сообщение поступило от кнопки `IDOK`, то приложение записывает содержимое списка в файл `notebook.dat`.

Диалоговая процедура AddRecDiaProc очень проста и не требует дополнительных пояснений. Но все же стоит обратить внимание на вызов метода item.DoOneString, выполняющего объединение четырех подстрок в одну С-строку.

На рис. 7.27 показан внешний вид работающей программы MyNotebook1.

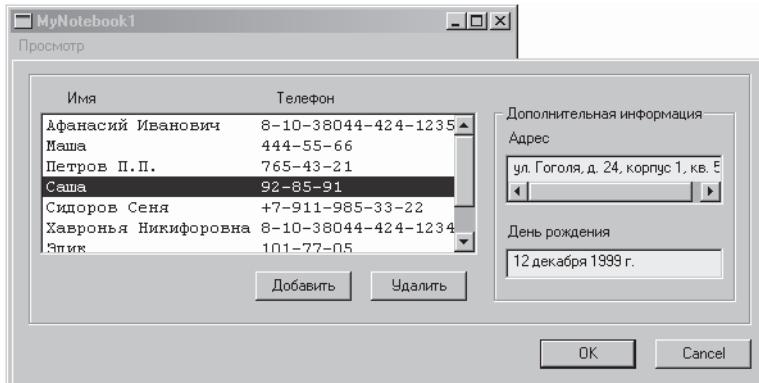


Рис. 7.27. Просмотр содержимого списка в приложении MyNotebook1

Комбинированный список

Комбинированный список (Combo box), называемый также *комбинированным окном*, состоит из окна редактирования и окна списка. В классическом варианте комбинированный список выглядит как окно редактирования со стрелкой справа. Если на ней щелкнуть мышью, то появится выпадающий список.

Комбинированный список помещается на форму диалога при помощи мыши с предварительным выделением элемента Combo box на панели инструментов Controls. Сразу после щелчка на форме диалога элемент Combo box выглядит примерно так, как показано на рис. 7.28.

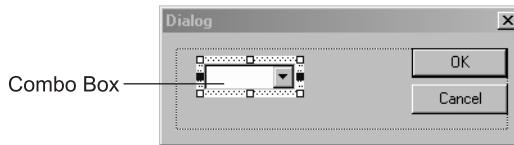


Рис. 7.28. Размещение комбинированного списка на форме диалога

Обратите внимание на специальную «редакторскую» рамку вокруг окна элемента Combo box. Она говорит о том, что элемент управления сейчас находится в выделенном состоянии. Мы уже имели дело с выделением элементов управления, занимаясь выравниванием их размещения на форме диалога с помощью подменю Layout. Но сейчас речь пойдет совсем о другом.

Щелкните на стрелке справа, чтобы открыть окно выпадающего списка. Редактор диалога покажет размеры этого окна, растянув прямоугольник рамки вниз. А теперь — внимание! Если размеры окна списка окажутся слишком маленькими и их не будет хватать для отображения хотя бы одного-двух элементов списка, то элемент управления Combo box работать не будет! Поэтому растяните мышью ог-

раничивающую рамку окна списка до того размера, который вы хотите увидеть в работающем приложении (рис. 7.29).

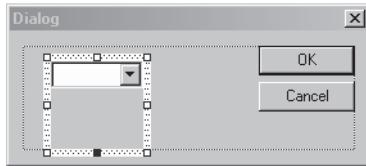


Рис. 7.29. Установка размеров для выпадающего списка

К сожалению, ни в справочной помощи MSDN, ни в других источниках информации об этой технологической «мелочи» ничего не говорится. Я знаю программистов, которые потеряли много часов, выясняя, почему у них не работает *Combo box*, пока они не находили ответ случайно в процессе экспериментов либо благодаря подсказке более опытных коллег.

После размещения элемента управления *Combo box* на форме диалога вы определяете его атрибуты, вызывая окно свойств *Combo box Properties*.

На вкладке *General* в поле ID введите необходимый идентификатор элемента управления. На вкладке *Styles* (рис. 7.30) установите дополнительные свойства комбинированного списка.

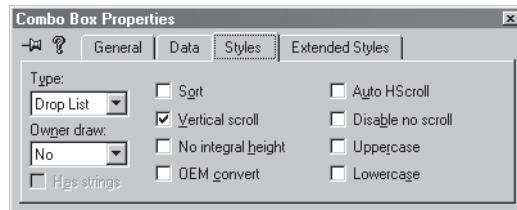


Рис. 7.30. Вкладка Styles в окне Combo Box Properties

Многие свойства *Combo box* аналогичны свойствам элемента управления *List box*. Но есть и некоторые различия. Стиль списка выбирается в окне Type из следующих значений:

Стиль	Описание
Simple	Комбинированный список отображается окном редактирования и раскрытым окном списка. Содержание окна редактирования можно изменять
Dropdown	Комбинированный список отображается в свернутом состоянии со стрелкой справа. Содержание окна редактирования можно изменять
Drop List	Комбинированный список отображается в свернутом состоянии со стрелкой справа. Содержание окна редактирования изменять нельзя

В окне свойств появилась новая вкладка *Data*. Разработчики, создающие приложения на базе библиотеки классов MFC, пользуются вкладкой *Data* для ввода строк, которыми инициализируется *Combo box*. Редактор диалоговых окон сохраняет введенные строки в файле описания ресурсов в виде определения ресурса **DLGINIT**. К сожалению, работая с Win32 API, вы не можете использовать подобный способ инициализации списка, так как в этом случае отсутствует поддержка

интерпретации ресурса типа **DLGINIT**. Для инициализации комбинированного списка применяется посылка сообщений **CB_ADDSTRING**.

Обмен сообщениями с комбинированным списком

Обмен сообщениями с комбинированным списком во многом похож на обмен сообщениями с элементами **Edit box** и **List box**. При этом идентификаторы сообщений, отправляемых элементу **Combo box**, начинаются с префикса **CB_**, а идентификаторы уведомительных сообщений, посылаемых от **Combo box** своему родительскому окну, имеют префикс **CBN_**.

В табл. 7.11 приведено сопоставление некоторых кодов сообщений, посылаемых элементу управления **Combo box**, с аналогичными сообщениями, посылаемыми элементам управления **Edit box** и **List box**.

Таблица 7.11. Однотипные сообщения для элементов управления **Edit box**, **List box** и **Combo box**

Код сообщения для Edit box или List box	Код сообщения для Combo box
EM_GETSEL	CB_GETEDITSEL
EM_SETSEL	CB_SETEDITSEL
LB_ADDSTRING	CB_ADDSTRING
LB_DELETESTRING	CB_DELETESTRING
LB_GETCOUNT	CB_GETCOUNT
LB_GETCURSEL	CB_GETCURSEL
LB_GETTEXT	CB_GETLBTEXT
LB_SETCURSEL	CB_SETCURSEL

Полный список сообщений, посылаемых элементу **Combo box**, можно найти в справочных материалах MSDN. Всего существует более шестидесяти подобных сообщений.

В табл. 7.12 приведено аналогичное сопоставление некоторых кодов уведомительных сообщений.

Таблица 7.12. Однотипные уведомительные сообщения от **Edit box**, **List box** и **Combo box**

Код сообщения от Edit box или List box	Код сообщения от Combo box
EN_SETFOCUS , LBN_SETFOCUS	CBN_SETFOCUS
EN_KILLFOCUS , LBN_KILLFOCUS	CBN_KILLFOCUS
EN_UPDATE	CBN_EDITUPDATE
EN_CHANGE	CBN_EDITCHANGE
EN_ERRSPACE , LBN_ERRSPACE	CBN_ERRSPACE
LBN_SELCHANGE	CBN_SELCHANGE

Пример использования элемента управления **Combo box**

Сделаем небольшую доработку предыдущего приложения. Добавим возможность для пользователя выбирать размер шрифта в окне списка нашей электронной записной книжки. Нам будут очень признательны те пользователи, у которых есть проблемы со зрением и которым трудно читать мелкий шрифт.

Для выбора размера шрифта используем элемент управления **Combo box**. В новой программе кроме изменения шрифта, используемого в окне списка IDC_LIST1, необходимо изменять ширину этого окна так, чтобы в нем помещались первые два поля каждой записи. Кроме того, должны регулироваться размеры диалогового окна и позиции других элементов управления в зависимости от используемого шрифта.

Проблему изменения размеров и позиции окна приходится решать в самых разных приложениях. Поэтому мы разработаем функцию ShiftWindow общего применения, код которой будет размещен в одном файле с реализацией класса **KWnd**.

Прежде всего создайте новый проект с именем **MyNotebook2**. Скопируйте из папки проекта **MyNotebook1** (см. листинг 7.4) в папку проекта **MyNotebook2** файлы с расширениями **.cpp**, **.h** и **.rc**, скорректировав их имена заменой подстроки **MyNotebook1** на **MyNotebook2**. Измените имена файлов **KWnd.h**, **KWnd.cpp**, содержащих интерфейс и реализацию класса **KWnd**, на имена **KWndPlut.h**, **KWndPlut.cpp**¹. Добавьте все перечисленные файлы в состав проекта.

Откройте шаблон диалога **IDD_VIEW** двойным щелчком мыши на соответствующем значке на вкладке **Resource View** в окне **Workspace**. Добавьте на форму диалога (рис. 7.31) элементы управления **Static Text** (с текстом **Шрифт:**) и **Combo box**.

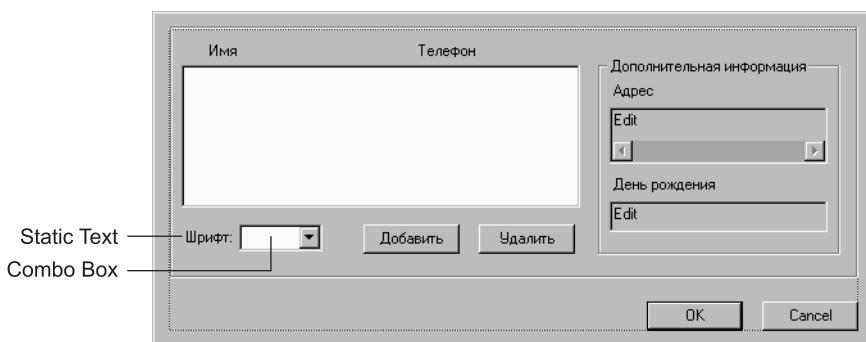


Рис. 7.31. Размещение на форме диалога новых элементов управления

В окне свойств элемента **Combo box** укажите идентификатор **IDC_FONT**. На вкладке **Styles** выберите стиль списка **Drop List** и сбросьте флагок **Sort**.

Измените текст в файлах **KWndPlut.h**, **KWndPlut.cpp** и **MyNotebook2.cpp** так, чтобы он соответствовал листингу 7.5.

Листинг 7.5. Проект MyNotebook2

```
////////////////////////////////////////////////////////////////
// KWndPlut.h
#include <windows.h>

class KWnd {
/* Текст из листинга 1.2 */
};

//=====

```

продолжение ↗

¹ Plut – plus utilities.

Листинг 7.5 (продолжение)

```
// Утилиты общего применения
//=====
// Сдвиг окна верхнего уровня с модификацией размеров
void ShiftWindow(HWND hwnd, int dx = 0, int dY = 0,
                  int dw = 0, int dh = 0);
// Сдвиг дочернего окна с модификацией размеров
void ShiftWindow(HWND hChild, HWND hParent, int dX = 0,
                  int dY = 0, int dW = 0, int dH = 0);
// Сдвиг окна элемента управления с модификацией размеров
void ShiftWindow(int ctrlID, HWND hParent, int dX = 0,
                  int dY = 0, int dW = 0, int dH = 0);
///////////////////////////////
// KWndPlut.cpp
#include "KWndPlut.h"

KWnd::KWnd(LPCTSTR windowName, HINSTANCE hInst, int cmdShow,
           LRESULT (WINAPI *pWndProc)(HWND,UINT,WPARAM,LPARAM),
           LPCTSTR menuName, int x, int y, int width, int height,
           UINT classStyle, DWORD windowStyle, HWND hParent)
{
/* Текст из листинга 1.2 */
}

//=====
// Утилиты общего применения
//=====
// Сдвиг окна верхнего уровня с модификацией размеров
void ShiftWindow(HWND hwnd, int dx, int dY, int dW, int dH) {
    RECT rect;

    GetWindowRect(hwnd, &rect);
    int x0 = rect.left + dx;
    int y0 = rect.top + dY;
    int width = rect.right - rect.left + dW;
    int height = rect.bottom - rect.top + dH;
    MoveWindow(hwnd, x0, y0, width, height, TRUE);
}

// Сдвиг дочернего окна с модификацией размеров
void ShiftWindow(HWND hChild, HWND hParent, int dX, int dY, int dW, int dH) {
    RECT rect;
    POINT p0;

    GetWindowRect(hChild, &rect);
    int width = rect.right - rect.left + dW;
    int height = rect.bottom - rect.top + dH;

    p0.x = rect.left + dX;
    p0.y = rect.top + dY;
    ScreenToClient(hParent, &p0);

    MoveWindow(hChild, p0.x, p0.y, width, height, TRUE);
}

// Сдвиг окна элемента управления с модификацией размеров
void ShiftWindow(int ctrlID, HWND hParent, int dX, int dY, int dW, int dH) {
    RECT rect;
    HWND hCtrl = GetDlgItem(hParent, ctrlID);
```

```
POINT p0;

GetWindowRect(hCtrl, &rect);
int width = rect.right - rect.left + dW;
int height = rect.bottom - rect.top + dH;

p0.x = rect.left + dX;
p0.y = rect.top + dY;
ScreenToClient(hParent, &p0);

ShowWindow(hCtrl, SW_HIDE);
MoveWindow(hCtrl, p0.x, p0.y, width, height, TRUE);
ShowWindow(hCtrl, SW_SHOW);
}

///////////
// MyNotebook2.cpp
#include <windows.h>
#include <fstream>
using namespace std;

#include "KWndPlut.h"
#include "resource.h"

#define N1 21
#define N2 81
#define TOTAL_SIZE (3*N1 + N2)

fstream fdat;

struct SubStr {
    /* Текст из листинга 7.4 */
};

union Data {
    /* Текст из листинга 7.4 */
};

class ListItem {
    /* Текст из листинга 7.4 */
};

ListItem item;
int fontHeight[] = { 16, 20, 24, 28, 30 };
int iFont = 0; // текущий шрифт для окна List box

BOOL CALLBACK ViewDlgProc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK AddRecDiaProc(HWND, UINT, WPARAM, LPARAM);
void AdjustDialogSize(HWND hDlg);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("MyNotebook2", hInstance, nCmdShow, WndProc,
        MAKEINTRESOURCE(IDR_MENU1), 100, 100, 400, 300);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
```

продолжение ↗

Листинг 7.5 (продолжение)

```
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    /* Текст из листинга 7.4 */
}

//=====
// Настройка размеров диалога и размещение элементов управления
// по метрикам шрифта, установленного в окне списка
void AdjustDialogSize(HWND hDlg) {

    int nCtrlId[] = { IDC_GROUP_1, IDC_STATIC_ADDR, IDC_STATIC_BDAY,
                      IDC_INFO_ADDR, IDC_INFO_BDAY, IDOK, IDCANCEL, IDC_REC_ADD,
                      IDC_REC_DELETE };
    HWND hListBox = GetDlgItem(hDlg, IDC_LIST1);
    static LOGFONT lf;
    static HFONT hFont1;
    HDC hDC;

    TEXTMETRIC tm;
    int showTextWidth; // длина текста (в пикселях), показываемого
                       // в окне списка
    RECT rclB;         // первоначальное размещение окна списка
    int shift;          // сдвиг правой границы окна списка для
                       // отображения текста длиной showTextWidth

    // Модификация шрифта для элемента hListBox
    if (hFont1) DeleteObject(hFont1);
    lf.lfHeight = fontHeight[iFont];
    lstrcpy( (LPSTR)&lf.lfFaceName, "Courier" );
    hFont1 = CreateFontIndirect(&lf);
    SendMessage(hListBox, WM_SETFONT, (WPARAM)hFont1, TRUE );

    // Модификация формы диалога с учетом размера шрифта,
    // используемого в окне списка
    hDC = GetDC(hListBox);
    SelectObject(hDC, hFont1);
    GetTextMetrics(hDC, &tm);
    ReleaseDC(hListBox, hDC);

    GetWindowRect(hListBox, &rclB);
    showTextWidth = tm.tmAveCharWidth * 2 * N1 + 10;
    shift = showTextWidth - (rclB.right - rclB.left);

    ShiftWindow(hDlg, 0, 0, shift);
    ShiftWindow(IDC_LIST1, hDlg, 0, 0, shift);
    ShiftWindow(IDC_GROUP_0, hDlg, 0, 0, shift);

    for (int i = 0; i < sizeof(nCtrlId) / sizeof(int); ++i)
        ShiftWindow(nCtrlId[i], hDlg, shift);

    ShiftWindow(IDC_STATIC_PHONE, hDlg, shift / 2);
}
```

```
//=====
BOOL CALLBACK ViewDlgProc(HWND hDlg, UINT uMsg, WPARAM wParam,
    LPARAM lParam) {

    static HWND hListBox, hComboFont;
    static HWND hEditInfoAddr, hEditInfoBday;
    HWND hCtrl;
    COLORREF editInfoColor;
    static HBRUSH hBrush;
    int nItems; // количество элементов в списке
    int iCurItem; // текущий выбранный элемент списка
    BOOL dlgYes;
    int i;
    char buf[10];

    switch (uMsg) {
        case WM_INITDIALOG:
            hListBox = GetDlgItem(hDlg, IDC_LIST1);
            hComboFont = GetDlgItem(hDlg, IDC_COMBO_FONT);
            hEditInfoAddr = GetDlgItem(hDlg, IDC_INFO_ADDR);
            hEditInfoBday = GetDlgItem(hDlg, IDC_INFO_BDAY);

            // Создание кисти для фона элементов управления
            editInfoColor = RGB(190, 255, 255);
            hBrush = CreateSolidBrush(editInfoColor);

            // Настройка формы диалога
            AdjustDialogSize(hDlg);

            // Инициализация комбинированного списка
            for (i = 0; i < sizeof(fontHeight) / sizeof(int); ++i) {
                sprintf(buf, "%d", fontHeight[i]);
                SendMessage(hComboFont, CB_ADDSTRING, 0, (LPARAM)buf);
            }
            SendDlgItemMessage(hDlg, IDC_COMBO_FONT, CB_SETCURSEL, iFont, 0);

            // Чтение "базы данных" из файла и инициализация списка
            fdat.open("notebook.dat", ios::in);
            if (!fdat) { fdat.clear(); return TRUE; }
            else while (1) {
                fdat.getline(item.dat.line, sizeof(Data));
                if (fdat.eof()) { fdat.clear(); break; }
                SendMessage(hListBox, LB_ADDSTRING, 0,
                           (LPARAM)item.dat.line);
            }
            fdat.close();
            return TRUE;

        case WM_CTLCOLORSTATIC:
            /* Текст из листинга 7.4 */
            break;

        case WM_COMMAND:
            switch (LOWORD(wParam))
            {
                case IDC_REC_ADD:
                    /* Текст из листинга 7.4 */
                    break;
            }
    }
}
```

продолжение ↗

Листинг 7.5 (продолжение)

```

        break;

    case IDC_REC_DELETE:
    /* Текст из листинга 7.4 */
        break;

    case IDC_LIST1:
    /* Текст из листинга 7.4 */
        break;

    case IDOK:
    /* Текст из листинга 7.4 */
        return TRUE;

    case IDCANCEL:
        EndDialog(hDlg, FALSE);
        return TRUE;

    case IDC_COMBO_FONT:
        switch (HIWORD(wParam)) {
            case CBN_SELCHANGE:
                // Выбираем используемый шрифт
                iFont = SendMessage(hComboFont, CB_GETCURSEL, 0, 0);
                // Настройка формы диалога
                AdjustDialogSize(hDlg);
                return TRUE;
            }
            break;
        }
        break;
    }
    return FALSE;
}

//=====
BOOL CALLBACK AddRecDiaProc(HWND hDlg, UINT uMsg,
    WPARAM wParam, LPARAM lParam)
{
    /* Текст из листинга 7.4 */
}
///////////

```

Обрабатывая уведомительное сообщение **CBN_SELCHANGE**, мы посыпаем комбинированному списку сообщение **CB_GETCURSEL**, чтобы узнать, какой элемент списка был выбран пользователем. Индекс выбранного шрифта сохраняется в глобальной переменной **iFont**. После этого вызывается новая функция **AdjustDialogSize**, которая не использовалась в предыдущем приложении. Она выполняет всю рутинную работу по настройке формы диалога. Во-первых, функция устанавливает выбранный шрифт в окне списка **hListBox**. Во-вторых, она изменяет размеры диалогового окна и окна списка **hListBox** с учетом метрик выбранного шрифта так, чтобы в окне списка отображались только первые два поля каждой записи. В-третьих, она изменяет размещение по горизонтали некоторых элементов управления, чтобы их позиция согласовывалась с шириной окна списка.

Для выполнения описанной работы функция **AdjustDialogSize** пользуется услугами новой перегруженной функции **ShiftWindow**. Прототипы функции **ShiftWindow**

объявлены в файле KWndPlut.h, а реализация перегруженных версий функции определена в файле KWndPlut.cpp.

Первая версия функции `ShiftWindow` с пятью параметрами предназначена для перемещения и изменения размеров окна верхнего уровня (*top-level window*). Дескриптор окна `hwnd` передается в качестве первого параметра. В нашей программе эта версия функции вызывается для изменения размеров диалогового окна `hDlg`.

Вторая версия функции `ShiftWindow` с шестью параметрами предназначена для перемещения и изменения размеров дочернего окна. Первый параметр этой функции имеет тип `HWND`. Он принимает дескриптор дочернего окна. Второй параметр позволяет задавать дескриптор родительского окна. В реализации функции учитывается, что функция `GetWindowRect` определяет позицию окна, выраженную в экранах координатах. В то же время функция `MoveWindow` работает с экранными координатами, если применяется к окну верхнего уровня, и с клиентскими координатами, если применяется к дочернему окну. Необходимое преобразование экранных координат к клиентским координатам достигается вызовом функции `ScreenToClient`. Данная версия функции в этом приложении не используется, но может быть применена, например, для немодальных диалоговых окон.

Третья версия функции `ShiftWindow` предназначена для перемещения и изменения размеров дочерних окон элементов управления. Она тоже принимает шесть параметров, но первый параметр имеет тип `int`. Первый параметр этой функции позволяет указывать идентификатор элемента управления, а второй параметр — дескриптор родительского окна.

На рис. 7.32 показан внешний вид работающей программы `MyNotebook2`, когда в окне `Combo box` выбран шрифт размером 24 логические единицы.

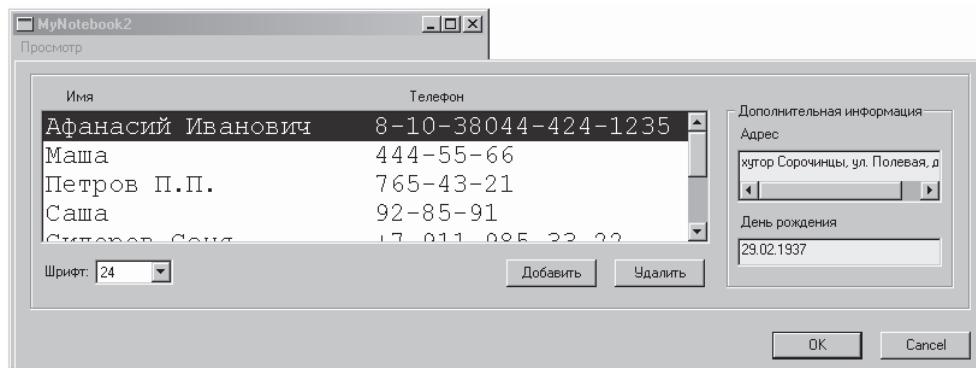


Рис. 7.32. Просмотр списка в приложении MyNotebook2. Установлен размер шрифта 24

Обратите внимание на то, что файлы `KWndPlut.h`, `KWndPlut.cpp` вы можете включать и в другие проекты. В этом случае помимо класса `KWnd` в вашем распоряжении оказывается также перегруженная функция `ShiftWindow`.

До сих пор мы изучали применение элементов управления, размещая их в модальном диалоговом окне. В принципе, подобным же образом они могут работать и в составе немодального диалогового окна. Однако создание немодального диалогового окна и работа с ним несколько отличаются от создания и обработки модального диалога.

Немодальный диалог

Немодальные диалоговые окна позволяют пользователю переключаться между диалоговым окном и окном, в котором оно было создано. Окна этого типа больше напоминают обычные всплывающие окна, которые могут создаваться программой. Немодальные диалоговые окна лучше использовать, когда пользователь работает с ними и с основным окном одновременно. Пожалуй, самые распространенные немодальные окна — это окна инструментов поиска *Find* и *Replace*, отображаемые программами обработки текстов. Пользователю может потребоваться оставить такой диалог в активном состоянии, если нужно выполнить несколько операций поиска и замены. При отображении этого окна можно также редактировать документ, в котором выполняется поиск или замена.

Иногда немодальное диалоговое окно создается в момент старта приложения и может оставаться на экране до окончания работы приложения.

Различия между модальными и немодальными окнами диалога

Мы уже знаем, что модальные диалоговые окна создаются при помощи функции *DialogBox*. Эта функция возвращает управление только после закрытия диалогового окна. Немодальные диалоговые окна создаются с помощью функции *CreateDialog*. Она принимает такие же параметры, что и функция *DialogBox*:

```
hDlgModeless = CreateDialog(hInst, MAKEINTRESOURCE(IDD_DLG), hWnd, DlgProc);
```

то есть функции передаются дескриптор экземпляра приложения, идентификатор шаблона диалога, дескриптор родительского окна и адрес диалоговой процедуры.

Различие состоит в том, что функция *CreateDialog* сразу возвращает дескриптор диалогового окна. Как правило, этот дескриптор хранится в глобальной переменной.

Определяя свойства шаблона диалогового окна, обязательно установите флагок *Visible* на вкладке *More Styles* окна *Dialog Properties*. Если этот флагок сброшен, то для появления окна на экране потребуется после вызова функции *CreateDialog* дополнительно вызвать функцию *ShowWindow*:

```
ShowWindow(hDlgModeless, SW_SHOW);
```

При сброшенном флагке *Visible* и отсутствии указанного вызова функции *ShowWindow* немодальный диалог вообще не появится на экране. Модальное диалоговое окно менее притязательно: оно появляется на экране и при сброшенном флагке *Visible*.

В отличие от сообщений для модальных диалоговых окон, направляемых системой непосредственно менеджеру диалогового окна, сообщения для немодального диалогового окна проходят через очередь сообщений программы. Поэтому цикл обработки сообщений в теле функции *WinMain* должен быть приведен к следующему виду:

```
while (GetMessage(&msg, NULL, 0, 0)) {  
    if (!IsDialogMessage(hModelessDlg, &msg)) {  
        TranslateMessage(&msg);  
        DispatchMessage(&msg);  
    }  
}
```

Функция `IsDialogMessage` определяет, относится ли сообщение, сохраненное в переменной `msg`, к диалоговому окну `hModelessDlg`. Если да, то функция обрабатывает это сообщение, отправляя его диалоговой процедуре и возвращая значение `TRUE`. В ином случае функция ничего не делает с сообщением и возвращает значение `FALSE`.

Сообщения, переданные на обработку диалоговой процедуре, не должны обрабатываться функциями `TranslateMessage` и `DispatchMessage`. Это обеспечивает условный оператор `if`, анализирующий код возврата функции `IsDialogMessage`. Заметим, что если немодальное диалоговое окно еще не создано, то дескриптор `hModelessDlg` должен быть равен нулю и в этом случае функция `IsDialogMessage` тоже возвращает значение `FALSE`.

Для закрытия немодального диалогового окна вместо функции `EndDialog` вызывается функция `DestroyWindow`. Если диалог закрывается, а приложение продолжает работать, то рекомендуется дескриптор `hModelessDlg` установить в нулевое значение.

Если немодальное диалоговое окно имеет заголовок, в котором размещается кнопка закрытия окна, то пользователь по привычке может попытаться закрыть диалог с помощью данной кнопки. Менеджер немодального диалогового окна не обрабатывает сообщение `WM_CLOSE`. Поэтому реакция приложения будет зависеть от решения программиста. Если вы считаете, что окно можно закрыть, то в диалоговую процедуру нужно добавить следующий код:

```
case WM_CLOSE:  
    DestroyWindow(hDlg);  
    hModelessDlg = 0;  
    break;
```

Это же сообщение будет обрабатываться в случае выбора команды `Close` (`Alt+F4`) в системном меню диалогового окна.

Пример использования немодального окна диалога

В приложении `ModelessDlg`, которое мы сейчас разрабатываем, используется немодальное диалоговое окно, содержащее три полосы прокрутки. При помощи этих полос пользователь может изменять цвет фона в клиентской области основного окна.

Создайте новый проект типа `Win32 Application` с именем `ModelessDlg`. Скопируйте в папку проекта файлы `KWndPlut.h`, `KWndPlut.cpp` из папки проекта `MyNotebook2`, после чего добавьте их в состав нового проекта.

Включите в приложение ресурс диалогового окна с идентификатором `IDD_MODELESS` и заголовком `Цвет фона`. На вкладке `Styles` окна `Dialog Properties` нужно выбрать стиль окна `Child`, а на вкладке `More Styles` установить флагок `Visible`. Остальные свойства окна оставьте со значениями по умолчанию.

Удалите с заготовки диалогового окна кнопки `OK` и `Cancel`.

Поместите на форму диалога надписи `Red`, `Green` и `Blue`, а под ними — элементы управления типа `Vertical Scroll Bar` (рис. 7.33).

Для полос прокрутки задайте, соответственно, идентификаторы `IDC_RED`, `IDC_GREEN`, `IDC_BLUE`. В окне свойств каждой полосы прокрутки должны быть установлены флагки `Visible` и `Tab stop`.

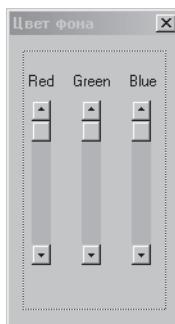


Рис. 7.33. Размещение элементов управления на форме диалога IDD_MODELESS

Добавьте к проекту файл ModelessDlg.cpp, текст которого приведен в листинге 7.6.

Листинг 7.6. Проект ModelessDlg

```
////////////////////////////////////////////////////////////////
// ModelessDlg.cpp
#include <windows.h>

#include "KWndPlut.h"
#include "resource.h"

enum UserMsg { UM_CHANGE = WM_USER+1 };

HWND hModelessDlg;
RECT rcWork; // прямоугольник рабочей области
int rgb[3]; // интенсивность для R-, G-, B-компонентов цвета

BOOL CALLBACK ModelessDlgProc(HWND, UINT, WPARAM, LPARAM);
void AdjustDlgPlace(HWND, HWND);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====================================================
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;

    KWnd mainWnd("ModelessDlg", hInstance, nCmdShow, WndProc,
        NULL, 100, 100, 400, 300);

    while (GetMessage(&msg, NULL, 0, 0)) {
        if (!IsDialogMessage(hModelessDlg, &msg)) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
    return msg.wParam;
}

//=====================================================
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
```

```
{  
    switch (uMsg) {  
  
        case WM_CREATE:  
            hModelessDlg = CreateDialog(GetModuleHandle(NULL),  
                MAKEINTRESOURCE(IDD_MODELESS), hWnd, ModelessDlgProc);  
            AdjustDlgPlace(hWnd, hModelessDlg);  
            break;  
  
        case WM_SIZE:  
            AdjustDlgPlace(hWnd, hModelessDlg);  
            break;  
  
        case UM_CHANGE:  
            // Изменение цвета фона основного окна  
            SetClassLong(hWnd, GCL_HBRBACKGROUND, (LONG)CreateSolidBrush(  
                RGB(rgb[0], rgb[1], rgb[2])));  
            InvalidateRect(hWnd, &rcWork, TRUE);  
            break;  
  
        case WM_DESTROY:  
            DestroyWindow(hModelessDlg);  
            PostQuitMessage(0);  
            break;  
  
        default:  
            return DefWindowProc(hWnd, uMsg, wParam, lParam);  
    }  
    return 0;  
}  
  
//=====  
BOOL CALLBACK ModelessDlgProc(HWND hDlg, UINT uMsg, WPARAM wParam,  
    LPARAM lParam) {  
  
    HWND hCtrl;  
    int iCtrlID, iIndex;  
    HWND hRValue = GetDlgItem(hDlg, IDC_RED);  
    HWND hGValue = GetDlgItem(hDlg, IDC_GREEN);  
    HWND hBValue = GetDlgItem(hDlg, IDC_BLUE);  
  
    switch (uMsg) {  
        case WM_INITDIALOG:  
            SetScrollRange(hRValue, SB_CTL, 0, 255, FALSE);  
            SetScrollPos(hRValue, SB_CTL, 0, FALSE);  
  
            SetScrollRange(hGValue, SB_CTL, 0, 255, FALSE);  
            SetScrollPos(hGValue, SB_CTL, 0, FALSE);  
  
            SetScrollRange(hBValue, SB_CTL, 0, 255, FALSE);  
            SetScrollPos(hBValue, SB_CTL, 0, FALSE);  
            return TRUE;  
  
        case WM_VSCROLL:  
            hCtrl = (HWND)lParam;  
            iCtrlID = GetDlgItemID(hCtrl);  
    }
```

продолжение ↗

Листинг 7.6 (продолжение)

```

iIndex = iCtrlID - IDC_RED;

switch(LOWORD(wParam)) {
    case SB_LINEUP:
        rgb[iIndex] = max(0, rgb[iIndex] - 1);
        break;
    case SB_LINEDOWN:
        rgb[iIndex] = min(255, rgb[iIndex] + 1);
        break;
    case SB_PAGEUP:
        rgb[iIndex] -= 15;
        break;
    case SB_PAGEDOWN:
        rgb[iIndex] += 15;
        break;
    case SB_THUMBTRACK:
        rgb[iIndex] = HIWORD(wParam);
        break;
}
SetScrollPos(hCtrl, SB_CTL, rgb[iIndex], TRUE);

// Сообщение родительскому окну об изменении цвета фона
SendMessage(GetParent(hDlg), UM_CHANGE, 0, 0);

break;
}
return FALSE;
}

//=====
void AdjustDlgPlace(HWND hParent, HWND hDlg) {

RECT rcParent, rcDlg;

// Установка размеров окна hDlg
GetClientRect(hParent, &rcParent);
GetWindowRect(hDlg, &rcDlg);
int width = rcDlg.right - rcDlg.left;
int height = rcDlg.bottom - rcDlg.top;
int dH = rcParent.bottom - height;
ShiftWindow(hDlg, hParent, 0, 0, 0, dH);
// Установка размеров рабочей области для окна hParent
rcWork = rcParent;
rcWork.left += width;
}
///////////

```

Немодальное диалоговое окно `hModelessDlg` с шаблоном `IDD_MODELESS` создается в оконной процедуре `WndProc` при помощи вызова функции `CreateDialog`, который размещается в блоке обработки сообщения `WM_CREATE`.

После этого вызывается функция `AdjustDlgPlace`, которая регулирует высоту диалогового окна таким образом, чтобы она совпадала с высотой клиентской области главного окна. Для решения этой подзадачи вызывается вторая версия перегруженной функции `ShiftWindow`. Кроме того, функция `AdjustDlgPlace` вычисляет прямоугольник рабочей области главного окна, к которой относится вся клиентская область, за исключением участка, закрытого диалоговым окном. Этот

прямоугольник `rcWork` используется оконной процедурой `WndProc` при вызове функции `InvalidateRect`, чтобы ограничить обновляемый регион только видимой частью клиентской области.

Диалоговая процедура `ModelessDlgProc` осуществляет обработку сообщений от полос прокрутки `IDC_RED`, `IDC_GREEN` и `IDC_BLUE`. Элементы управления типа `Vertical Scroll Bar` посыпают родительскому окну такие же сообщения, как и вертикальная полоса прокрутки главного окна приложения. Пример обработки этих сообщений мы уже рассматривали в приложении `TextViewer` (см. листинг 2.2).

В результате обработки сообщений от полос прокрутки изменяются значения глобального массива `rgb`, в котором хранятся текущие величины для RGB-компонентов цвета. Если изменилось хотя бы одно из значений, то диалоговая процедура при помощи функции `SendMessage` посыпает пользовательское сообщение `UM_CHANGE`, адресуя его оконной процедуре родительского окна. Константа `UM_CHANGE` является значением перечисляемого типа `UserMsg`, который определен оператором `enum`.

Оконная процедура `WndProc` содержит код обработки сообщения `UM_CHANGE`. В этом блоке вызывается функция `SetClassLong` для изменения цвета фона главного окна приложения. Чтобы окно перерисовалось, нужно также вызвать функцию `InvalidateRect`. Обратите внимание на то, что если в качестве второго аргумента функции `InvalidateRect` передать `NULL`, то при перерисовке возникнет неприятное мерцание фона диалогового окна.

Уничтожение диалогового окна происходит в блоке обработки сообщения `WM_DESTROY`.

Последнее замечание: если для диалогового окна `IDD_MODELESS` на вкладке `Styles` окна `Dialog Properties` задать стиль `PopUp`, то в теле функции `AdjustDlgPlace` для установки позиции окна следует вызывать *первую* версию перегруженной функции `ShiftWindow`:

```
ShiftWindow(hDlg, 0, 0, 0, dH);
```

Окно работающего приложения `ModelessDlg` показано на рис. 7.34.

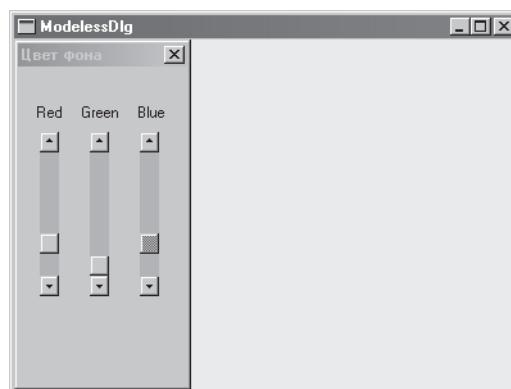


Рис. 7.34. Приложение ModelessDlg

К сожалению, черно-белый рисунок не может передать всей красоты нежнейшего светло-салатового цвета в клиентской области окна, который установлен в настоящий момент элементами управления диалогового окна `Цвет фона`.

Окно сообщений

Окно сообщений, которое вызывается функцией `MessageBox`, является простейшим типом диалогового окна. В предыдущих проектах эта функция уже неоднократно применялась, но при этом были использованы далеко не все ее возможности.

Функция `MessageBox` позволяет создавать, отображать и выполнять различные действия с окном сообщения. Окно сообщения содержит текст, определяемый приложением, и заголовок, а также любое сочетание предопределенных пиктограмм и кнопок.

Функция `MessageBox` чаще всего используется для сообщений об ошибках и предупреждающих сообщений. Иногда ее применяют для отладочной печати данных в качестве замены оператора `printf` в консольных приложениях. Также окна сообщения могут использоваться в качестве функций-заглушек при разработке программы. Например, окна сообщений могут вызываться для пунктов меню, которые еще не реализованы.

Функция имеет следующий прототип:

```
int MessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);
```

Параметр `hWnd` принимает дескриптор родительского окна. Если этот параметр равен `NULL`, то окно сообщения не имеет окна-владельца.

Параметр `lpText` является указателем на С-строку, содержащую текст, который должен быть отображен в окне. Вы можете использовать в составе строки управляющие символы `\n`, если хотите сделать текст многострочным.

Параметр `lpCaption` является указателем на С-строку, которая отображается в заголовке диалогового окна. Если этот параметр равен `NULL`, то применяется заданный по умолчанию заголовок `Error`.

Параметр `uType` определяет содержимое и правила поведения диалогового окна. Его значением может быть комбинация флагов из групп флагов, которые приведены в табл. 7.13–7.15.

Для указания состава отображаемых кнопок используйте одно из значений, приведенных в табл. 7.13.

Таблица 7.13. Флаги, определяющие состав кнопок

Флаг	Описание
<code>MB_CANCELTRYCONTINUE</code>	Окно сообщений содержит кнопки <code>Cancel</code> , <code>Try Again</code> , <code>Continue</code>
<code>MB_HELP</code>	Добавляет кнопку <code>Help</code> в окно сообщений. Когда пользователь нажимает эту кнопку, система посыпает сообщение <code>WM_HELP</code> окну-владельцу
<code>MB_OK</code>	Окно сообщений содержит единственную кнопку <code>OK</code> . Это значение применяется по умолчанию
<code>MB_OKCANCEL</code>	Окно сообщений содержит кнопки <code>OK</code> и <code>Cancel</code>
<code>MB_RETRYCANCEL</code>	Окно сообщений содержит кнопки <code>Retry</code> и <code>Cancel</code>
<code>MB_YESNO</code>	Окно сообщений содержит кнопки <code>Yes</code> и <code>No</code>
<code>MB_YESNOCANCEL</code>	Окно сообщений содержит кнопки <code>OK</code> , <code>No</code> и <code>Cancel</code>

Чтобы указать отображаемую пиктограмму, нужно выбрать одно из значений, приведенных в табл. 7.14.

Таблица 7.14. Флаги, определяющие отображаемую пиктограмму

Флаг	Описание
MB_ICONEXCLAMATION, MB_ICONWARNING	Пиктограмма с изображением восклицательного знака
MB_ICONINFORMATION, MB_ICONASTERISK	Пиктограмма с изображением буквы «i»
MB_ICONQUESTION	Пиктограмма с изображением вопросительного знака
MB_ICONSTOP, MB_ICONERROR, MB_ICONHAND	Пиктограмма с изображением знака «Стоп»

Чтобы указать «вид модальности», используйте одно из значений, приведенных в табл. 7.15.

Таблица 7.15. Флаги, определяющие вид модальности окна сообщений

Флаг	Описание
MB_APPLMODAL	Пользователь должен щелкнуть на одной из кнопок в окне сообщения, прежде чем сможет продолжить работу с окном, указанным в параметре hWnd. Пока окно сообщений не закрыто, окно hWnd и все его дочерние окна недоступны. Однако окна стиля «рорир», открытые ранее в данном приложении, доступны для работы с ними. Пользователь также может переключиться в окна других приложений и работать с ними
MB_SYSTEMMODAL	Поведение окна сообщений такое же, как и с флагом MB_APPLMODAL, за исключением того, что окно сообщений имеет стиль WS_EX_TOPMOST, то есть располагается всегда поверх других окон. Рекомендуется использовать этот флаг для сообщений о серьезных ошибках или проблемах, требующих немедленной реакции пользователя
MB_TASKMODAL	Поведение окна сообщений такое же, как и с флагом MB_APPLMODAL, за исключением того, что, если параметр hWnd равен NULL, недоступными будут все окна данного приложения. Этот флаг используется обычно в тех ситуациях, когда дескриптор окна-владельца неизвестен (например, при разработке библиотек)

Есть и другие, реже применяемые флаги, которые перечислены в справочных материалах MSDN.

Если функция завершается успешно, она возвращает одно из следующих значений:

Возвращаемое значение	Описание
IDABORT	Была выбрана кнопка Abort
IDCANCEL	Была выбрана кнопка Abort или нажата клавиша Esc
IDCONTINUE	Была выбрана кнопка Continue
IDIGNORE	Была выбрана кнопка Ignore
IDNO	Была выбрана кнопка No
IDOK	Была выбрана кнопка OK
IDRETRY	Была выбрана кнопка Retry
IDTRYAGAIN	Была выбрана кнопка Try Again
IDYES	Была выбрана кнопка Yes

Если произошла какая-то ошибка, то функция возвращает нулевое значение.

Технику применения функции MessageBox довольно легко понять, поэтому специальные примеры в этой главе не приводятся.

Диалоговые окна общего пользования

Одна из важнейших целей, поставленных перед разработчиками системы Windows, заключалась в том, чтобы способствовать разработке приложений со стандартизованным интерфейсом пользователя. Для многих общепринятых пунктов меню это было сделано довольно быстро. Почти все разработчики программного обеспечения стали использовать последовательность команд **File ▶ Open** (Файл ▶ Открыть) для вызова диалогового окна, при помощи которого пользователь мог открыть файл. Однако сами окна диалога для открытия файла часто были не очень похожи друг на друга.

Начиная с версии Windows 3.1, решение этой проблемы было реализовано в виде *библиотеки диалоговых окон общего пользования* (*Common Dialog Box Library*). Библиотека содержит несколько функций, которые вызывают стандартные окна диалога для открытия и сохранения файлов, поиска и замены, выбора цвета, выбора шрифта и другие стандартные окна.

Для каждой функции, создающей и вызывающей стандартное диалоговое окно, в заголовочном файле `commndl.h` определен соответствующий тип структуры. Чтобы работать со стандартным диалогом, вы должны определить подходящую структурную переменную и инициализировать ее. После этого вызывается функция стандартного диалога, которой передается адрес этой структурной переменной. Когда пользователь закрывает окно диалога, вызванная функция возвращает управление программе. В этот момент вы можете получить требуемую информацию из указанной структурной переменной.

В демонстрационном приложении *CommonDialogs*, которое мы сейчас разрабатываем, иллюстрируется, как можно использовать стандартные диалоговые окна **Open** (Открыть), **Save As** (Сохранить как), **Color** (Цвет) и **Font** (Шрифт).

Создайте новый проект с именем *CommonDialogs*.

Добавьте к приложению ресурс меню с идентификатором **IDR_MENU1**. Главное меню должно содержать пункты с атрибутами, указанными в табл. 7.16.

Таблица 7.16. Пункты главного меню

Имя пункта	Тип пункта	Идентификатор
&File	Подменю	—
&Background color	Команда	IDM_BKGR_COLOR
&Text color	Команда	IDM_TEXT_COLOR
&Choose font	Команда	IDM_CHOOSE_FONT

Подменю **File** должно содержать пункты с атрибутами, приведенными в табл. 7.17.

Таблица 7.17. Пункты подменю File

Имя пункта	Тип пункта	Идентификатор
&Open...	Команда	IDM_OPEN
Save &As...	Команда	IDM_SAVE_AS
—	SEPARATOR	—
E&xit	Команда	IDM_EXIT

Добавьте в состав проекта файл *CommonDialogs.cpp* с текстом, приведенным в листинге 7.7.

Листинг 7.7. Проект CommonDialogs

```
////////////////////////////////////////////////////////////////
// CommonDialogs.cpp
#include <windows.h>
#include <stdio.h>
#include "KWnd.h"
#include "resource.h"

#define ESC_OF "Отказ от выбора или ошибка выполнения функции"

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====================================================
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("CommonDialogs", hInstance, nCmdShow, WndProc,
        MAKEINTRESOURCE(IDR_MENU1), 100, 100, 400, 300);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

//=====================================================
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    PAINTSTRUCT ps;
    RECT rcClient;
    BOOL success;
    static COLORREF textColor;

    // Переменные для стандартных диалогов "Open", "Save As"
    static OPENFILENAME ofn;
    static char szFile[MAX_PATH];

    // Переменные для стандартного диалога "Color"
    static CHOOSECOLOR cc; // common dialog box structure
    static COLORREF acrCustClr[16]; // array of custom colors

    // Переменные для стандартного диалога "Font"
    static CHOOSEFONT chf;
    static HFONT hFont;
    static LOGFONT lf;

    switch (uMsg)
    {
    case WM_CREATE:
        // Инициализация структуры ofn
        ofn.lStructSize = sizeof(OPENFILENAME);
        ofn.hwndOwner = hWnd;
        ofn.lpstrFile = szFile;
        ofn.nMaxFile = sizeof(szFile);

        // Инициализация структуры cc
        cc.dwSize = sizeof(cc);
        cc.hwndOwner = hWnd;
        cc.hInst = hInstance;
        cc.lpszTitle = "Выбор цвета";
        cc.lpfnHook = NULL;
        cc.nColor = RGB(255, 0, 0);
        cc.nCustomCrs = 16;
        cc.acrCustClr = acrCustClr;
```

продолжение ↗

Листинг 7.7 (продолжение)

```
cc.lStructSize = sizeof(CHOOSECOLOR);
cc.hwndOwner = hWnd;
cc.lpCustColors = (LPDWORD) acrCustClr;
cc.Flags = CC_FULLSCREEN | CC_RGBINIT;

// Инициализация структуры chf
chf.lStructSize = sizeof(CHOOSEFONT);
chf.hwndOwner = hWnd;
chf.lplgLogFont = &lf;
chf.Flags = CF_SCREENFONTS | CF_INITTOLOGFONTSTRUCT;
chf.nFontType = SIMULATED_FONTTYPE;
break;

case WM_COMMAND:
switch (LOWORD(wParam))
{
case IDM_OPEN:
strcpy(szFile, "");
success = GetOpenFileName(&ofn);
if (success)
MessageBox(hWnd, ofn.lpszFile, "Открывается файл:", MB_OK);
else
MessageBox(hWnd, ESC_OF"GetOpenFileName",
"Отказ от выбора или ошибка", MB_ICONWARNING);
break;

case IDM_SAVE_AS:
strcpy(szFile, "");
success = GetSaveFileName(&ofn);
if (success)
MessageBox(hWnd, ofn.lpszFile,
"Файл сохраняется с именем:", MB_OK);
else
MessageBox(hWnd, ESC_OF"GetSaveFileName",
"Отказ от выбора или ошибка", MB_ICONWARNING);
break;

case IDM_BKGR_COLOR:
if (ChooseColor(&cc))
SetClassLong(hWnd, GCL_HBRBACKGROUND,
(LONG>CreateSolidBrush(cc.rgbResult));
break;

case IDM_TEXT_COLOR:
if (ChooseColor(&cc)) textColor = cc.rgbResult;
break;

case IDM_CHOOSE_FONT:
if(ChooseFont(&chf)) hFont = CreateFontIndirect(chf.lplgLogFont);
break;

case IDM_EXIT:
SendMessage(hWnd, WM_DESTROY, 0, 0);
break;

default:
break;
}
```

```
InvalidateRect(hWnd, NULL, TRUE);
break;

case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);
    SetBkMode(hDC, TRANSPARENT);
    SetTextColor(hDC, textColor);
    if (hFont)
        DeleteObject(SelectObject(hDC, hFont));

    GetClientRect(hWnd, &rcClient);
    DrawText(hDC, "Common Dialogs", -1, &rcClient,
             DT_SINGLELINE | DT_CENTER | DT_VCENTER);
    EndPaint(hWnd, &ps);
break;

case WM_DESTROY:
    PostQuitMessage(0);
break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
////////////////////////////////////////////////////////////////
```

В приложении используются предопределенные структуры трех типов. Структура **OPENFILENAME** применяется для создания стандартных диалогов Open и Save As. Структура **CHOOSECOLOR** позволяет вызывать диалоговое окно Color, а структура **CHOOSEFONT** предназначена для работы со стандартным диалоговым окном Font. Соответствующие структурные переменные **ofn**, **cc** и **chf** объявлены со спецификатором **static**. Это решает две проблемы: инициализацию нулевым значением тех полей структур, для которых можно использовать значения по умолчанию, и сохранение значений полей между двумя вызовами функции **WndProc**.

Инициализация переменных **ofn**, **cc** и **chf** осуществляется в блоке обработки сообщения **WM_CREATE**.



Рис. 7.35. Приложение CommonDialogs

Стандартные диалоговые окна создаются и вызываются следующими функциями:

Стандартное диалоговое окно	Функция
Open	GetOpenFileName
Save As	GetSaveFileName
Color	ChooseColor
Font	ChooseFont

Приложение выводит в центре окна текстовую строку Common Dialogs (рис. 7.35). Пользователь может выбрать любой цвет фона и любой цвет текста, а также любой шрифт из установленных в системе шрифтов, с любым возможным размером и начертанием шрифта.

На рисунке выбран шрифт Monotype Corsiva с размером 36 логических единиц.

Протестируйте приложение, выполняя различные команды, в том числе и для подменю File. Обратите внимание на то, что удобный, а главное, стандартизованный интерфейс пользователя в нашей программе был реализован минимальными средствами.

8

Элементы управления общего пользования

По мере развития операционной системы Windows ее пользовательский интерфейс непрерывно улучшался. Так, начиная с Windows 95, в системе было определено несколько новых элементов, называемых *общими элементами управления*. Чтобы сделать эти элементы доступными для разработчиков программного обеспечения, Microsoft разработала *библиотеку элементов управления общего пользования* (*common control library*). Новые элементы управления дополняют базовые элементы управления, рассмотренные в предыдущей главе, и придают программам более совершенный вид. Библиотека элементов управления общего пользования реализована в виде динамически загружаемой библиотеки comctl32.dll.

Элементы управления общего пользования можно разделить на четыре категории, в которые входят элементы управления главного окна, составные диалоговые элементы управления, элементы управления Windows Explorer и иные дополнительные элементы управления. В табл. 8.1–8.4 приведены краткие описания этих элементов.

Таблица 8.1. Элементы управления главного окна

Элемент управления	Описание
Toolbar (Панель инструментов)	Состоит из кнопок быстрого доступа
Tooltip (Окно подсказки)	Показывает текст во всплывающем окне
Status bar (Строка состояния)	Информационная строка, обычно размещаемая в нижней части окна приложения

Таблица 8.2. Составные диалоговые элементы управления

Элемент управления	Описание
Tab control (Закладки)	Имитация закладок в записной книжке. При выборе одной из закладок на экране отображается связанное с ней диалоговое окно
Property sheet (Набор страниц свойств)	Диалоговое окно с набором страниц, аналогичных закладкам, содержащее кроме этого общие для всех страниц кнопки
Property page (Страница свойств)	Диалоговое окно, используемое как страница в элементе управления Property sheet

Таблица 8.3. Элементы управления Windows Explorer

Элемент управления	Описание
Tree view (Дерево просмотра)	Отображает иерархически структурированный список (левая панель окна программы Windows Explorer)
List view (Список просмотра)	Отображает список элементов, идентифицируемых пиктограммами и текстовыми данными (правая панель окна программы Windows Explorer)

Таблица 8.4. Другие элементы управления

Элемент управления	Описание
Animation (Анимационное изображение)	Воспроизводит анимационную последовательность для индикации длительной операции
Header (Заголовок списка просмотра)	Отображает горизонтальные заголовки для столбцов и используется совместно с элементом List view
Image list (Список изображений)	Элемент управления для хранения набора растровых изображений, не являющийся отдельным окном
Progress bar (Индикатор процесса)	Элемент управления, который отображает динамику длительной операции в виде процентного соотношения выполненной части задачи
Rich edit (Усовершенствованный редактор)	Редактор, поддерживающий множество шрифтов и базовые возможности OLE-контейнера
Slider (Регулятор)	Ползунок, перемещаемый в пределах шкалы (разновидность полосы прокрутки)
Spin (Счетчик или стрелки)	Полоса прокрутки, состоящая из двух кнопок со стрелками, для увеличения или уменьшения на единицу целого числа, находящегося в «приятельском» окне редактирования

Основы применения

Большинство элементов управления общего пользования реализовано в виде окна соответствующего предопределенного класса. С этой точки зрения элементы управления общего пользования похожи на базовые элементы управления. И те и другие элементы могут быть созданы вызовом функции `CreateWindow`, которой передаются конкретные флаги стиля класса. И те и другие элементы управленияются специфичными для данного класса сообщениями. Оба типа элементов управления посыпают уведомляющие сообщения родительскому окну, информируя его обо всех происходящих событиях.

Разница между базовыми элементами управления и элементами управления общего пользования состоит в типе посыпаемых уведомительных сообщений. Базовые элементы управления посыпают сообщения `WM_COMMAND`, а элементы управления общего пользования почти всегда посыпают сообщения `WM_NOTIFY`.

Инициализация библиотеки

Чтобы использовать в приложении какой-либо элемент управления общего пользования, сначала нужно вызвать функцию `InitCommonControlsEx`, которая

регистрирует оконные классы элементов управления. Функция имеет следующий прототип:

```
BOOL InitCommonControlsEx(LPINITCOMMONCONTROLSEX lpInitCtrls);
```

В параметре `lpInitCtrls` передается адрес структурной переменной типа `INITCOMMONCONTROLSEX`, содержащей информацию о том, какие классы элементов управления должны быть зарегистрированы.

Структура `INITCOMMONCONTROLSEX` имеет следующее определение:

```
typedef struct tagINITCOMMONCONTROLSEX {
    DWORD dwSize; // размер структуры в байтах
    DWORD dwICC; // флаги загрузки классов из DLL
} INITCOMMONCONTROLSEX, *LPINITCOMMONCONTROLSEX;
```

Второй параметр может принимать одно или несколько значений, перечисленных в табл. 8.5.

Таблица 8.5. Некоторые из возможных флагов для функции `InitCommonControlsEx`

Флаг	Загружаются оконные классы для элементов управления
<code>ICC_ANIMATE_CLASS</code>	<code>animate</code>
<code>ICC_BAR_CLASSES</code>	<code>toolbar, status bar, slider, tooltip</code>
<code>ICC_LISTVIEW_CLASSES</code>	<code>list view, Header</code>
<code>ICC_PROGRESS_CLASS</code>	<code>progress bar</code>
<code>ICC_TAB_CLASSES</code>	<code>tab, tooltip</code>
<code>ICC_TREEVIEW_CLASSES</code>	<code>tree view, tooltip</code>
<code>ICC_UPDOWN_CLASS</code>	<code>up-down</code>
<code>ICC_WIN95_CLASSES</code>	<code>animate, header, hot key, list view, progress bar, status bar, tab, tooltip, toolbar, slider, tree view, up-down</code>

Полный список флагов можно найти в справочных материалах MSDN.

Описание функции `InitCommonControlsEx` вместе с другими описаниями, необходимыми для использования библиотеки, находятся в файле `commctrl.h`. Этот файл не входит в группу файлов, ссылки на которые помещены в файле `windows.h`. Поэтому в начале любого исходного файла, содержащего вызовы функций данной библиотеки, необходимо поместить следующую директиву:

```
#include <commctrl.h>
```

Помимо этого следует указать компоновщику расположение библиотечного файла `comctl32.lib`. Если применяется среда разработки Visual Studio 6.0, то добавьте имя этой библиотеки в текстовое поле `Object/library modules` диалогового окна настроек проекта, которое вызывается при помощи команды меню `Project ▶ Settings... ▶ Link`.

Если вы забудете это сделать, то компоновщик выведет сообщение следующего вида:

```
error LNK2001: unresolved external symbol __imp_InitCommonControls@0
```

Учтите, что если вы забудете вызвать функцию `InitCommonControlsEx`, то ни компилятор, ни компоновщик этого «не заметят», но ваша программа, возможно, будет вести себя странным образом. Например, диалоговое окно, в котором был размещен элемент управления общего пользования, может вообще не появиться на экране. Так что будьте внимательны!

Элемент управления Rich edit из-за его сложности и большого размера располагается в его собственной динамически подключаемой библиотеке riched32.dll. Для работы с усовершенствованным редактором нужно загрузить эту библиотеку при помощи вызова функции LoadLibrary:

```
LoadLibrary("riched32.dll");
```

а в начале файлов, использующих функции или константы из данной библиотеки, поместить директиву

```
#include <richedit.h>
```

Создание элементов управления общего пользования

Наиболее традиционным способом создания элемента управления общего пользования является вызов функции CreateWindow или CreateWindowEx. Например, приведенный вызов создает панель инструментов:

```
HWND hwndToolBar = CreateWindow(TOOLBARCLASSNAME, NULL, WS_CHILD |  
    WS_VISIBLE | WS_BORDER, 0, 0, 16, 16, hwndParent, (HMENU)1, hInst, 0);
```

Имя оконного класса TOOLBARCLASSNAME здесь задается без кавычек, поскольку это именованная константа, определение которой зависит от набора символов, выбранного при построении программы. Для набора символов ANSI имя TOOLBARCLASSNAME заменяется строковой константой ToolbarWindow32, для набора символов UNICODE — строковой константой L"ToolbarWindow32". Имена других оконных классов элементов управления общего пользования определяются аналогично.

Обычно элементы управления общего пользования создаются как дочерние окна, что определяется флагом WS_CHILD и передачей дескриптора родительского окна hwndParent.

Альтернативой вызову функции CreateWindow является вызов специализированной функции создания элемента управления, которая в то же время может выполнять некоторую стандартную инициализацию. Например, панель инструментов может быть создана функцией CreateToolbarEx.

Для некоторых элементов управления соответствующие оконные классы не определены. Такие элементы управления могут быть созданы только с помощью специализированных функций. В табл. 8.6 приведены сведения об оконных классах и специализированных функциях создания для элементов управления общего пользования.

Таблица 8.6. Оконные классы и специализированные функции создания для элементов управления общего пользования

Элемент управления	Оконный класс	Функция создания
Toolbar	TOOLBARCLASSNAME	CreateToolbarEx
Tooltip	TOOLTIPS_CLASS	Нет
Status bar	STATUSCLASSNAME	CreateStatusWindow
Tab control	WC_TABCONTROL	Нет
Property sheet	Нет	PropertySheet

Элемент управления	Окноный класс	Функция создания
Property page	Нет	CreatePropertySheetPage
Tree view	WC_TREEVIEW	Нет
List view	WC_LISTVIEW	Нет
Animation	ANIMATE_CLASS	Нет
Header	WC_HEADER	Нет
Image list	Нет	ImageList_Create
Progress bar	PROGRESS_CLASS	Нет
Rich edit	"RichEdit" (ANSI) или L"RichEdit" (UNICODE)	Нет
Slider	TRACKBAR_CLASS	Нет
Spin	UPDOWN_CLASS	CreateUpDownControl

Стили элементов управления общего пользования

Независимо от способа создания — с помощью `CreateWindow` или специализированной функции — при вызове соответствующей функции необходимо задать набор стилей элемента управления.

Флаги стилей можно разделить на следующие четыре категории: флаги основного стиля окна (с префиксом `WS_`)¹, флаги расширенного стиля окна (`WS_EX_`)², флаги основного стиля элемента управления общего пользования (`CCS_`) и флаги стиля, специфичные для конкретного элемента управления.

Библиотека элементов управления общего пользования поддерживает набор значений стиля с префиксом `CCS_`. Они применяются для панелей инструментов, строк состояния и заголовков списка просмотра. В табл. 8.7 перечислены эти стили с кратким описанием их назначения.

Таблица 8.7. Флаги основного стиля элемента управления общего пользования

Флаг	Эффект
<code>CCS_ADJUSTABLE</code>	Состав кнопок панели инструментов и их последовательность могут изменяться пользователем
<code>CCS_BOTTOM</code>	Элемент управления располагается внизу родительского окна. Этот стиль установлен по умолчанию для элемента управления <code>Status bar</code>
<code>CCS_NODIVIDER</code>	Между элементом управления и родительским окном нет разделятельной линии
<code>CCS_NOMOVEY</code>	Элемент управления не может быть растянут или сдвинут по вертикали
<code>CCS_NORESIZE</code>	Размер и позиция элемента управления фиксированы
<code>CCS_TOP</code>	Элемент управления располагается вверху родительского окна. Этот стиль установлен по умолчанию для элемента управления <code>Tool bar</code>

Флаги стиля, специфичные для конкретных элементов управления, имеют уникальные префиксы, которые приведены в табл. 8.8.

¹ Флаги основного стиля окна приведены в табл. 1.8 (глава 1).

² Флаги расширенного стиля окна приведены в табл. 1.9 (глава 1).

Таблица 8.8. Префиксы флагов стиля, специфичных для конкретных элементов управления

Элемент управления	Префикс флага стиля
Toolbar	TBSTYLE_
Tooltip	TTS_
Status bar	SBARS_
Tab control	TCS_
Property sheet	Нет
Property page	Нет
Tree view	TVS_
List view	LVS_
Animation	ACS_
Header	HDS_
Image list	Нет
Progress bar	Нет
Rich edit	ES_
Slider	TBS_
Spin	UDS_

Обмен сообщениями

После создания элемента управления общего пользования приложение управляет его действиями, посыпая необходимые сообщения при помощи функции `SendMessage`. Для каждого типа элемента управления существуют свои специфические сообщения.

Например, для добавления элемента в дерево просмотра (*Tree view*) с дескриптором `hwndTV` необходимо послать элементу управления сообщение `TVM_INSERTITEM`. Это можно реализовать следующим вызовом функции `SendMessage`:

```
hItem = (HTREEITEM)SendMessage(hwndTV, TVM_INSERTITEM, 0,
    (LPARAM)(LPTV_INSERTSTRUCT) &tvis);
```

Альтернативой вызовам функции `SendMessage` является использование набора макросов, определенных в файле `commctrl.h`. Например, добавление элемента в дерево можно реализовать, используя макрос `TreeView_InsertItem`:

```
hItem = TreeView_InsertItem (hwndTV, &tvis);
```

Как видно, текст с макросом выглядит значительно проще для чтения.

К сожалению, файлы Win32 содержат определения макросов не для всех элементов управления общего пользования. В файле `commctrl.h` находятся определения макросов только для элементов управления *Tab control*, *Tree view*, *List view*, *Animation* и *Header*. Файл `prsht.h` содержит определения макросов для элемента управления *Property sheet*.

Как и базовые элементы управления, элементы управления общего пользования посыпают своему родительскому окну уведомляющие сообщения, содержащие информацию о произошедших событиях. Но если базовые элементы управления используют сообщение `WM_COMMAND`, то элементы управления общего пользования обычно посыпают уведомления при помощи сообщений `WM_NOTIFY`.

Однако не все уведомления реализуются подобным образом. Например, панель инструментов, использующая сообщение WM_NOTIFY для большинства уведомлений, посыпает сообщение WM_COMMAND, когда пользователь нажимает одну из кнопок. Дело в том, что панель инструментов обычно дублирует команды меню, поэтому имеющийся код обработки сообщений WM_COMMAND от команд меню будет одновременно обрабатывать и сообщения от кнопок панели инструментов.

Еще одно исключение составляют полосы прокрутки элементов управления Slider и Spin, которые посыпают сообщение WM_VSCROLL или WM_HSCROLL.

Хотя каждый элемент управления общего пользования имеет свой собственный набор кодов уведомления, существует набор уведомлений, общий для всех элементов. Эти уведомления приведены в табл. 8.9.

Таблица 8.9. Общие уведомления

Код уведомления	Описание
NM_CLICK	Пользователь сделал щелчок левой кнопкой мыши
NM_DBCLK	Пользователь сделал двойной щелчок левой кнопкой мыши
NM_KILLFOCUS	Элемент управления потерял фокус ввода
NM_OUTOFMEMORY	Ошибка «Не хватает памяти»
NM_RCLICK	Пользователь сделал щелчок правой кнопкой мыши
NM_RDBLCLK	Пользователь сделал двойной щелчок правой кнопкой мыши
NM_RETURN	Пользователь нажал клавишу Enter
NM_SETFOCUS	Элемент управления получил фокус ввода

Не все элементы управления обязательно посыпают каждое из этих уведомляющих сообщений. Например, набор закладок (Tab control) не посыпают уведомлений об изменении фокуса ввода.

Перед использованием каждого элемента управления общего пользования следует разобраться с тем, какие уведомляющие сообщения он посыпает.

Из-за ограниченного объема книги мы рассмотрим приемы работы только со следующими элементами управления общего пользования: Toolbar, Tooltip, Status bar, Progress bar, Slider и Spin.

Элементы управления главного окна

В оформлении главного окна приложения часто применяются следующие элементы управления общего пользования: панели инструментов, окна подсказки и строки состояния. На рис. 8.1 показано, как выглядят эти элементы управления в основном окне MS Visual Studio 6.0.

Панель инструментов

Панель инструментов — это дочернее окно, обычно расположено под меню приложения и содержащее одну или несколько кнопок. Когда пользователь щелкает мышью на кнопке панели инструментов, она посыпает сообщение WM_COMMAND своему родительскому окну. Традиционно кнопки панели инструментов соответствуют некоторым пунктам в меню приложения, обеспечивая более удобный

способ доступа пользователя к командам меню. Следует отметить, что в простых приложениях меню может отсутствовать, а все необходимые команды могут быть реализованы с помощью панели инструментов.

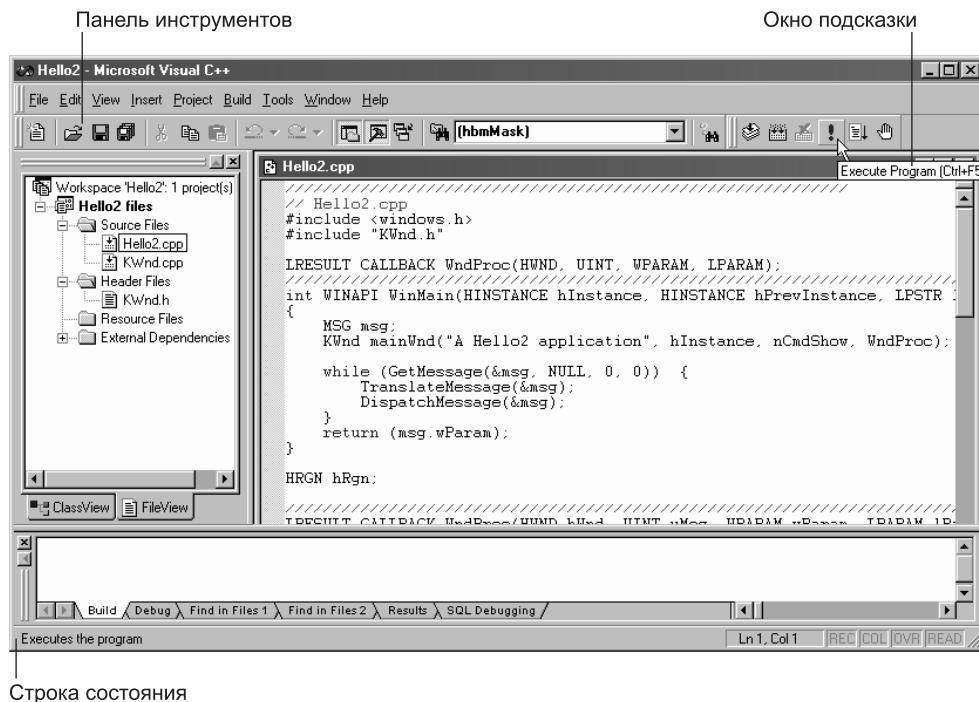


Рис. 8.1. Панель инструментов, строка состояния и окно подсказки

Кнопки панели инструментов сами по себе не являются окнами. Они реализованы как графические объекты, нарисованные на поверхности окна панели инструментов. Изображения на кнопках поясняют их назначение. Иногда помимо изображения кнопка может содержать текстовую метку, расположенную или правее, или ниже картинки.

Панель инструментов устанавливает одинаковые размеры всех кнопок, а в случае наличия текстовых меток определяет размеры так, чтобы разместить самый длинный текст. Поэтому для текстовых меток следует выбирать короткие строки, иначе кнопки могут получиться слишком большими. В большинстве случаев кнопки на панелях инструментов содержат только растровые изображения, а назначение кнопок поясняется с помощью всплывающих окон подсказок.

Чтобы сделать интерфейс панели инструментов более удобным для пользователя, кнопки часто объединяют в группы. Кнопки, объединенные в группу, следуют одна за другой, а между группами остается небольшой промежуток. Такие промежутки реализуются при помощи кнопок стиля TBSTYLE_SEP, называемых *кнопками-разделителями*.

Кроме кнопок, панель инструментов может содержать и другие дочерние окна элементов управления, такие, как, например, комбинированный список (*combo box*). Встроенные элементы управления создаются при помощи функции CreateWindow.

Для добавления к приложению панели инструментов необходимо выполнить следующую последовательность действий:

1. Определить ресурс растрового образа панели инструментов.
2. Объявить и инициализировать массив структур типа `TBBUTTON`, содержащий информацию о кнопках панели инструментов.
3. Вызвать функцию `CreateToolBarEx` для создания и инициализации панели инструментов.

Рассмотрим реализацию этих шагов на примере разработки приложения `ToolBar`, представляющего собой модифицированную версию программы `MenuDemo1`, описанной в главе 6. В результате модификации к интерфейсу приложения будет добавлена панель инструментов, дублирующая некоторые команды меню.

Начало разработки приложения `ToolBar`

Создайте новый проект с именем `ToolBar`. Скопируйте из папки проекта `MenuDemo1` (см. листинг 6.1) в папку проекта `ToolBar` файлы с расширениями `.cpp`, `.h` и `.rc`, скорректировав их имена заменой `MenuDemo1` на `ToolBar`. Добавьте эти файлы в состав проекта.

Укажите имя библиотеки `comctl32.lib` в настройках компоновщика в проекте. В среде Visual Studio 6.0 эти настройки находятся на вкладке `Link` диалогового окна `Project Settings`, которое открывается при помощи команды меню `Project > Settings`.

Определение ресурса растрового образа панели инструментов

В главном меню Visual Studio выполните команду `Insert > Resource`. В появившемся диалоговом окне `Insert Resource` укажите тип ресурса `Toolbar` и нажмите кнопку `New`. В результате будет открыто окно **редактора панели инструментов** с заготовкой растрового образа панели инструментов, содержащего первую и пока единственную кнопку (рис. 8.2).

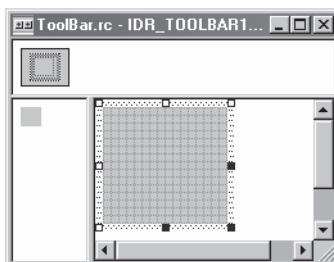


Рис. 8.2. Редактор панелей инструментов с заготовкой растрового образа панели

По умолчанию редактор присваивает растровому образу панели инструментов идентификатор `IDR_TOOLBAR1`. Если вы хотите изменить назначенный идентификатор, то на вкладке `ResourceView` в окне `Workspace` сделайте щелчок правой кнопкой мыши на элементе `IDR_TOOLBAR1` и в появившемся контекстном меню выберите пункт `Properties`. В открывшемся диалоговом окне `Toolbar Properties` введите в окне редактирования ID нужный идентификатор.

В нашем проекте можно оставить предложенный редактором идентификатор ресурса `IDR_TOOLBAR1`.

Окно редактора панелей инструментов, показанное на рис. 8.2, разделено на три части. В верхней части окна находится изображение создаваемой панели инструментов. В левой нижней части располагается изображение раstra для текущей кнопки в натуральную величину, а в правой нижней части — увеличенное изображение раstra для текущей кнопки. Растр для каждой кнопки имеет размеры 16×15 пикселов.

Для создаваемой кнопки нужно нарисовать картинку и определить идентификатор кнопки.

Пользуясь инструментами рисования на панели *Graphics*, создайте нужный рисунок на кнопке так же, как это делалось для пиктограмм. После этого сделайте двойной щелчок мышью на изображении кнопки в верхней части окна редактора. В открывшемся диалоговом окне *Toolbar Button Properties* введите нужный идентификатор кнопки. Если кнопка дублирует некоторый пункт меню, то идентификатор кнопки должен быть таким же, как и у дублируемого пункта меню.

Допустим, что вы нарисовали на кнопке изображение прямоугольника и указали ее идентификатор *ID_RECTANGLE*. После выполненных операций редактор панелей инструментов автоматически создает следующую кнопку, расположенную правее от созданной кнопки (рис. 8.3).

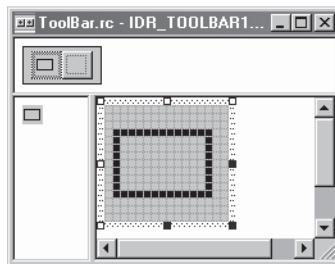


Рис. 8.3. Редактор панелей инструментов предлагает заготовку следующей кнопки

Щелкнув на новой кнопке мышью, можно перейти в режим редактирования этой кнопки, повторяя описанные выше действия.

Если текущая кнопка используется как кнопка-разделитель между группами кнопок, то изображение на ней создавать не нужно. В качестве идентификатора кнопок-разделителей в рассматриваемом примере будет использоваться именованная константа *ID_SEP* (вы можете использовать любое другое имя).

Если нужно изменить порядок размещения уже созданных кнопок, то это можно сделать при помощи мыши, ухватившись за перемещаемую кнопку и перетащив ее в нужную позицию. Для удаления созданной ранее кнопки просто схватите ее мышью и переместите за пределы изображения панели инструментов.

Вернемся к нашей программе. Используя описанную процедуру, определите раstroвый образ панели инструментов, как показано на рис. 8.4.



Рис. 8.4. Раstroвый образ панели инструментов

Раstroвый образ должен содержать 11 кнопок, причем две из них используются как кнопки-разделители. Изображения для кнопок принято нумеровать слева

направо, начиная с нуля. Подразумеваемые индексы изображений, описания изображений и идентификаторы кнопок приведены в табл. 8.10.

Таблица 8.10. Индексы изображений, их описание и идентификаторы кнопок

Индекс изображения	Описание изображения	Идентификатор кнопки
0	Прямоугольник	ID_RECTANGLE
1	Ромб	ID_RHOMB
2	Эллипс	ID_ELLIPSE
3	—	ID_SEP
4	Красный кружок	ID_RED
5	Зеленый кружок	ID_GREEN
6	Синий кружок	ID_BLUE
7	—	ID_SEP
8	Черный квадрат	ID_DARK
9	Серый квадрат	ID_MEDIUM
10	Белый квадрат	ID_LIGHT

Следует отметить, что идентификаторы кнопок, кроме кнопок-разделителей, совпадают с идентификаторами дублируемых пунктов меню.

Закончив визуальное проектирование панели инструментов, сохраните созданный растровый образ в файле описания ресурсов.

Обратите внимание на то, что теперь в составе проекта появился файл toolbar1.bmp, содержащий растровый образ панели инструментов. Имя этого файла использует вторую часть идентификатора панели инструментов.

Если открыть файл описания ресурсов ToolBar.rc в текстовом режиме, то в нем можно найти многострочное определение панели инструментов:

```
IDR_TOOLBAR1 TOOLBAR DISCARDABLE 16, 15
BEGIN
    BUTTON      ID_RECTANGLE
    BUTTON      ID_RHOMB
    ...
END
```

Помимо этого в описании ресурсов есть ссылка на указанный файл:

```
IDR_TOOLBAR1           BITMAP DISCARDABLE "toolbar1.bmp"
```

Заполнение массива структур типа TBBUTTON

Если для создания панели инструментов используется вызов функции CreateToolbarEx, то один из параметров функции должен получить адрес массива структур типа TBBUTTON. Структура TBBUTTON определена в файле commctrl.h следующим образом:

```
typedef struct _TBBUTTON {
    int iBitmap;        // индекс изображения кнопки или
                       // ширина кнопки-разделителя (в пикселях)
    int idCommand;     // идентификатор кнопки
    BYTE fsState;      // начальное состояние кнопки
    BYTE fsStyle;      // стиль кнопки
    DWORD dwData;      // дополнительные данные
    INT_PTR iString;   // индекс (относительно нуля) текстовой метки кнопки
} TBBUTTON;
```

Интерпретация полей структуры частично поясняется в комментариях.
Поле `fsState` может принимать комбинацию из следующих значений:

Флаг состояния	Описание
<code>TBSTATE_CHECKED</code>	Кнопка со стилем <code>TBSTYLE_CHECK</code> находится в нажатом состоянии
<code>TBSTATE_PRESSED</code>	Кнопка любого стиля находится в нажатом состоянии
<code>TBSTATE_ENABLED</code>	Кнопка доступна (может реагировать на действия мыши)
<code>TBSTATE_HIDDEN</code>	Скрытая кнопка (для пользователя недоступна)
<code>TBSTATE_INDETERMINATE</code>	Кнопка отображается серым цветом и недоступна для действий пользователя

Поле `fsStyle` может принимать одно из следующих значений:

Флаг стиля	Описание
<code>TBSTYLE_BUTTON</code>	Стандартная кнопка. Может быть нажата, но не может оставаться в нажатом состоянии
<code>TBSTYLE_SEP</code>	Разделитель для создания промежутка между группами кнопок. Может использоваться для резервирования места для дочерних элементов управления
<code>TBSTYLE_CHECK</code>	Кнопка ведет себя, как флагок (check box). Каждый щелчок мыши изменяет состояние кнопки (нажата / отжата)
<code>TBSTYLE_GROUP</code>	Кнопка является членом группы кнопок типа переключателей (radio button)
<code>TBSTYLE_CHECKGROUP</code>	Объединяет свойства стилей <code>TBSTYLE_CHECK</code> и <code>TBSTYLE_GROUP</code>

Поле `dwData` можно не использовать. Но при желании в нем можно сохранить указатель на дополнительную информацию, специфичную для конкретной кнопки. Разработчик может получить значение этого поля, послав сообщение `TB_GETBUTTON`.

Последнее поле `iString` используется только в тех случаях, когда кроме изображения кнопка имеет текстовую метку. В этом случае программист должен создать список текстовых строк панели инструментов, отправив сообщения `TB_ADDSTRING`.

Таким образом, прежде чем вызывать функцию `CreateToolbarEx`, следует объявить массив структур типа `TBBUTTON`:

`TBBUTTON tbb[NUM_BUTTONS];`
и заполнить поля элементов массива подходящими значениями.

Вызов функции `CreateToolbarEx`

Функция `CreateToolbarEx`, создающая и инициализирующая панель инструментов, имеет следующий прототип:

```
HWND CreateToolbarEx(
    HWND hwnd,           // дескриптор родительского окна
    DWORD ws,            // стили панели инструментов
    UINT wID,             // идентификатор панели инструментов
    int nBitmaps,         // количество изображений кнопок
    HINSTANCE hBMInst,   // дескриптор экземпляра приложения, содержащего
                         // ресурс растрового образа панели инструментов
    UINT wBMID,           // идентификатор ресурса растрового образа
    LPCTBBUTTON lpButtons, // адрес массива структур типа TBBUTTON
    int iNumButtons,      // количество кнопок
    int dxButton,          // ширина кнопок в пикселях
    int dyButton)          // высота кнопок в пикселях
```

```
int dxBitmap,           // ширина изображения кнопки в пикселях
int dyBitmap,           // высота изображения кнопки в пикселях
UINT uStructSize       // размер структуры TBBUTTON
);
```

Стили панели инструментов определяются параметром `ws`. Он обязательно должен содержать флаги `WS_CHILD` и `WS_VISIBLE`. Могут также указываться и другие стандартные стили, например `WS_BORDER`. Кроме того, есть два специальных стиля, которые предназначены для панелей инструментов. Один из них, `TBSTYLE_TOOLTIPS`, задает поддержку всплывающих подсказок для кнопок панели инструментов. Вторым стилем является `TBSTYLE_WRAPABLE`. Включение этого флага означает, что слишком длинные панели инструментов будут автоматически переходить на другую строку. Дополнительно могут указываться флаги основного стиля элементов управления общего пользования, приведенные в табл. 8.7. Если вы не указываете эти флаги, то по умолчанию применяется стиль `CCS_TOP`.

Параметру `wID` необходимо передать идентификатор панели инструментов. Вы можете использовать любой целочисленный идентификатор, но нужно следить за тем, чтобы это значение не повторялось среди идентификаторов, определенных в файле `resource.h`.

Параметры `dxButton` и `dyButton` задают ширину и высоту самих кнопок в пикселях, а параметры `dxBitmap` и `dyBitmap` — ширину и высоту изображения кнопки в пикселях. Если вы не хотите особых приключений, то передайте нулевое значение для всех четырех параметров. В этом случае используются значения по умолчанию: `dxButton=16, dyButton=16, dxBitmap=16, dyBitmap=15`. На самом деле Windows выводит изображения кнопок, включающие рамки, шириной не менее `dxButton+7` и высотой не менее `dyButton+6` пикселов.

Изменение размеров панели инструментов

Когда панель инструментов инициализирована, ее размеры устанавливаются соответственно текущим размерам родительского окна. Но если размер окна изменяется, то размер панели инструментов автоматически изменяться не будет. Особых неприятностей это не причиняет, однако при увеличении окна панель инструментов может оказаться слишком маленькой. Для решения этой проблемы необходимо посыпать сообщение `TB_AUTOSIZE` панели инструментов каждый раз, когда изменяются размеры родительского окна. Проще всего это сделать при обработке сообщения `WM_SIZE`:

```
SendMessage(hwndToolBar, TB_AUTOSIZE, 0, 0);
```

Реакция панели инструментов на это сообщение определяется флагами ее стиля (см. табл. 8.7). Например, панель инструментов со стилем `CCS_TOP` устанавливает свое местоположение и размеры. С другой стороны, панель инструментов со стилем `CCS_NORESIZE` игнорирует это сообщение и требует явной установки ее местоположения и размеров.

Поддержка элемента управления «подсказка»

Окно подсказки (Tooltip) — это маленькое окно, содержащее текст подсказки. Обычно подсказка всплывает, когда курсор мыши оказывается в «горячей» (*hot*) области, обслуживаемой данным элементом управления, задерживаясь в ней на некоторое время.

Элемент управления *Tooltip* хранит список «горячих» областей, которые для него ассоциируются с *инструментами (tool)*. Каждая «горячая» область опреде-

ляется как некоторая прямоугольная область в обслуживаемом окне или относится к окну целиком.

Существует две категории окон подсказки:

- ❑ *автономные* элементы управления Tooltip, создаваемые при помощи функции CreateWindowEx как окна предопределенного оконного класса TOOLTIPS_CLASS;
- ❑ элементы управления Tooltip, *встроенные* в панель инструментов.

Техника работы с ними различна. Первую категорию мы рассмотрим позже. Сейчас нас интересуют подсказки, встроенные в панель инструментов.

Если при создании панели инструментов указан стиль TBSTYLE_TOOLTIPS, то каждая кнопка, добавляемая в панель инструментов, регистрирует инструмент подсказки. Когда подсказка становится активной, она посыпает сообщение WM_NOTIFY с кодом уведомления WM_NEEDTEXT. Панель инструментов передает это сообщение своему родительскому окну, которое должно его обработать, если необходимо отобразить на экране текст подсказки.

При получении сообщения WM_NOTIFY параметр lParam содержит адрес структуры типа TOOLTIPTEXT, имеющей следующее определение:

```
typedef struct {
    NMHDR     hdr;           // информация об уведомительном сообщении
    LPTSTR    lpszText;       // указатель на строку или строковый ресурс
    WCHAR     szText[80];     // буфер для текста подсказки
    HINSTANCE hinst;         // дескриптор экземпляра приложения, содержащего
                            // строковый ресурс
    UINT      uflags;        // это поле не используется
} TOOLTIPTEXT;
```

Поле `hdr` содержит данные типа NMHDR, сопровождающие любое сообщение WM_NOTIFY. Эти данные включают информацию об элементе управления, пославшем сообщение, и информацию о самом сообщении:

```
typedef struct tagNMHDR {
    HWND hwndFrom;          // дескриптор окна элемента управления
    UINT idFrom;             // идентификатор элемента управления
    UINT code;                // код уведомления
} NMHDR;
```

Обрабатывая сообщение WM_NOTIFY, приложение обычно сохраняет значение параметра lParam в переменной lpTTT типа LPTOOLTIPTEXT, после чего проверяет значение поля lpTTT->hdr.code. Если оно равно TTN_NEEDTEXT, то это свидетельствует о поступлении запроса на отображение подсказки. Идентификатор кнопки, которая запросила отображение подсказки, хранится в поле lpTTT->hdr.idFrom. Зная этот идентификатор, приложение может выбрать необходимый текст для передачи окну подсказки.

Структура TOOLTIPTEXT позволяет использовать три способа передачи текста в подсказку:

1. Копирование текста в буфер с адресом lpTTT->szText.
2. Запись адреса буфера, содержащего текст, в поле lpTTT->lpszText.
3. Запись идентификатора строки, хранящейся в ресурсе таблицы строк, в поле lpTTT->lpszText и одновременно запись дескриптора экземпляра приложения, содержащего строковый ресурс, в поле lpTTT->hinst.

Реализация первых двух способов требует применения оператора `switch`, при помощи которого выбирается источник копирования текста в зависимости от значения поля `lpTTT->hdr.idFrom`.

Третий способ позволяет применить наиболее красивое решение. Оно основано на довольно простой идеи. Если все строки с текстами подсказок поместить в ресурс таблицы строк¹ и присвоить им идентификаторы, совпадающие с идентификаторами кнопок панели инструментов, то можно обойтись без оператора `switch` с гирляндой блоков `case`. Обработка сообщения `WM_NOTIFY` в этом случае будет выглядеть следующим образом:

```
case WM_NOTIFY:
    lpTTT = (LPTOOLTIPTEXT)lParam;
    if (lpTTT->hdr.code == TTN_NEEDTEXT) {
        lpTTT->hinst = GetModuleHandle(NULL);
        lpTTT->lpszText = MAKEINTRESOURCE(lpTTT->hdr.idFrom);
    }
    break;
```

В результате такой обработки панель инструментов отобразит подсказку для кнопки с идентификатором `lpTTT->hdr.idFrom`. При этом содержимым подсказки будет строка из таблицы строк точно с таким же идентификатором.

Применим новые знания, вернувшись к разработке приложения `ToolBar`.

Продолжение разработки приложения `ToolBar`

Добавьте к приложению ресурс таблицы строк. Введите в таблицу следующие строчки:

Идентификатор	Строка
ID_RECTANGLE	Прямоугольник
ID_RHOMB	Ромб
ID_ELLIPSE	Эллипс
ID_RED	Красная составляющая цвета
ID_GREEN	Зеленая составляющая цвета
ID_BLUE	Синяя составляющая цвета
ID_DARK	Темный цвет
ID_MEDIUM	Средний цвет
ID_LIGHT	Светлый цвет

В настоящий момент текст в файле `ToolBar.cpp` является точной копией текста файла `MenuDemo1.cpp`. Замените его текстом, приведенным в листинге 8.1.

Листинг 8.1. Проект `ToolBar`

```
///////////////////////////////
// ToolBar.cpp
#include <windows.h>
#include <commctrl.h>
#include <stdio.h>
#include "KWnd.h"
```

продолжение ↗

¹ Работа с ресурсом «Таблица строк» описана в главе 5.

Листинг 8.1 (продолжение)

```
#include "resource.h"

#define W 200 // ширина фигуры
#define H 140 // высота фигуры
enum ShapeSize { MAX, MIN };

typedef struct {
    int id_shape; // идентификатор фигуры
    BOOL fRed; // компонент красного цвета
    BOOL fGreen; // компонент зеленого цвета
    BOOL fBlue; // компонент синего цвета
    int id_bright; // идентификатор яркости цвета
} ShapeData;

#define ID_TOOLBAR 201
#define NUM_BUTTONS 11
#define SEPARATOR_WIDTH 10

HWND hWndToolBar;

HWND InitToolBar(HWND hWnd);
void UpdateToolBar(ShapeData& sd);

LRESULT CALLBACK WndProc(HWND hWnd, UINT, WPARAM, LPARAM);
//=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("ToolBar", hInstance, nCmdShow, WndProc,
        MAKEINTRESOURCE(IDR_MENU1), 100, 100, 400, 300);

    // Инициализация библиотеки "Common Control Library"
    INITCOMMONCONTROLSEX icc;
    icc.dwSize = sizeof(INITCOMMONCONTROLSEX);
    icc.dwICC = ICC_WIN95_CLASSES;
    InitCommonControlsEx(&icc);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}
//=====
HWND InitToolBar(HWND hWnd) {
    HWND hToolBar;
    int btnID[NUM_BUTTONS] = { ID_RECTANGLE, ID_RHOMB, ID_ELLIPSE, ID_SEP,
        ID_RED, ID_GREEN, ID_BLUE, ID_SEP, ID_DARK, ID_MEDIUM, ID_LIGHT };

    int btnStyle[NUM_BUTTONS] = { TBSTYLE_BUTTON, TBSTYLE_BUTTON,
        TBSTYLE_BUTTON, TBSTYLE_SEP, TBSTYLE_CHECK, TBSTYLE_CHECK,
        TBSTYLE_CHECK, TBSTYLE_SEP, TBSTYLE_CHECKGROUP, TBSTYLE_CHECKGROUP,
        TBSTYLE_CHECKGROUP };

    TBBUTTON tbb[NUM_BUTTONS];
```

```
memset(tbb, 0, sizeof(tbb));

for (int i = 0; i < NUM_BUTTONS; ++i) {
    if (btnID[i] == ID_SEP)
        tbb[i].iBitmap = SEPARATOR_WIDTH;
    else tbb[i].iBitmap = i;

    tbb[i].idCommand = btnID[i];
    tbb[i].fsState = TBSTATE_ENABLED;
    tbb[i].fsStyle = btnStyle[i];
}

hToolBar = CreateToolBarEx(hWnd,
    WS_CHILD | WS_VISIBLE | WS_BORDER | TBSTYLE_TOOLTIPS,
    ID_TOOLBAR, NUM_BUTTONS, GetModuleHandle(NULL), IDR_TOOLBAR1,
    tbb, NUM_BUTTONS, 0, 0, 0, 0, sizeof(TBBUTTON));
return hToolBar;
}
//=====
void UpdateToolBar(ShapeData& sd) {
    SendMessage(hwndToolBar, TB_CHECKBUTTON, ID_RED, sd.fRed);
    SendMessage(hwndToolBar, TB_CHECKBUTTON, ID_GREEN, sd.fGreen);
    SendMessage(hwndToolBar, TB_CHECKBUTTON, ID_BLUE, sd.fBlue);

    SendMessage(hwndToolBar, TB_CHECKBUTTON, ID_DARK,
        (sd.id_bright == ID_DARK));
    SendMessage(hwndToolBar, TB_CHECKBUTTON, ID_MEDIUM,
        (sd.id_bright == ID_MEDIUM));
    SendMessage(hwndToolBar, TB_CHECKBUTTON, ID_LIGHT,
        (sd.id_bright == ID_LIGHT));
}
//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    RECT rect;
    static HMENU hMenu; // дескриптор главного меню
    int x0, y0, x1, y1, x2, y2;
    POINT pt[4];
    static ShapeSize shapeSize = MIN;
    static BOOL bShow = TRUE;
    static HBRUSH hBrush, hOldBrush;
    char* itemResizedName[2] = { "Decrease!", "Increase!" };
    int intensity[3] = { 85, 170, 255 }; // интенсивность RGB-компонентов цвета
    int brightness;
    static ShapeData shapeData;

    RECT rcTB; // позиция и размеры окна hwndToolBar
    int tbHeight; // высота окна hwndToolBar
    LPTOOLTIPTEXT lptTTT;

    switch (uMsg)
    {
    case WM_CREATE:
        hMenu = GetMenu(hWnd);
        SetMenuItemDefaultItem(GetSubMenu(hMenu, 0), IDM_OPEN, FALSE);
    }
```

продолжение ↗

Листинг 8.1 (продолжение)

```

CheckMenuItem(GetSubMenu(hMenu, 1), IDM_SHOW_SHAPE,
    IDM_HIDE_SHAPE, IDM_SHOW_SHAPE, MF_BYCOMMAND);
CheckMenuItem(GetSubMenu(hMenu, 2), ID_RECTANGLE,
    ID_ELLIPSE, ID_RECTANGLE, MF_BYCOMMAND);
CheckMenuItem(GetSubMenu(hMenu, 2), ID_DARK,
    ID_LIGHT, ID_DARK, MF_BYCOMMAND);
shapeData.id_shape = ID_RECTANGLE;
shapeData.id_bright = ID_DARK;

// Создание панели инструментов
hwndToolBar = InitToolBar(hWnd);
SendMessage(hwndToolBar, TB_AUTOSIZE, 0, 0);
break;

// Настройка размеров панели инструментов
case WM_SIZE:
    SendMessage(hwndToolBar, TB_AUTOSIZE, 0, 0);
    break;

// Обработка запроса на вывод подсказки
case WM_NOTIFY:
    lpTTT = (LPTOOLTIPTEXT)lParam;
    if (lpTTT->hdr.code == TTN_NEEDTEXT) {
        lpTTT->hinst = GetModuleHandle(NULL);
        lpTTT->lpszText = MAKEINTRESOURCE(lpTTT->hdr.idFrom);
    }
    break;

case WM_COMMAND:
    switch (LOWORD(wParam))
    {
        /* Текст оператора из листинга 6.1 */
    }
    UpdateToolBar(shapeData); // обновление состояния кнопок
    InvalidateRect(hWnd, NULL, TRUE);
    break;

case WM_PAINT:
    hdc = BeginPaint(hWnd, &ps);
    brightness = intensity[shapeData.id_bright - ID_DARK];

    if (bShow) {
        hBrush = CreateSolidBrush(RGB(
            shapeData.fRed? brightness : 0,
            shapeData.fGreen? brightness : 0,
            shapeData.fBlue? brightness : 0));
        hOldBrush = (HBRUSH)SelectObject(hdc, hBrush);

        // Определение центра фигуры (x0, y0)
        GetClientRect(hWnd, &rect);
        // Модифицированный код
        // Учитываем уменьшение клиентской области из-за
        // размещения панели инструментов
        GetWindowRect(hwndToolBar, &rcTB);
        tbHeight = rcTB.bottom - rcTB.top;
        x0 = rect.right / 2;
    }
}

```

```
y0 = tbHeight + (rect.bottom - tbHeight) / 2;

// Координаты прямоугольника и эллипса
if (shapeSize == MIN) {
    x1 = x0 - W/2;    y1 = y0 - H/2;
    x2 = x0 + W/2;    y2 = y0 + H/2;
}
else {
    x1 = 0;           y1 = tbHeight;
    x2 = rect.right; y2 = rect.bottom;
}
// Далее - заимствованный код из MenuDemo1
// Координаты ромба
pt[0].x = (x1 + x2) / 2; pt[0].y = y1;
pt[1].x = x2;           pt[1].y = (y1 + y2) / 2;
pt[2].x = (x1 + x2) / 2; pt[2].y = y2;
pt[3].x = x1;           pt[3].y = (y1 + y2) / 2;

switch (shapeData.id_shape) {
case ID_RECTANGLE:
    Rectangle(hDC, x1, y1, x2, y2);    break;
case ID_RHOMB:
    Polygon(hDC, pt, 4);             break;
case ID_ELLIPSE:
    Ellipse(hDC, x1, y1, x2, y2);   break;
}
DeleteObject(SelectObject(hDC, hOldBrush));
}
EndPaint(hWnd, &ps);
break;

case WM_DESTROY:
PostQuitMessage(0);
break;

default:
return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
////////////////////////////////////////////////////////////////
```

В программе появились две новые (по сравнению с *MenuDemo1*) функции: *InitToolBar* и *UpdateToolBar*.

В теле функции *InitToolBar* создается и инициализируется панель инструментов *hToolBar*. Массив структур *tbb* типа *TBBUTTON* заполняется в цикле *for*. Обратите внимание на то, что для первой группы из трех кнопок назначается стиль *TBSTYLE_BUTTON*, для второй группы — стиль *TBSTYLE_CHECK*, а для третьей — *TBSTYLE_CHECKGROUP*. Полю *tbb[i].iBitmap* всех кнопок, за исключением кнопок-разделителей, присваивается индекс изображения в растровом образе панели инструментов. Для кнопок-разделителей это поле получает значение константы *SEPARATOR_WIDTH*.

При вызове функции *CreateToolbarEx* помимо основных стилей окна передается также стиль *TBSTYLE_TOOLTIPS*, что позволяет обеспечить поддержку встроенной подсказки для кнопок панели инструментов. Идентификатор панели инструментов *ID_TOOLBAR*, передаваемый функции, определен с помощью директивы *#define*. Функция *InitToolBar* вызывается из оконной процедуры *WndProc* в блоке обработки сообщения *WM_CREATE*.

Кнопки панели инструментов, имеющие стиль `TBSTYLE_CHECK` или `TBSTYLE_CHECKGROUP`, требуют к себе повышенного внимания. Дело в том, что для них нужно обеспечить «обратную связь» от меню. Если изменение опции выполнено пользователем через пункт меню, то состояние указанных кнопок необходимо обновить, иначе оно не будет синхронизировано с изменившимся состоянием приложения. Эту подзадачу решает функция `UpdateToolBar`, посылая обновляемым кнопкам сообщение `TB_CHECKBUTTON`. Функция `UpdateToolBar` вызывается в конце блока обработки сообщения `WM_COMMAND`.

Обрабатывая сообщение `WM_SIZE`, приложение посылает панели инструментов сообщение `TB_AUTOSIZE`, чтобы панель изменила свои размеры в соответствии с изменившимися размерами родительского окна. Это же сообщение посыпается сразу после вызова функции `InitToolBar`. В принципе, в данной программе это не обязательно. Но если вы создаете панель инструментов со стилем `TBSTYLE_WRAPABLE`, то приведенная инструкция обеспечивает нормальную настройку начального вида панели инструментов.

Обрабатывая сообщение `WM_NOTIFY`, приложение выбирает из таблицы строк текст подсказки для соответствующей кнопки панели инструментов. В результате панель инструментов отображает требуемое окно подсказки. Приведенный код мы разбирали выше в разделе «Поддержка элемента управления «подсказка»».

Код обработки сообщения `WM_PAINT` слегка модифицирован, чтобы учесть уменьшение видимой части клиентской области из-за размещения панели инструментов.

На рис. 8.5 показан вид работающего приложения `ToolBar`.

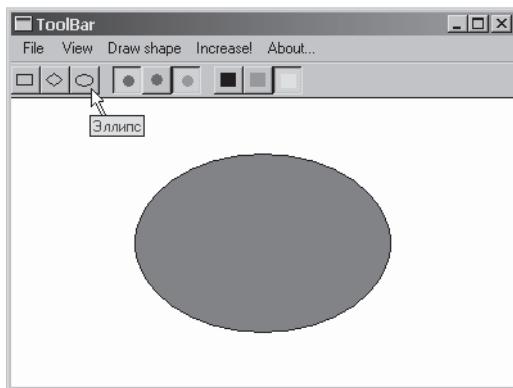


Рис. 8.5. Окно приложения `ToolBar` с активизированной подсказкой «Эллипс»

Панель инструментов с дополнительными текстовыми метками

Ранее говорилось, что кнопки панели инструментов помимо растровых изображений могут содержать и текстовые метки. Элемент управления `Toolbar` имеет встроенный список текстовых строк, который можно заполнять, отправляя элементу сообщения `TB_ADDSTRING`. Кроме того, при инициализации массива `tbb` необходимо присвоить полю `tbb[i].iString` соответствующий индекс (удобнее всего, если этот индекс будет совпадать с индексом изображения).

Чтобы посмотреть, как все это работает, сделаем небольшую доработку предыдущей программы. Создайте новый проект с именем ToolBarWithText. Скопируйте из папки проекта ToolBar (см. листинг 8.1) в папку проекта ToolBarWithText файлы с расширениями .cpp, .h и .rc, скорректирував их имена заменой подстроки ToolBar на ToolBarWithText. Скопируйте также файл toolbar1.bmp. Добавьте эти файлы в состав проекта. Также добавьте к настройкам проекта на вкладке Link библиотеку comctl32.lib.

Отредактируйте текст файла ToolBarWithText.cpp так, чтобы он соответствовал листингу 8.2.

Листинг 8.2. Проект ToolBarWithText

```
////////////////////////////////////////////////////////////////////////
// ToolBarWithText.cpp
/* Директивы препроцессора, объявления типов и глобальных переменных из листинга 8.1
*/
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====================================================
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("ToolBarWithText", hInstance, nCmdShow, WndProc,
        MAKEINTRESOURCE(IDR_MENU1), 100, 100, 430, 300);

    /* Здесь такой же текст, как и в листинге 8.1 */

    return msg.wParam;
}
//=====================================================
HWND InitToolBar(HWND hWnd) {
    /* Определения переменной hToolBar и массивов btnID и btnStyle
     - из листинга 8.1 */
    TBBUTTON tbb[NUM_BUTTONS];
    memset(tbb, 0, sizeof(tbb));

    for (int i = 0; i < NUM_BUTTONS; ++i) {
        if (btnID[i] == ID_SEP)
            tbb[i].iBitmap = SEPARATOR_WIDTH;
        else
            tbb[i].iBitmap = i;

        tbb[i].idCommand = btnID[i];
        tbb[i].fsState = TBSTATE_ENABLED;
        tbb[i].fsStyle = btnStyle[i];
        tbb[i].iString = i; // индекс строки в списке строк
    }

    hToolBar = CreateToolbarEx(hWnd,
        WS_CHILD | WS_VISIBLE | WS_BORDER | TBSTYLE_TOOLTIPS,
        ID_TOOLBAR, NUM_BUTTONS, GetModuleHandle(NULL), IDR_TOOLBAR1,
        tbb, NUM_BUTTONS, 0, 0, 0, 0, sizeof(TBBUTTON));
}

// Заполнение списка строк в элементе управления Toolbar
```

продолжение ↗

Листинг 8.2 (продолжение)

```

const char* str[NUM_BUTTONS] = { "Rect", "Rhomb", "Ellips", "",  

    "Red", "Green", "Blue", "", "Dark", "Medium", "Light" };  

for (i = 0; i < NUM_BUTTONS; ++i)  

    SendMessage(hToolBar, TB_ADDSTRING, 0, (LPARAM)str[i]);  
  

    return hToolBar;  

}  
=====  
void UpdateToolBar(ShapeData& sd) {  

    /* Здесь такой же текст, как и в листинге 8.1 */  

}  
=====  
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)  

{  

    /* Здесь такой же текст, как и в листинге 8.1 */  

}
=====

```

Обратите внимание на изменения и дополнения в тексте функции `InitToolBar`. В цикле инициализации массива `tbb` добавлено присваивание `tbb[i].iString = i`. После создания панели инструментов ее встроенный список строк наполняется текстовыми метками из массива `str`. Чтобы индексы этих строк совпадали с индексами изображений кнопок в ресурсе `IDR_TOOLBAR1`, массив `str` содержит пустые строки в тех позициях, которые соответствуют кнопкам-разделителям.

Окно работающего приложения показано на рис. 8.6.

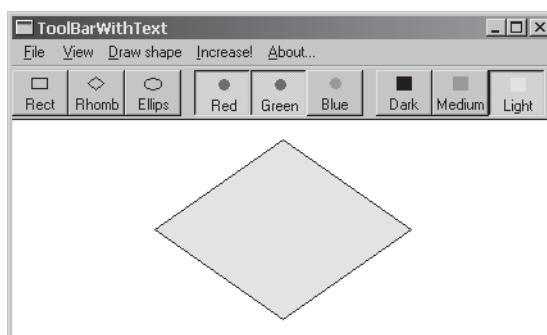


Рис. 8.6. Окно приложения ToolBarWithText

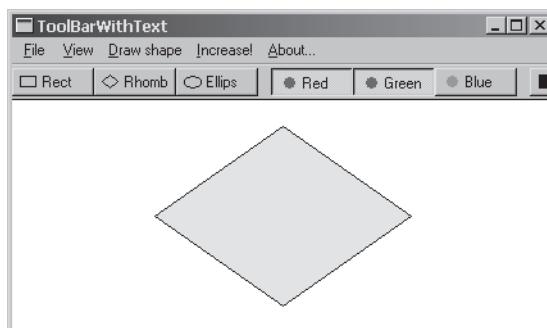


Рис. 8.7. Панель инструментов со стилем `TBSTYLE_LIST`

Теперь панель инструментов увеличила размеры кнопок с учетом размещения самого длинного текста. По умолчанию текстовые метки выводятся ниже рисунков. Но можно расположить их справа, если при вызове функции `CreateToolBarEx` указать дополнительный стиль `TBSTYLE_LIST`. Вид панели инструментов для этого стиля показан на рис. 8.7.

Кнопки вытянулись в ширину, поэтому при заданных размерах окна пользователь видит только первые шесть кнопок. Чтобы увидеть панель инструментов полностью, он вынужден раздвигать окно приложения с помощью мыши.

Но если при вызове функции `CreateToolBarEx` указать не только стиль `TBSTYLE_LIST`, но еще и стиль `TBSTYLE_WRAPABLE`, то панель инструментов будет автоматически переходить на новую строку, когда ширина окна оказывается недостаточной для нормального отображения панели. Этот вариант реализации панели инструментов показан на рис. 8.8.

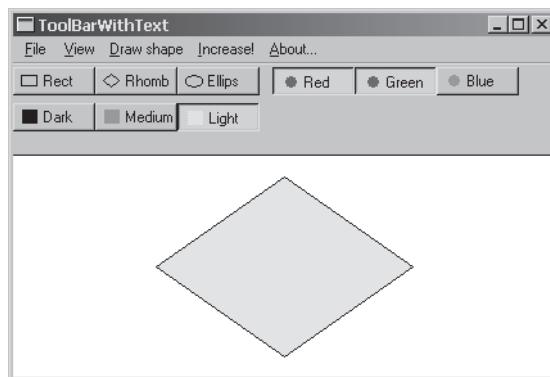


Рис. 8.8. Панель инструментов со стилями `TBSTYLE_LIST` | `TBSTYLE_WRAPABLE`

Размещение на панели инструментов других элементов управления

Панель инструментов поддерживает только кнопки, поэтому для расположения на ее поверхности какого-либо другого элемента управления следует создать дочернее окно. Наиболее часто на панель инструментов добавляются комбинированные списки (*combo box*).

Добавление дочернего окна элемента управления на панель инструментов связано с решением трех проблем: а) резервирование места для дочернего окна; б) обработка сообщений от дочернего окна; в) поддержка подсказки для дочернего окна.

Резервирование места под встроенный элемент управления несложно обеспечить, поместив кнопку-разделитель требуемой ширины. Напомним, что фактическая ширина кнопки-разделителя задается присваиванием нужного значения полю `tbb[i].iBitmap`. Обычно это значение подбирается экспериментально, исходя из желательной ширины встраиваемого элемента управления.

Уведомительные сообщения от встроенного элемента управления поступают в виде сообщений `WM_COMMAND` в родительское окно панели инструментов. Так как оконная процедура панели инструментов (спрятанная в недрах Windows) эти

сообщения не обрабатывает, она передает их родительскому окну панели инструментов, то есть главному окну приложения. Поэтому в блоке обработки сообщения **WM_COMMAND** оконной процедуры **WndProc** необходимо предусмотреть обработку этих уведомительных сообщений.

Мы уже знаем, что панель инструментов поддерживает окна подсказок для всех своих кнопок. Но на встроенные элементы управления эта поддержка не распространяется. В следующем разделе будет показано, как решить эту проблему с помощью автономных элементов управления **Tooltip**.

Рассмотрим технику добавления комбинированного списка на панель инструментов на примере разработки приложения **ComboInToolbar**, которое является модификацией приложения **ToolBar**. Цель модификации — добавить возможность выбора толщины пера, которым обводится контур рисуемой фигуры (до сих пор использовалось перо по умолчанию толщиной в 1 пиксел).

Создайте новый проект с именем **ComboInToolbar**. Затем скопируйте из папки проекта **ToolBar** (см. листинг 8.1) в папку проекта **ComboInToolbar** файлы с расширениями .cpp, .h и .rc, скорректировав их имена заменой подстроки **ToolBar** на **ComboInToolbar**. Скопируйте также файл **toolbar1.bmp**. Добавьте эти файлы в состав проекта. Добавьте к настройкам проекта на вкладке **Link** библиотеку **comctl32.lib**.

Откройте вкладку **ResourceView** в окне **Workspace** и вызовите редактор панелей инструментов двойным щелчком мыши на элементе **IDR_TOOLBAR1**. Добавьте в начале панели инструментов две кнопки без изображений с идентификаторами **ID_SEP**.

Откройте текст файла **ComboInToolbar.cpp** и отредактируйте его так, чтобы он соответствовал листингу 8.3.

Листинг 8.3. Проект **ComboInToolbar**

```
////////////////////////////////////////////////////////////////////////
// ComboInToolBar.cpp

/* Директивы #include и #define, объявление типов ShapeSize и ShapeData
 - из листинга 8.1 */

#define ID_TOOLBAR      201
#define IDC_TB_COMBOBOX 202
#define NUM_BUTTONS     13
#define SEPARATOR_WIDTH 10
#define COMBO_SPACE_WIDTH 50
#define COMBO_SPACE_HEIGHT 100

HWND hwndToolBar;
HWNDF hwndCombo;

HWND InitToolBar(HWND hWnd);
void UpdateToolBar(ShapeData& sd);

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
```

```
KWnd mainWnd("ComboInToolBar", hInstance, nCmdShow, WndProc,
    MAKEINTRESOURCE(IDR_MENU1), 100, 100, 400, 300);

/* Здесь такой же текст, как и в листинге 8.1 */

return msg.wParam;
}

//=====
HWND InitToolBar(HWND hWnd) {

    HWND hToolBar;

    int btnID[NUM_BUTTONS] = { ID_SEP, ID_SEP, ID_RECTANGLE, ID_RHOMB,
        ID_ELLIPSE, ID_SEP, ID_RED, ID_GREEN, ID_BLUE, ID_SEP,
        ID_DARK, ID_MEDIUM, ID_LIGHT };

    int btnStyle[NUM_BUTTONS] = { TBSTYLE_SEP, TBSTYLE_SEP,
        TBSTYLE_BUTTON, TBSTYLE_BUTTON, TBSTYLE_BUTTON, TBSTYLE_SEP,
        TBSTYLE_CHECK, TBSTYLE_CHECK, TBSTYLE_CHECK, TBSTYLE_SEP,
        TBSTYLE_CHECKGROUP, TBSTYLE_CHECKGROUP, TBSTYLE_CHECKGROUP };

    TBBUTTON tbb[NUM_BUTTONS];
    memset(tbb, 0, sizeof(tbb));

    for (int i = 0; i < NUM_BUTTONS; ++i) {
        if (!i) tbb[i].iBitmap = COMBO_SPACE_WIDTH;
        else if (btnID[i] == ID_SEP)
            tbb[i].iBitmap = SEPARATOR_WIDTH;
        else tbb[i].iBitmap = i;

        tbb[i].idCommand = btnID[i];
        tbb[i].fsState = TBSTATE_ENABLED;
        tbb[i].fsStyle = btnStyle[i];
    }

    hToolBar = CreateToolbarEx(hWnd,
        WS_CHILD | WS_VISIBLE | WS_BORDER | TBSTYLE_TOOLTIPS,
        ID_TOOLBAR, NUM_BUTTONS, GetModuleHandle(NULL), IDR_TOOLBAR1,
        tbb, NUM_BUTTONS, 0, 0, 0, 0, sizeof(TBBUTTON));

    // Определение позиции и размеров для элемента управления Combo box
    int x, y, cx, cy;
    RECT rcItem;
    SendMessage(hToolBar, TB_GETITEMRECT, 0, (LPARAM)&rcItem);
    x = rcItem.left + 2;      y = rcItem.top;
    cx = COMBO_SPACE_WIDTH;  cy = COMBO_SPACE_HEIGHT;

    // Создание элемента управления Combo box
    hwndCombo = CreateWindow("combobox", NULL, WS_CHILD | WS_VISIBLE |
        CBS_DROPDOWN, x, y, cx, cy, hToolBar, (HMENU)IDC_TB_COMBOBOX,
        GetModuleHandle(NULL), 0);

    // Инициализация списка для hwndCombo
    SendMessage(hwndCombo, CB_ADDSTRING, 0, (LPARAM)" 1 ");
    SendMessage(hwndCombo, CB_ADDSTRING, 0, (LPARAM)" 2 ");

}
```

продолжение ↗

Листинг 8.3 (продолжение)

```
SendMessage(hwndCombo, CB_ADDSTRING, 0, (LPARAM)" 5 ");
SendMessage(hwndCombo, CB_ADDSTRING, 0, (LPARAM)" 10 ");
SendMessage(hwndCombo, CB_SETCURSEL, 0, 0);

return hToolBar;
}
//=====
void UpdateToolBar(ShapeData& sd)
/* Здесь такой же текст, как и в листинге 8.1 */
{
}
//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
/* Объявления локальных переменных из листинга 8.1 */

static HPEN hPen, hOldPen;
static int indPen;
int penWidth[] = { 1, 2, 5, 10 };

switch (uMsg)
{
case WM_CREATE:

/* Здесь такой же текст, как и в листинге 8.1 */

break;

case WM_SIZE:
SendMessage(hToolBar, TB_AUTOSIZE, 0, 0);
break;

case WM_NOTIFY:
/* Здесь такой же текст, как и в листинге 8.1 */
break;

case WM_COMMAND:
switch (LOWORD(wParam))
{

/* Обработка для кодов сообщений IDM_OPEN, IDM_CLOSE, ... , IDM_RESIZE -
как в листинге 8.1 */

case IDM_ABOUT:
MessageBox(hWnd,
"ComboInToolBar\nVersion 1.0\nCopyright: "
"Finesoft Corporation, 2005.",
"About ComboInToolBar", MB_OK);
break;

// Обработка уведомительного сообщения от комбинированного списка
case IDC_TB_COMBOBOX:
switch (HIWORD(wParam)) {
case CBN_SELCHANGE:
// Выбираем текущий индекс пера
indPen = SendMessage(hwndCombo, CB_GETCURSEL, 0, 0);
}
}
```

```
        break;
    default:
        break;
    }

    UpdateToolBar(shapeData);
    InvalidateRect(hWnd, NULL, TRUE);
    break;

case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);
    brightness = intensity[shapeData.id_bright - ID_DARK];

    if (bShow) {

        hPen = CreatePen(PS_SOLID, penWidth[indPen], RGB(0,0,0));
        hOldPen = (HPEN)SelectObject(hDC, hPen);

        /* Здесь такой же текст, как и в листинге 8.1 */

        DeleteObject(SelectObject(hDC, hOldBrush));
        DeleteObject(SelectObject(hDC, hOldPen));
    }
    EndPaint(hWnd, &ps);
    break;

case WM_DESTROY:
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
////////////////////////////////////////////////////////////////
```

В этой программе количество кнопок на панели инструментов увеличено до тринадцати, так как добавлены две кнопки-разделителя в начале панели. Первая из этих кнопок-разделителей, имеющая ширину COMBO_SPACE_WIDTH, используется для резервирования места под комбинированный список. Вторая кнопка-разделитель создает промежуток между комбинированным списком и первой стандартной кнопкой панели инструментов.

Обратите внимание на то, что в массивах btnID и btnStyle учтено появление новых кнопок. Первые два элемента в btnID инициализированы значением ID_SEP, а первые два элемента в btnStyle инициализированы значением TBSTYLE_SEP.

Панель инструментов создается, как и раньше, вызовом функции CreateToolbarEx. Затем определяются позиция и размеры для встраиваемого комбинированного списка. Поскольку он размещается на месте первой кнопки-разделителя, мы узнаем координаты изображения этой кнопки, имеющей нулевой индекс, отправив сообщение TB_GETITEMRECT. В результате структура rcItem получает искомые координаты кнопки-разделителя. Эти координаты используются затем для вычисления позиции (*x*, *y*) встраиваемого элемента управления.

Дочернее окно комбинированного списка `hwndCombo` создается вызовом функции `CreateWindow`. Инициализация списка осуществляется отправкой серии сообщений `CB_ADDSTRING`.

В тексте оконной процедуры `WndProc` произошли определенные изменения. Прежде всего были добавлены определения переменных:

```
static HPEN hPen, hOldPen;
static int indPen;
int penWidth[] = { 1, 2, 5, 10 };
```

Также добавлена обработка уведомительного сообщения `CBN_SELCHANGE` от комбинированного списка:

```
case IDC_TB_COMBOBOX:
    switch (HIWORD(wParam)) {
        case CBN_SELCHANGE:
            // Выбираем текущий индекс пера
            indPen = SendMessage(hwndCombo, CB_GETCURSEL, 0, 0);
    }
    break;
```

В блоке обработки сообщения `WM_PAINT` выбранный индекс пера используется при создании пера `hPen`:

```
hPen = CreatePen(PS_SOLID, penWidth[indPen], RGB(0,0,0));
hOldPen = (HPEN)SelectObject(hdc, hPen);
```

На рис. 8.9 показано окно работающего приложения `ComboInToolbar` после выбора в окне комбинированного списка значения 10.

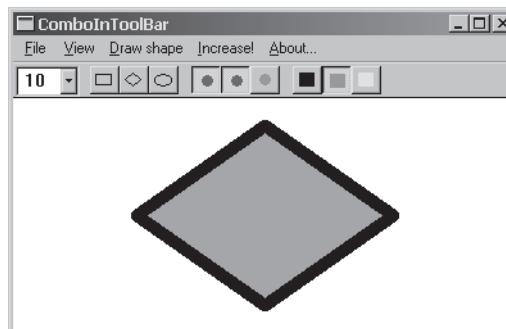


Рис. 8.9. Приложение `ComboInToolbar`. Выбрана толщина пера 10 логических единиц

К сожалению, в нашей новой программе комбинированный список, встроенный на панель инструментов, отличается от кнопок панели тем, что не имеет окна подсказки. В следующем разделе мы сможем устранить этот недостаток, применив автономный элемент управления `Tooltip`.

Окно подсказки

В отличие от встроенного на панель инструментов элемента управления `Tooltip`, *автономный элемент управления Tooltip* создается в явном виде как окно предопределенного оконного класса `TOOLTIPS_CLASS` при помощи функции `CreateWindowEx`. В дальнейшем изложении, употребляя термин «окно подсказки», мы будем говорить именно об автономном элементе управления `Tooltip`.

Следующий фрагмент кода показывает, как обычно создается окно подсказки:

```
hwndTip = CreateWindowEx(WS_EX_TOPMOST, TOOLTIPS_CLASS, NULL,
WS_POPUP | TTS_NOPREFIX | TTS_ALWAYSTIP,
CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
hwndOwner, NULL, hInst, NULL);
```

Элемент управления **Tooltip** всегда имеет стили **WS_EX_TOPMOST** и **WS_POPUP**, независимо от того, указаны ли они при вызове функции **CreateWindowEx**. Дополнительно можно использовать стиль **TTS_NOPREFIX**, означающий, что символ амперсанда (&) не является префиксом для обозначения «горячей» клавиши в меню и выводится в строке подсказки наравне с другими символами. Также можно использовать стиль **TTS_ALWAYSTIP**, означающий, что окно подсказки появляется при наведении курсора мыши на инструмент даже тогда, когда окно-владелец подсказки неактивно.

Напомним, что элемент управления **Tooltip** может хранить список «горячих» областей или инструментов (*tool*). Каждый инструмент — это некоторая прямоугольная область в обслуживаемом окне (в пределе — все окно).

Элемент управления **Tooltip** начинает поддерживать некий инструмент только после его регистрации. *Регистрация инструмента* осуществляется отправкой сообщения **TTM_ADDTOOL**, например, при помощи следующей инструкции:

```
SendMessage(hwndTip, TTM_ADDTOOL, 0, (LPARAM)(LPTOOLINFO) &ti);
```

Последнему параметру функции **SendMessage** передается адрес структурной переменной **ti** типа **TOOLINFO**. Структура **TOOLINFO** имеет следующее определение:

```
typedef struct tagTOOLINFO {
    UINT      cbSize;    // размер структуры в байтах
    UINT      uFlags;    // флаги
    HWND      hwnd;      // дескриптор окна, содержащего инструмент
    WPARAM   uId;       // идентификатор инструмента
    RECT     rect;       // ограничивающий прямоугольник для инструмента
    HINSTANCE hinst;    // дескриптор экземпляра приложения, содержащего
                        // строковый ресурс с текстом подсказки
    LPTSTR   lpszText;  // указатель на буфер с С-строкой или идентификатор
                        // строкового ресурса
#ifdef _WIN32_IE >= 0x0300
    PARAM  lParam;     // необязательное 32-битное значение,
                      // связанное с инструментом
#endif
} TOOLINFO, FAR* LPTOOLINFO;
```

Поле **uFlags** может содержать один или несколько флагов из тех, что приведены в табл. 8.11.

Таблица 8.11. Флаги для поля **uFlags**

Флаг	Описание
TTF_CENTERTIP	Окно подсказки выводится в центре по горизонтали под инструментом, а не как по умолчанию — ниже и справа относительно курсора мыши
TTF_IDISHWND	Означает, что в поле uId задается дескриптор дочернего окна. Если этот флаг не задан, то поле uId является идентификатором инструмента
TTF_SUBCLASS	Означает, что окно подсказки будет иметь свою оконную процедуру для перехвата и обработки сообщений, таких как WM_MOUSEMOVE

Если флаг **TTF_IDISHWND** не указан, то поле **uId** содержит произвольный идентификатор для инструмента. В принципе, этот идентификатор может и не использоваться в дальнейшей работе.

Комментарий к полю `lpszText` показывает, что текст подсказки может храниться либо в виде С-строки, либо в виде строкового ресурса в таблице строк приложения.

Таким образом, создав элемент управления `Tooltip` и зарегистрировав в нем «горячую» область какого-либо окна, можно обеспечить вывод требуемой подсказки. Перечисленные действия имеют смысл инкапсулировать в теле какой-нибудь функции. Такой подход будет продемонстрирован в рассмотренном ниже примере.

Замена класса `KWnd` на класс `KWndEx`

Вернемся к нашей последней программе, в которой комбинированный список, встроенный в панель инструментов, не имел всплывающей подсказки. Попробуем устранить этот недостаток.

Одновременно заменим хорошо послуживший нам класс `KWnd` на его модификацию — класс `KWndEx`. Цель замены — обеспечить регистрацию оконных классов для элементов управления общего пользования в конструкторе класса, чтобы освободить программиста от этой рутинной операции. Кроме того, в файлах `KWndEx.h`, `KWndEx.cpp` будут размещены также интерфейс и реализация *функций общего применения*.

В состав функций общего применения войдут следующие функции:

- ❑ перегруженная функция `ShiftWindow`, предназначенная для сдвига и модификации размеров окон (эта функция использовалась ранее в листинге 7.5);
- ❑ функция `AddTooltip`, добавляющая окно подсказки к указанному окну;
- ❑ функция `TRACE`, которую можно использовать для отладочного вывода в окно `Output` (аналогично макросу `TRACE` в библиотеке MFC).

Все во имя программиста, все для блага программиста!

Создайте новый проект с именем `ToolTip`. Скопируйте из папки проекта `ComboInToolbar` (см. листинг 8.3) в папку проекта `ToolTip` файлы с расширениями `.cpp`, `.h` и `.rc`, скорректировав их имена заменой подстроки `ComboInToolbar` на `ToolTip`. Скопируйте также файл `toolbar1.bmp`. Измените имена файлов `Kwnd.h`, `Kwnd.cpp` на `KwndEx.h` и `KwndEx.cpp` соответственно. Добавьте все перечисленные файлы в состав проекта. Также к настройкам проекта на вкладке `Link` надо добавить библиотеку `comctl32.lib`.

Откройте вкладку `ResourceView` в окне `Workspace`. В списке ресурсов откройте папку `String table` и вызовите редактор таблицы строк двойным щелчком мыши на элементе `String table`. Добавьте в таблицу строки следующие две строки:

Идентификатор	Строка
<code>IDS_TB_COMBOBOX</code>	Доступ к списку для выбора толщины пера
<code>IDS_EDIT_IN_COMBO</code>	Толщина пера для контура фигуры

Обратите внимание на то, что потребовалось указать две строки, чтобы элемент управления `Combo box` получил все необходимые подсказки. Первый текст подсказки будет отображаться при наведении курсора мыши на стрелку, используемую для открывания списка, второй текст подсказки — при наведении курсора мыши на окно редактирования.

Отредактируйте тексты файлов `KwndEx.h`, `KwndEx.cpp`, `ToolTip.cpp` так, чтобы они соответствовали листингу 8.4.

Листинг 8.4. Проект ToolTip

```
////////////////////////////////////////////////////////////////
// KWndEx.h
#include <windows.h>

class KWndEx {
public:
    KWndEx(LPCTSTR windowName, HINSTANCE hInst, int cmdShow,
           LRESULT (WINAPI *pWndProc)(HWND,UINT,WPARAM,LPARAM),
           LPCTSTR menuName = NULL,
           int x = CW_USEDEFAULT, int y = 0,
           int width = CW_USEDEFAULT, int height = 0,
           UINT classStyle = CS_HREDRAW | CS_VREDRAW,
           DWORD windowStyle = WS_OVERLAPPEDWINDOW,
           HWND hParent = NULL);

    HWND GetHWnd() { return hWnd; }

protected:
    HWND hWnd;
    WNDCLASSEX wc;
};

//================================================================
// Функции общего применения
//-----
// Сдвиги окон
// Сдвиг окна верхнего уровня с модификацией размеров
void ShiftWindow(HWND hwnd, int dX = 0, int dY = 0,
                  int dw = 0, int dh = 0);
// Сдвиг дочернего окна с модификацией размеров
void ShiftWindow(HWND hChild, HWND hParent, int dX = 0,
                  int dY = 0, int dw = 0, int dh = 0);
// Сдвиг окна элемента управления с модификацией размеров
void ShiftWindow(int ctrlID, HWND hParent, int dX = 0,
                  int dY = 0, int dw = 0, int dh = 0);
//-----
// Добавление окна подсказки
void AddTooltip (HWND hWndOwner, LPTSTR lpszMsg);

//-----
// Функция отладочной печати (вывод в окно "Output")
// - работает аналогично макросу TRACE в библиотеке MFC
void TRACE(LPCTSTR lpszFormat, ...);

////////////////////////////////////////////////////////////////
// KWndEx.cpp
#include "KWndEx.h"
#include <commctrl.h>

KWndEx::KWndEx(LPCTSTR windowName, HINSTANCE hInst, int cmdShow,
               LRESULT (WINAPI *pWndProc)(HWND,UINT,WPARAM,LPARAM),
               LPCTSTR menuName, int x, int y, int width, int height,
               UINT classStyle, DWORD windowStyle, HWND hParent)
{
    /* Такой же текст, как и в конструкторе KWnd (листинг 1.2) */

    // Инициализация библиотеки "Common Control Library"
```

Листинг 8.4 (продолжение)

```

INITCOMMONCONTROLSEX icc;
icc.dwSize = sizeof(INITCOMMONCONTROLSEX);
icc.dwICC = ICC_WIN95_CLASSES;
InitCommonControlsEx(&icc);

}

//=====
// Функции общего применения
//-----
// Сдвиг окна верхнего уровня с модификацией размеров
void ShiftWindow(HWND hwnd, int dx, int dY, int dw, int dh) {
    /* Текст функции из листинга 7.5 (файл KWndPlut.cpp) */
}

// Сдвиг дочернего окна с модификацией размеров
void ShiftWindow(HWND hChild, HWND hParent, int dx, int dY, int dw, int dh) {
    /* Текст функции из листинга 7.5 */
}

// Сдвиг окна элемента управления с модификацией размеров
void ShiftWindow(int ctrlID, HWND hParent, int dx, int dY, int dw, int dh) {
    /* Текст функции из листинга 7.5 */
}

//-----
// Добавление окна подсказки
void AddTooltip (HWND hwndOwner, LPTSTR lpMsg) {
    HWND hwndTip;           // дескриптор элемента управления Tooltip
    static TOOLINFO ti;     // информация об инструменте (о «горячей» области),
                           // обслуживаемом элементом hwndTip

    HINSTANCE hInst = (HINSTANCE)GetWindowLong(hwndOwner, GWL_HINSTANCE);

    hwndTip = CreateWindowEx(WS_EX_TOPMOST, TOOLTIPS_CLASS, NULL,
                           WS_POPUP | TTS_NOPREFIX | TTS_ALWAYSTIP,
                           CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
                           hwndOwner, NULL, hInst, NULL);

    // Инициализация структуры ti
    ti.cbSize = sizeof(TOOLINFO);
    ti.uFlags = TTF_SUBCLASS;
    ti.hwnd = hwndOwner;
    ti.hinst = hInst;
    ti.uId = 0;
    ti.lpszText = lpMsg;

    // Местоположение инструмента в окне hwndOwner
    // (инструмент покрывает всю клиентскую область)
    GetClientRect (hwndOwner, &ti.rect);

    // Регистрация инструмента с информацией ti
    SendMessage(hwndTip, TTM_ADDTOOL, 0, (LPARAM)(LPTOOLINFO) &ti);
}

//-----

```

```
// Функция отладочной печати (аналог макроса TRACE в библиотеке MFC)

#define N 512

void TRACE(LPCTSTR szFormat, ...)
{
    va_list args;
    va_start(args, szFormat);
    int nBuf;
    char szBuffer[N];

    nBuf = _vsnprintf(szBuffer, N, szFormat, args);

    if (nBuf < 0) {
        MessageBox(NULL, "Слишком длинная строка для TRACE!",
            "Ошибка", MB_OK | MB_ICONSTOP);
        szBuffer[N-2] = '\n';
        szBuffer[N-1] = 0;
    }

    OutputDebugString(szBuffer);

    va_end(args);
}

////////////////////////////////////////////////////////////////////////
// ToolTip.cpp
#include <windows.h>
#include <commctrl.h>
#include <stdio.h>
#include "KWndEx.h"
#include "resource.h"

/* Определения макросов, типов и глобальных переменных из листинга 8.3 */

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWndEx mainWnd("ToolTip", hInstance, nCmdShow, WndProc,
        MAKEINTRESOURCE(IDR_MENU1), 100, 100, 400, 300);

    // Добавление окна подсказки
    AddTooltip (mainWnd.GetHWnd(),
        "Это клиентская область главного окна приложения");

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}
=====
```

Листинг 8.4 (продолжение)

```

HWND InitToolBar(HWND hWnd) {
    /* Здесь такой же текст, как и в листинге 8.3 */

    // Добавление окон подсказки
    AddTooltip (hwndCombo, MAKEINTRESOURCE(IDS_TB_COMBOBOX));
    HWND hwndEdit = GetWindow(hwndCombo, GW_CHILD);
    AddTooltip (hwndEdit, MAKEINTRESOURCE(IDS_EDIT_IN_COMBO));
    return hToolBar;
}
//=====
void UpdateToolBar(ShapeData& sd) {
    /* Здесь такой же текст, как и в листинге 8.3 */
}
//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    /* Здесь такой же текст, как и в листинге 8.3, за исключением фрагмента:
    case IDM_ABOUT:
        MessageBox(hWnd,
            "ToolTip\nVersion 1.0\nCopyright: Finesoft Corporation, 2005.",
            "About ToolTip", MB_OK);
        break;
    */
}
//=====

```

Обратите внимание на следующие особенности программы:

- Файлы KWndEx.h и KWndEx.cpp содержат интерфейс и реализацию класса KWndEx, а также интерфейс и реализацию функций общего применения.

Класс KWndEx отличается от класса KWnd только реализацией конструктора. В новом классе добавлен вызов функции InitCommonControlsEx, регистрирующей оконные классы элементов управления общего пользования.

- Функция AddTooltip, имеющая следующий прототип:

```
void AddTooltip (HWND hWndOwner, LPTSTR lpMsg);
```

предназначена для добавления элемента управления Tooltip к окну hWndOwner. Второму параметру функции передается либо указатель на С-строку, либо идентификатор строкового ресурса в таблице строк приложения.

Хотя элемент управления Tooltip допускает регистрацию нескольких инструментов для окна hWndOwner, для упрощения реализации функция AddTooltip регистрирует только один инструмент, интерпретируемый как вся клиентская область окна hWndOwner.

Первый раз функция AddTooltip вызывается в теле функции WinMain:

```
AddTooltip (mainWnd.GetHWnd(),
    "Это клиентская область главного окна приложения");
```

Второй и третий вызовы включены в текст функции InitToolBar:

```
AddTooltip (hwndCombo, MAKEINTRESOURCE(IDS_TB_COMBOBOX));
HWND hwndEdit = GetWindow(hwndCombo, GW_CHILD);
AddTooltip (hwndEdit, MAKEINTRESOURCE(IDS_EDIT_IN_COMBO));
```

Обратите внимание на использованный способ получения дескриптора окна редактирования, входящего в состав комбинированного списка.

□ **Функция TRACE, имеющая следующий прототип:**

```
void TRACE(LPCTSTR szFormat, ...)
```

предназначена для отладочного вывода в окно Output. Функция принимает переменное количество аргументов, и обращение к ней похоже на обращение к функции printf из библиотеки С. Реализация функции основана на использовании функции OutputDebugString.

Программисты, написавшие хотя бы один проект с библиотекой MFC, сразу оценивают удобство использования макроса TRACE в режиме отладки. К сожалению, Win32 API этот макрос не поддерживает. Но теперь, подключив к любому проекту файлы KwndEx.h и KwndEx.cpp, вы не будете чувствовать себя ущемленными!

Чтобы использовать отладочную печать, не забудьте выполнить два условия. Во-первых, проект должен компилироваться в отладочной конфигурации (Win32 Debug). Во-вторых, программа должна запускаться на выполнение в режиме отладки при помощи команды меню Start Debug ▶ Go или клавиши F5.

Откомпилируйте проект и проверьте, как работает подсказка для комбинированного списка в панели инструментов и для главного окна приложения.

Проверьте также работу функции отладочного вывода TRACE. В блок обработки сообщения WM_NOTIFY можно добавить следующую инструкцию:

```
TRACE("uMsg = %04X, wParam = %08X, lParam = %08X\n", uMsg, wParam, lParam);
```

Теперь, если запустить программу в режиме отладки, благодаря функции TRACE в окно Output будут выводиться значения uMsg, wParam и lParam при каждом поступлении в оконную процедуру сообщения WM_NOTIFY. Отладочная печать будет выглядеть примерно так:

```
uMsg = 004E, wParam = 000000C9, lParam = 0012F768  
uMsg = 004E, wParam = 00009C48, lParam = 00135AC0  
uMsg = 004E, wParam = 00009C48, lParam = 0012FA34
```

Строка состояния

Строка состояния (status bar) — это окно, обычно располагающееся в нижней части главного окна приложения и предназначенное для информирования пользователя о текущем состоянии программы, о выполняемых операциях и режимах. Довольно часто строка состояния используется для отображения подробного описания пунктов меню при наведении на них курсора мыши. Когда меню не просматривается, приложение может выводить другую справочную информацию.

Для создания строки состояния следует использовать функцию CreateStatusWindow:

```
hwndStatusBar = CreateStatusWindow(WS_CHILD | WS_VISIBLE, "", hWndParent,  
ID_STATUS_BAR);
```

В процессе своего выполнения эта функция вызывает функцию CreateWindow, которая создает дочернее окно с родительским окном hWndParent, идентификатором ID_STATUS_BAR и выводит в окно строки состояния текст, заданный вторым параметром. В данном примере будет использоваться пустая строка.

По умолчанию окно строки состояния имеет дополнительные стили CCS_BOTTOM (см. табл. 8.7) и SBARS_SIZEGRIP. Стиль SBARS_SIZEGRIP задает наличие «манипулятора размера» в правом углу строки состояния. Этот декоративный элемент создает область, за которую можно ухватиться при изменении размера окна приложения.

Строка состояния может работать в двух режимах:

- *Стандартный, или многочастный, режим (Multiple-Part Status Bars)*, в котором строка состояния разбивается на несколько частей (полей). В каждом поле выводится отдельная строка текста.
- *Простой режим (Simple Mode Status Bars)*, в котором строка состояния реализована как единый элемент и отображает только одну строку. Простой режим чаще всего используется при отображении справочной информации для пунктов меню, когда пользователь перемещается по меню.

Переключение между режимами осуществляется посылкой сообщения SB_SIMPLE:
`SendMessage(hwndStatusBar, SB_SIMPLE, fMode, 0);`

Когда параметр `fMode` имеет значение `TRUE`, то устанавливается простой режим. При использовании значения `FALSE` строка состояния будет работать в стандартном (многочастном) режиме.

Разделение строки состояния на поля

Если строка состояния используется в стандартном режиме, то вы должны разделить ее на отдельные поля при помощи отсылки сообщения SB_SETPARTS:

`SendMessage(hwndStatusBar, SB_SETPARTS, nParts, (LPARAM)aWidths);`

Здесь параметр `nParts` задает число полей, а `aWidths` — адрес целочисленного массива, каждый элемент которого определяет позицию (в клиентских координатах) правой границы соответствующего поля. Если элемент массива `aWidths` равен `-1`, то границей соответствующего поля считается правая граница строки состояния.

Обычно в каждое поле строки состояния выводится отдельное текстовое сообщение. Но вы можете также разместить в любом поле другой элемент управления, например индикатор процесса (*Progress bar*). При таком размещении полезно знать клиентские координаты этого поля, которые могут быть получены при помощи отправки сообщения SB_GETRECT:

`SendMessage(hwndStatusBar, SB_GETRECT, iPart, (LPARAM)&rect);`

где `iPart` — номер поля, отсчитываемый от нуля, а `rect` — структура типа `RECT`, принимающая координаты поля.

Вывод текстового сообщения

Для отображения текста в строке состояния посыпается сообщение SB_SETTEXT:

`SendMessage(hwndStatusBar, SB_SETTEXT, wParam, (LPARAM)szText);`

где `szText` — указатель на С-строку, а `wParam` задает номер поля в строке состояния и графический стиль для этого поля.

Значение параметра `wParam` может быть задано в виде объединения `iPart | uType`.

Если строка состояния работает в стандартном режиме, то `iPart` содержит номер поля. Если используется простой режим, то параметр `iPart` должен иметь значение 255.

Флаг *uType* может иметь одно из следующих значений:

Значение флага <i>uType</i>	Интерпретация
0 (значение по умолчанию)	Поле рисуется с вдавленной рамкой
SBT_NOBORDERS	Поле рисуется без рамки
SBT_POPOUT	Поле рисуется с выпуклой рамкой
SBT_OWNERDRAW	За прорисовку поля отвечает родительское окно

Максимальная длина строки, выводимой в каждом поле, составляет 127 символов.

Размеры и позиция строки состояния

Оконная процедура дочернего окна строки состояния автоматически устанавливает начальную позицию и размеры этого элемента управления. Ширина строки состояния равна ширине клиентской области родительского окна. Высота строки состояния устанавливается на основе метрик шрифта, выбранного по умолчанию в контекст устройства элемента управления.

В дальнейшем оконная процедура строки состояния автоматически регулирует ее позицию и ширину всякий раз, когда получает сообщение WM_SIZE. Из этого вытекает, что при каждом изменении размеров родительского окна, то есть при получении сообщения WM_SIZE, оконная процедура *WndProc* должна отправить строке состояния такое же сообщение, передав текущие значения параметров wParam и lParam:

```
SendMessage (hwndStatusBar, WM_SIZE, wParam, lParam);
```

Если высота строки состояния по умолчанию вас не устраивает, вы можете ее изменить, послав сообщение SB_SETMINHEIGHT:

```
SendMessage(hwndStatusBar, SB_SETMINHEIGHT, minHeight, 0);
```

где *minHeight* — минимальная высота окна строки состояния в пикселях. На самом деле Windows устанавливает высоту окна строки состояния, равной *minHeight* + 2*wVB, где *wVB* — ширина вертикальной рамки окна.

Возможность увеличить высоту строки состояния бывает полезной, если на ней размещены другие дочерние окна элементов управления (см. ниже приложение ProgressBar).

Уменьшение видимой части клиентской области главного окна

Так же как и панель инструментов, строка состояния занимает часть клиентской области главного окна приложения. К сожалению, Windows не корректирует автоматически размеры и начало координат клиентской области при появлении панели инструментов и строки состояния. В приложении *ToolBar* мы вычисляли прямоугольник, ограничивающий видимую часть клиентской области, с учетом высоты панели инструментов *tbHeight*. Точно так же при использовании строки состояния необходимо учитывать уменьшение высоты видимой части клиентской области на величину, равную высоте строки состояния.

Поддержка просмотра меню

Пользователи, работающие с Windows-программами, привыкли, что в строке состояния отображается информация о назначении выбранного ими пункта меню. Меню посыпает сообщение WM_MENUSELECT, когда пользователь осуществляет

навигацию по его пунктам с помощью мыши или клавиатуры. Поэтому для поддержки отображения справочной информации необходимо обрабатывать это сообщение.

Когда оконная процедура получает сообщение WM_MENUSELECT, из его параметров можно извлечь следующую информацию:

- ❑ Младшее слово параметра wParam содержит идентификатор пункта меню, если это пункт-команда, или индекс подменю (если это пункт-подменю).
- ❑ Старшее слово параметра wParam содержит один из флагов, характеризующих статус выбранного пункта меню. Из всех возможных флагов сейчас нас будет интересовать только флаг MF_POPUP, означающий, что данный пункт открывает всплывающее меню или подменю.
- ❑ Параметр lParam содержит дескриптор меню, которому принадлежит выбранный пункт.

Win32 API содержит функцию MenuHelp, предназначенную для упрощения обработки сообщений WM_MENUSELECT и отображения текста в строке состояния. Технология использования этой функции описана в [1]. Однако, на мой взгляд, эта технология довольно громоздка и неудобна. Здесь предлагается альтернативная технология, в которой функция MenuHelp не применяется.

Ранее мы уже убедились, что бывает очень удобно использовать ресурс таблицы строк при отображении окон подсказок для кнопок панели инструментов. Простота решения базировалась на совпадении идентификаторов кнопок и идентификаторов строк.

Хотелось бы использовать аналогичную идею и для решения рассматриваемой проблемы. Однако если пункты-команды в меню имеют уникальные идентификаторы и мы можем использовать эти же идентификаторы для строк в таблице строк, то с пунктами-подменю ситуация гораздо сложней. В пределах меню каждого уровня пункты-подменю нумеруются начиная с нуля, поэтому ни о какой уникальности не может быть и речи.

Идея решения, предлагаемая здесь, базируется на некоторой договоренности о системе идентификации таких пунктов меню. Сразу уточним, что мы рассматриваем только те меню, которые имеют не более трех уровней. Нулевой уровень — это меню верхнего уровня, подменю в нем имеет первый уровень, а подменю в меню первого уровня получает второй уровень. Большинство Windows-программ удовлетворяет этому ограничению.

Процедура однозначной идентификации пунктов-подменю

Для пунктов-подменю, принадлежащих меню нулевого уровня, в качестве идентификаторов используются индексы, присваиваемые им Windows, то есть 0, 1, 2 и т. д. Эти значения содержатся в младшем слове параметра wParam сообщения WM_MENUSELECT.

Для пунктов-подменю, принадлежащих некоторому меню первого уровня, уникальный идентификатор формируется на базе индекса itemID, извлекаемого из младшего слова параметра wParam, посредством добавления соответствующего смещения:

```
itemID += 100 * (submenuID + 1);
```

где submenuID — индекс рассматриваемого подменю в меню нулевого уровня.

Таким образом, для меню первого уровня с нулевым индексом (`submenuID=0`) входящие в него пункты-подменю получат номера 100, 101, 102 и т. д. Для меню первого уровня с единичным индексом входящие в него пункты-подменю получат номера 200, 201, 202 и т. д. *Такие же идентификаторы* мы будем использовать и для строк в таблице строк приложения.

Из принятого нами ограничения на количество уровней меню вытекает, что все пункты второго (последнего) уровня являются пунктами-командами и поэтому имеют уникальные идентификаторы.

Получив в результате обработки сообщения `WM_MENUSELECT` уникальный идентификатор пункта меню `itemId`, мы можем извлечь из таблицы строк соответствующую символьную строку в текстовый буфер `text`, а затем отобразить этот текст в строке состояния, которая работает в простом режиме. Это может быть реализовано следующими инструкциями:

```
SendMessage(hwndStatusBar, SB_SIMPLE, TRUE, 0);
LoadString(GetModuleHandle(NULL), itemId, text, 200);
SendMessage(hwndStatusBar, SB_SETTEXT, 255, (LPARAM)text);
```

Для демонстрации предложенной технологии разработаем приложение `StatusBar`, являющееся модификацией приложения `ToolBar`. В результате доработки к программе будет добавлена строка состояния, работающая в простом и стандартном режимах.

Приложение StatusBar

Создайте новый проект с именем `StatusBar`. Скопируйте из папки проекта `ToolBar` (см. листинг 8.1) в папку проекта `StatusBar` файлы `ToolBar.cpp`, `ToolBar.rc` и `resource.h`, скорректировав имена первых двух файлов заменой подстроки `ToolBar` на `StatusBar`. Скопируйте также файл `toolbar1.bmp`. Наконец, скопируйте из папки проекта `ToolTip` (см. листинг 8.4) файлы `KWndEx.h` и `KWndEx.cpp`.

Добавьте скопированные файлы в состав проекта. К настройкам проекта на вкладке `Link` нужно добавить библиотеку `comctl32.lib`.

Теперь откройте вкладку `ResourceView` в окне `Workspace`. В списке ресурсов откройте папку `String table` и вызовите редактор таблицы строк двойным щелчком мыши на элементе `String table`.

Добавьте в таблицу строки следующие строки:

Идентификатор	Строка
0	Операции с файлами
1	Параметры просмотра
2	Параметры фигуры
300	Тип фигуры
301	Цвет фигуры
IDM_OPEN	Открыть файл
IDM_CLOSE	Закрыть файл
IDM_SAVE	Сохранить файл
IDM_EXIT	Выход из программы
IDM_SHOW_SHAPE	Показать рисунок
IDM_HIDE_SHAPE	Скрыть рисунок
IDM_RESIZE	Изменение размеров фигуры
IDM_ABOUT	Информация о программе

Обратите внимание на целочисленные идентификаторы 0, 1, 2, 300, 301, назначенные в соответствии с описанной выше процедурой однозначной идентификации пунктов-подменю.

Отредактируйте текст файла StatusBar.cpp так, чтобы он соответствовал листингу 8.5.

Листинг 8.5. Проект StatusBar

```
////////////////////////////////////////////////////////////////
// StatusBar.cpp
#include <windows.h>
#include <commctrl.h>
#include <stdio.h>
#include "KWndEx.h"
#include "resource.h"

#define W 200 // ширина фигуры
#define H 140 // высота фигуры
enum ShapeSize { MAX, MIN };

typedef struct {
    int id_shape; // идентификатор фигуры
    BOOL fRed; // компонент красного цвета
    BOOL fGreen; // компонент зеленого цвета
    BOOL fBlue; // компонент синего цвета
    int id_bright; // идентификатор яркости цвета
} ShapeData;

#define ID_TOOLBAR 201
#define ID_STATUSBAR 202
#define NUM_BUTTONS 11
#define SEPARATOR_WIDTH 10

#define N_PARTS 4

HWND hWndToolBar;
HWND hWndStatusBar;

HWND InitToolBar(HWND hWnd);
void UpdateToolBar(ShapeData& sd);
void UpdateStatusBar(HWND hwnd, ShapeData& sd);

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWndEx mainWnd("StatusBar", hInstance, nCmdShow, WndProc,
        MAKEINTRESOURCE(IDR_MENU1), 100, 100, 400, 300);

    /* Здесь такой же текст, как в листинге 8.1 */
}

HWND InitToolBar(HWND hWnd) {
    /* Здесь такой же текст, как в листинге 8.1 */
}
```

```
void UpdateToolBar(ShapeData& sd) {
    /* Здесь такой же текст, как в листинге 8.1 */
}

//=====
void UpdateStatusBar(HWND hwnd, ShapeData& sd) {

    int paneWidth;
    int aWidths[N_PARTS];
    char text[100];
    RECT rect;
    int brightness;
    int intensity[3] = { 85, 170, 255 };

    GetClientRect(hwnd, &rect);

    paneWidth = rect.right / N_PARTS;
    aWidths [0] = paneWidth;
    aWidths [1] = paneWidth * 2;
    aWidths [2] = paneWidth * 3;
    aWidths [3] = -1;
    SendMessage(hwndStatusBar, SB_SETPARTS, N_PARTS, (LPARAM)aWidths);

    LoadString(GetModuleHandle(NULL), sd.id_shape, text, 100);
    SendMessage(hwndStatusBar, SB_SETTEXT, 0, (LPARAM)text);

    brightness = intensity[sd.id_bright - ID_DARK];

    int red = sd.fRed? brightness : 0;
    sprintf(text, "Red = %d", red);
    SendMessage(hwndStatusBar, SB_SETTEXT, 1, (LPARAM)text);

    int green = sd.fGreen? brightness : 0;
    sprintf(text, "Green = %d", green);
    SendMessage(hwndStatusBar, SB_SETTEXT, 2, (LPARAM)text);

    int blue = sd.fBlue? brightness : 0;
    sprintf(text, "Blue = %d", blue);
    SendMessage(hwndStatusBar, SB_SETTEXT, 3, (LPARAM)text);
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    /* Определения локальных переменных из листинга 8.1 */

    RECT rcSB;      // позиция и размеры окна hWndStatusBar
    int sbHeight;   // высота окна hWndStatusBar
    HMENU hClickMenu;
    int itemID;
    static UINT submenuID;
    char text[200];

    switch (uMsg)
    {
        case WM_CREATE:

            /* Здесь такой же текст, как в листинге 8.1 */

            // Создание строки состояния
```

Листинг 8.5 (продолжение)

```
hwndStatusBar = CreateStatusWindow(WS_CHILD | WS_VISIBLE, "", hWnd,
ID_STATUSBAR);

break;

case WM_SIZE:
SendMessage(hWndToolBar, TB_AUTOSIZE, 0, 0);

UpdateStatusBar(hWnd, shapeData);
SendMessage(hWndStatusBar, WM_SIZE, wParam, lParam);
break;

case WM_NOTIFY:
/* Здесь такой же текст, как в листинге 8.1 */
break;

case WM_MENUSELECT:
itemID = LOWORD(wParam); // меню нулевого уровня
hClickMenu = (HMENU)lParam;

if (HIWORD(wParam) & MF_POPUP) {
    if (hClickMenu == hMenu)
        submenuID = itemID;
    else
        itemID += 100 * (submenuID + 1);
}
else if (!itemID)
    itemID = -1;

SendMessage(hWndStatusBar, SB_SIMPLE, TRUE, 0);
LoadString(GetModuleHandle(NULL), itemID, text, 200);
SendMessage(hWndStatusBar, SB_SETTEXT, 255, (LPARAM)text);
break;

case WM_COMMAND:
switch (LOWORD(wParam))
{
/* Обработка для кодов сообщений IDM_OPEN, IDM_CLOSE, ... , IDM_RESIZE -
как в листинге 8.1 */

case IDM_ABOUT:
MessageBox(hWnd,
"StatusBar\nVersion 1.0\nCopyright: "
"Finesoft Corporation, 2005.",
"About StatusBar", MB_OK);
break;

default:
break;
}

UpdateToolBar(shapeData);
SendMessage(hWndStatusBar, SB_SIMPLE, FALSE, 0);
UpdateStatusBar(hWnd, shapeData);

InvalidateRect(hWnd, NULL, TRUE);
```

```
break;

case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);
    brightness = intensity[shapeData.id_bright - ID_DARK];

    if (bShow) {
        hBrush = CreateSolidBrush(RGB(
            shapeData.fRed? brightness : 0,
            shapeData.fGreen? brightness : 0,
            shapeData.fBlue? brightness : 0));
        hOldBrush = (HBRUSH)SelectObject(hDC, hBrush);

        // Определение позиции и размеров фигуры с учетом
        // уменьшения клиентской области из-за размещения
        // панели инструментов и строки состояния
        GetClientRect(hWnd, &rect);
        GetWindowRect(hwndToolBar, &rcTB);
        tbHeight = rcTB.bottom - rcTB.top;
        GetWindowRect(hwndStatusBar, &rcSB);
        sbHeight = rcSB.bottom - rcSB.top;

        x0 = rect.right / 2;
        y0 = tbHeight + (rect.bottom - tbHeight - sbHeight) / 2;

        // Координаты прямоугольника и эллипса
        if (shapeSize == MIN) {
            x1 = x0 - W/2; y1 = y0 - H/2;
            x2 = x0 + W/2; y2 = y0 + H/2;
        }
        else {
            x1 = 0; y1 = tbHeight;
            x2 = rect.right; y2 = rect.bottom - sbHeight;
        }

        // Координаты ромба
        pt[0].x = (x1 + x2) / 2; pt[0].y = y1;
        pt[1].x = x2; pt[1].y = (y1 + y2) / 2;
        pt[2].x = (x1 + x2) / 2; pt[2].y = y2;
        pt[3].x = x1; pt[3].y = (y1 + y2) / 2;

        switch (shapeData.id_shape) {
        case ID_RECTANGLE:
            Rectangle(hDC, x1, y1, x2, y2); break;
        case ID_RHOMB:
            Polygon(hDC, pt, 4); break;
        case ID_ELLIPSE:
            Ellipse(hDC, x1, y1, x2, y2); break;
        }
        DeleteObject(SelectObject(hDC, hOldBrush));
    }
    EndPaint(hWnd, &ps);
    break;

case WM_DESTROY:
    PostQuitMessage(0);
```

Листинг 8.5 (продолжение)

```

break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}

return 0;
}
/////////////////////////////////////////////////////////////////

```

Приложение использует строку состояния в стандартном и простом режимах.

Действия по формированию строки состояния, работающей в стандартном режиме, инкапсулированы в функцию `UpdateStatusBar`. Эта функция вызывается при обработке сообщений `WM_SIZE` и `WM_COMMAND`.

Строка состояния, работающая в простом режиме, формируется кодом в блоке обработки сообщения `WM_MENUSELECT`:

```

itemID = LOWORD(wParam);
hClickMenu = (HMENU)lParam;

if (HIWORD(wParam) & MF_POPUP)
    if (hClickMenu == hMenu)
        submenuID = itemID;
    else
        itemID += 100 * (submenuID + 1);
}
else if (!itemID)
    itemID = -1;

```

Алгоритм вычисления значения `itemID` реализует описанную выше процедуру однозначной идентификации пунктов-подменю.

Дополнительного пояснения требуют две последние строки в этом фрагменте. В процессе отладки программы выяснилось, что при наведении курсора мыши на разделительную линию (`SEPARATOR`) Windows посылает сообщение `WM_MENUSELECT`, в котором `LOWORD(wParam)` равно нулю. Чтобы исключить интерпретацию этого значения как индекса подменю, добавлена последняя ветвь `else`. К счастью, старшее слово параметра `wParam` в этой ситуации не содержит флага `MF_POPUP`. Это позволяет обнаружить разделитель и присвоить идентификатору `itemID` значение `-1`. Идентификатор со значением `-1` гарантированно отсутствует в таблице строк приложения.

На рис. 8.10 показано окно работающего приложения `StatusBar`.

Рекомендуем вам проверить работу приложения в других режимах.

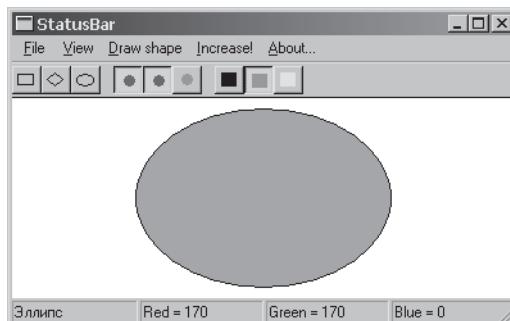


Рис. 8.10. Приложение `StatusBar`. Страна состояния — в стандартном режиме

Другие элементы управления

Из других элементов управления общего пользования мы рассмотрим только Progress bar, Slider и Spin.

Индикатор процесса

Индикатор процесса, или элемент управления Progress bar реализован как дочернее окно, которое может использоваться в приложении для отображения процесса выполнения некоторой длительной операции. Индикатор процесса выглядит как вытянутая и расположенная горизонтально или вертикально прямоугольная область. По мере выполнения операции область заполняется слева направо или снизу вверх небольшими прямоугольниками, что позволяет показать пользователю прогресс в выполнении текущей операции.

Индикатор процесса создается при помощи функции `CreateWindow` или `CreateWindowEx` с указанием идентификатора `PROGRESS_CLASS` в качестве оконного класса, например:

```
hwndProgressBar = CreateWindow(PROGRESS_CLASS, NULL, WS_CHILD | WS_VISIBLE,  
    x, y, width, height, hwndParent, NULL, NULL, NULL);
```

По умолчанию индикатор процесса имеет горизонтальную ориентацию и заполняется отдельными маленькими прямоугольниками, окрашенными системным цветом `COLOR_HIGHLIGHT`¹.

Если при вызове функции `CreateWindow` дополнительно указать стиль `PBS_VERTICAL`, то индикатор будет иметь вертикальную ориентацию. Термин «ориентация» здесь относится только к способу заполнения окна индикатора прямоугольниками: слева направо или снизу вверх. Форма же окна определяется только параметрами `width` и `height`.

Если вы предпочтете использовать сплошное заполнение окна индикатора процесса, без маленьких промежутков между прямоугольниками, то укажите дополнительно стиль `PBS_SMOOTH`.

Реализация процесса и его связи с индикацией полностью лежат на плечах разработчика приложения. Собственно индикатор является лишь средством отображения отношения двух величин.

Интервал (range) индикатора процесса ассоциируется с полным выполнением операции. Он задается как пара целых чисел `wMin` и `wMax`. *Текущая позиция (current position)* отображает смещение от начала интервала и соответствует степени завершения операции. Оконная процедура индикатора процесса использует интервал и текущую позицию, чтобы определить процент заполнения окна индикатора прямоугольниками.

Для управления индикатором процесса используются сообщения, приведенные в табл. 8.12 (полный список сообщений см. в MSDN).

Задавая значения `wMin` и `wMax` при определении интервала, выбирайте их из диапазона от 0 до 65 535. Если вы не определите интервал индикатора с помощью сообщения `PBM_SETRANGE`, то система использует установки по умолчанию, когда `wMin = 0`, а `wMax = 100`.

¹ Цвет, используемый Windows для подсвечивания выделенных элементов в дочерних окнах элементов управления.

Таблица 8.12. Сообщения для управления элементом Progress bar

Код сообщения	wParam	lParam	Описание
PBM_SETRANGE	0	MAKELPARAM (wMin, wMax)	Установка интервала для индикатора
PBM_SETPOS	nNewPos	0	Установка текущей позиции
PBM_DELTAPOS	nInc	0	Изменение текущей позиции прибавлением смещения nInc
PBM_SETSTEP	nStepInc	0	Установка шага приращения для индикатора
PBM_STEPIT	0	0	Изменение текущей позиции прибавлением шага nStepInc
PBM_SETBARCOLOR	0	(COLORREF) clrBar	Установка цвета заполняемых прямоугольников
PBM_SETBKCOLOR	0	(COLORREF) clrBk	Установка цвета фона

Как видно из таблицы, текущее состояние индикатора можно изменять тремя альтернативными способами. Для этого может использоваться сообщение PBM_SETPOS, PBM_DELTAPOS или PBM_STEPIT. В третьем способе текущая позиция изменяется прибавлением шага приращения nStepInc, который можно предварительно установить, отправив сообщение PBM_SETSTEP. Если вы не устанавливаете это значение, система использует значение по умолчанию nStepInc = 10.

Отметим, что если при обработке сообщения PBM_STEPIT индикатор достигнет значения wMax или превысит его, то его текущая позиция сбрасывается в значение wMin и индикатор процесса стартует сначала.

Два последних сообщения из табл. 8.12 позволяют изменить цветовые атрибуты индикатора процесса.

Для демонстрации применения индикатора процесса разработаем приложение ProgressBar, в котором решается задача загрузки некоторого файла. Приложение должно иметь строку состояния с двумя полями. В первом поле выводится текстовое сообщение о проценте выполненной операции. Во втором поле будет располагаться окно индикатора процесса.

Предположим, что размер загружаемого файла равен size байтов и файл читается порциями по DATA_CHUNK байтов. Чтобы упростить связь процесса загрузки файла с отображением степени завершения операции в индикаторе, будем считать, что wMin = 0, wMax = size / DATA_CHUNK, nStepInc = 1, а текущая позиция будет изменяться отправкой сообщения PBM_STEPIT.

Выбирая значение DATA_CHUNK, вы должны понимать, что величина wMax не может превышать значение 65 535.

Создайте новый проект с именем ProgressBar. Добавьте к настройкам проекта на вкладке Link библиотеку comctl32.lib. Скопируйте из папки проекта ToolTip файлы KWndEx.h и KWndEx.cpp, после чего добавьте их в состав проекта. Также добавьте к приложению ресурс меню IDR_MENU1 с одним пунктом. Пункт меню должен иметь имя LoadFile и идентификатор IDM_LOAD_FILE.

Добавьте к проекту файл ProgressBar.cpp с текстом, приведенным в листинге 8.6.

Листинг 8.6. Проект ProgressBar

```
///////////
// ProgressBar.cpp
#include <windows.h>
#include <commctrl.h>
```

```
#include <stdio.h>
#include <fstream>
using namespace std;

#include "KWndEx.h"
#include "resource.h"

#define ID_STATUSBAR 201
#define N_PARTS 2

#define DATA_CHUNK 64
#define FILE_NAME "ProgressBar.cpp"

HWND hwndStatusBar;
HWND hwndProgressBar;

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====================================================
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWndEx mainWnd("ProgressBar", hInstance, nCmdShow, WndProc,
        MAKEINTRESOURCE(IDR_MENU1), 100, 100, 400, 200);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}
//=====================================================
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    PAINTSTRUCT ps;
    char text[100];
    RECT rect, r1;

    int aWidths[N_PARTS];

    ifstream file;
    static int size;           // размер файла в байтах
    static int range;          // диапазон для индикатора процесса
    static int nBytesRead;     // число прочитанных байтов

    double percentage;
    char buf[DATA_CHUNK];

    static int i;
    MSG message;

    switch (uMsg)
    {
    case WM_CREATE:
        // Создание строки состояния
        hwndStatusBar = CreateStatusWindow(WS_CHILD | WS_VISIBLE, "", hWnd,
            ID_STATUSBAR);
        // Увеличиваем высоту строки состояния
```

продолжение ↗

Листинг 8.6 (продолжение)

```

SendMessage(hwndStatusBar, SB_SETMINHEIGHT, 24, 0);

aWidths [0] = 50;
aWidths [1] = -1;
SendMessage(hwndStatusBar, SB_SETPARTS, N_PARTS, (LPARAM)aWidths);
// Извлекаем координаты 1-го поля
SendMessage(hwndStatusBar, SB_GETRECT, 1, (LPARAM)&r1);

// Создание индикатора процесса
hwndProgressBar = CreateWindow(PROGRESS_CLASS, NULL,
    WS_CHILD | WS_VISIBLE,
    r1.left + 3, r1.top + 3, r1.right - r1.left, r1.bottom - r1.top,
    hwndStatusBar, NULL, NULL, NULL);
break;

case WM_SIZE:
    SendMessage (hwndStatusBar, WM_SIZE, wParam, lParam);
break;

case WM_COMMAND:
    switch (LOWORD(wParam))
    {
    case IDM_LOAD_FILE:
        // Открываем файл
        file.open(FILE_NAME, ios::in | ios::binary);
        if (!file) {
            MessageBox(hWnd, "Нет файла", NULL, MB_OK | MB_ICONSTOP);
            break;
        }
        // Определяем размер файла
        file.seekg(0, ios::end);
        size = file.tellg();
        file.seekg(0, ios::beg);

        // Настраиваем параметры индикатора процесса
        range = size / DATA_CHUNK;
        SendMessage(hwndProgressBar, PBM_SETRANGE, 0,
            MAKELPARAM(0, range));
        SendMessage(hwndProgressBar, PBM_SETSTEP, 1, 0);
        SendMessage(hwndProgressBar, PBM_SETPOS, 0, 0);
        InvalidateRect(hWnd, NULL, FALSE);

        nBytesRead = 0;
        // Загрузка файла
        while (!file.eof()) {

            // Чтение порции данных
            file.read(buf, DATA_CHUNK);
            nBytesRead += DATA_CHUNK;

            if (file.eof()) break;

            // Здесь возможен вызов процедуры для обработки порции данных
            // ...

            // Отображение процента выполненной операции
            percentage = 100.0 * nBytesRead / size;
        }
    }
}

```

```
sprintf(text, " %.0f %%", percentage);
SendMessage(hwndStatusBar, SB_SETTEXT, 0, (LPARAM)text);

SendMessage(hwndProgressBar, PBM_STEPIT, 0, 0);
Sleep(100);

// Предотвращение зависания приложения
if(PeekMessage(&message, hWnd, 0, 0, PM_REMOVE)) {
    TranslateMessage(&message);
    DispatchMessage(&message);
}

SendMessage(hwndStatusBar, SB_SETTEXT, 0, (LPARAM)" 100 %");
MessageBox(hWnd, "Файл загружен.", "Завершение операции",
    MB_OK | MB_ICONINFORMATION);
// Сброс индикатора процесса в исходное положение
SendMessage(hwndProgressBar, PBM_SETPOS, 0, 0);
SendMessage(hwndStatusBar, SB_SETTEXT, 0, (LPARAM)" Ready");
break;

case IDM_EXIT:
    SendMessage(hWnd, WM_DESTROY, 0, 0);
    break;

default:
    break;
}
break;

case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);
    GetClientRect(hWnd, &rect);
    sprintf(text, "size = %d, DATA_CHUNK = %d, range = %d",
        size, DATA_CHUNK, range);
    DrawText(hDC, text, -1, &rect, DT_SINGLELINE | DT_CENTER | 
        DT_VCENTER);
    EndPaint(hWnd, &ps);
    break;

case WM_DESTROY:
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
////////////////////////////////////////////////////////////////
```

Имя загружаемого файла в этом примере задано следующей директивой:

```
#define FILE_NAME "ProgressBar.cpp"
```

В приложении используется строка состояния с увеличенной высотой, чтобы лучше смотрелся встроенный индикатор процесса.

Прежде чем поместить индикатор процесса во второе поле строки состояния, мы узнаем координаты этого поля, посылая соответствующее сообщение:

```
SendMessage(hwndStatusBar, SB_GETRECT, 1, (LPARAM)&r1);
```

Загрузка файла осуществляется в цикле `while`. На каждой итерации цикла читается очередная порция данных, после чего окну индикатора процесса посыпается сообщение `PBM_STEPIIT`.

В теле цикла имеется инструкция `Sleep(100)`, которая приостанавливает выполнение программы на 100 мс. Преднамеренное замедление работы программы преследует демонстрационные цели. Дело в том, что загружаемый файл `ProgressBar.cpp` имеет небольшие размеры, и поэтому без этой задержки индикатор процесса будет заполнен практически мгновенно.

Также следует обратить внимание на фрагмент кода

```
if(PeekMessage(&message, hWnd, 0, 0, PM_REMOVE)) {
    TranslateMessage(&message);
    DispatchMessage(&message);
}
```

Он предотвращает «зависание» приложения на время, в течение которого происходит загрузка файла. Об этом инструментальном приеме мы рассказывали в главе 1.

На рис. 8.11 показан вид работающего приложения `ProgressBar` после выбора команды меню `LoadFile`.

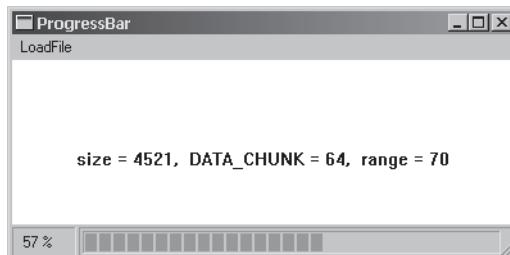


Рис. 8.11. Загрузка файла в приложении `ProgressBar`

Посмотрите, как изменится работа индикатора, если при его создании в функции `CreateWindow` задать дополнительный стиль `PBS_SMOOTH`.

Регулятор

Регулятор, или элемент управления `Slider`, который ранее назывался `Trackbar`, представляет собой окно с линейкой и перемещаемым по ней ползунком. Внешне этот элемент управления напоминает регуляторы тембра, используемые в бытовой радиоаппаратуре. Подобный регулятор дает возможность пользователю выбирать дискретные значения в заданном диапазоне.

Создание регулятора

Элемент управления `Slider` может быть создан с помощью функции `CreateWindow` или `CreateWindowEx` с указанием идентификатора `TRACKBAR_CLASS` в качестве оконного класса, например:

```
hwndSlider = CreateWindow(TRACKBAR_CLASS, NULL, WS_CHILD | WS_VISIBLE, x, y,
    width, height, hwndParent, NULL, NULL);
```

Кроме стилей **WS_CHILD** и **WS_VISIBLE** можно также задавать дополнительные стили, определяющие внешний вид элемента управления (табл. 8.13).

Таблица 8.13. Стили элемента управления Slider

Стиль	Описание
TBS_HORZ	Линейка имеет горизонтальную ориентацию (стиль по умолчанию)
TBS_VERT	Линейка имеет вертикальную ориентацию
TBS_AUTOTICKS	Линейка имеет метки для всех значений в заданном диапазоне значений. Без этого стиля линейка может иметь метки только для начального и конечного положений ползунка
TBS_NOTICKS	Исключает отображение каких-либо меток, в том числе и для начального и конечного положений ползунка
TBS_RIGHT	Элемент отображает метки справа (снизу) от линейки; ползунок направлен в ту же сторону. Этот стиль используется по умолчанию
TBS_LEFT	Элемент отображает метки слева (сверху) от линейки; ползунок направлен в ту же сторону
TBS_BOTH	Элемент отображает метки с обеих сторон; ползунок имеет прямоугольную форму
TBS_TOOLTIPS	Поддерживается всплывающая подсказка, отображающая текущую позицию ползунка

Альтернативный способ создания регулятора — с помощью редактора диалоговых окон на этапе визуального проектирования формы диалога. В этом случае вы выбираете щелчком мыши элемент *Slider* на панели инструментов *Controls* и повторным щелчком помещаете регулятор в нужное место формы диалога.

Размеры элемента управления при размещении на форме диалога, как обычно, регулируются с помощью мыши или при помощи команд подменю *Layout*.

Затем нужно вызвать окно свойств *Slider Properties* и на вкладке *General* в поле **ID** ввести идентификатор элемента управления. На вкладке *Styles*, показанной на рис. 8.12, можно задать дополнительные свойства регулятора.

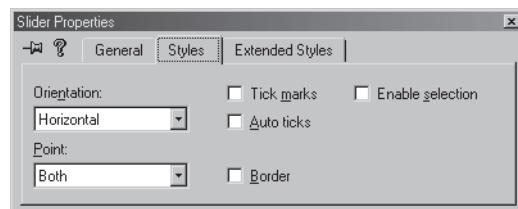


Рис. 8.12. Свойства элемента Slider на вкладке Styles

Открывающийся список **Orientation** позволяет выбрать ориентацию элемента управления: **Horizontal** (по умолчанию) или **Vertical**. Другой список, **Point**, предназначен для выбора стиля размещения меток: **Both** (по умолчанию), **Top/Left** или **Bottom/Right**. Флажок **Tick marks** задает наличие меток, флажок **Auto ticks** задает стиль **TBS_AUTOTICKS**, флажок **Border** определяет наличие рамки у элемента управления.

Создание регулятора с помощью редактора диалоговых окон является более простым способом. Именно эту технологию мы используем в рассматриваемом ниже примере разработки приложения Trackbar.

Параметры и состояние регулятора

С элементом управления Slider связан внутренний счетчик, определяющий его поведение. Счетчик имеет минимальное значение `wMin` и максимальное значение `wMax`. По умолчанию `wMin = 0`, а `wMax = 100`. Вы можете изменить диапазон регулятора, послав ему сообщение `TBM_SETRANGE`.

Текущее состояние счетчика однозначно связано с текущей позицией ползунка. Пользователь может перемещать ползунок по линейке регулятора как с помощью мыши, так и с помощью клавиатуры. Второй вариант работы предполагает, что регулятор имеет фокус ввода.

Минимальный интервал, на который можно изменить состояние регулятора с помощью клавиш со стрелками, называется «строкой» (*line*). По умолчанию он равен единице. Размер «строки» можно изменить, послав регулятору сообщение `TBM_SETLINESIZE`.

Более крупный интервал, на который можно изменить состояние регулятора с помощью клавиши *Page Up* или *Page Down*, называется «страницей» (*page*). По умолчанию размер «страницы» равен одной пятой части диапазона регулятора. Размер «страницы» можно изменить, послав регулятору сообщение `TBM_SETPAGESIZE`.

Если регулятор создан со стилем Auto ticks (`TBS_AUTOTICKS`), то линейка имеет метки во всем диапазоне значений с шагом `wFreq`, который по умолчанию равен единице. Вы можете изменить этот шаг, послав сообщение `TBM_SETTICFREQ`.

Действия пользователя и уведомляющие сообщения

Регулятор уведомляет свое родительское окно о действиях пользователя, посыпая сообщение `WM_HSCROLL` или `WM_VSCROLL` — в зависимости от ориентации элемента управления (Horizontal или Vertical). В любом случае в младшем слове параметра `wParam` содержится код уведомления, а параметр `lParam` содержит дескриптор регулятора. Возможные коды уведомления приведены в табл. 8.14. Для кодов `TB_THUMBPOSITION` и `TB_THUMBTRACK` старшее слово параметра `wParam` содержит позицию ползунка.

Таблица 8.14. Коды уведомления от элемента управления Slider

Код уведомления	Значение	Описание
<code>TB_LINEUP</code>	0	Нажата клавиша «стрелка влево» (<code>VK_LEFT</code>) или клавиша «стрелка вверх» (<code>VK_UP</code>) — ползунок сдвигается на величину «строки»
<code>TB_LINEDOWN</code>	1	Нажата клавиша «стрелка вправо» (<code>VK_RIGHT</code>) или клавиша «стрелка вниз» (<code>VK_DOWN</code>) — ползунок сдвигается на величину «строки»
<code>TB_PAGEUP</code>	2	Нажата клавиша « <i>Page Up</i> » (<code>VK_PRIOR</code>) или щелчок мышью на линейке левее или выше ползунка — ползунок сдвигается на величину «страницы» (левее — для горизонтальной ориентации регулятора, выше — для вертикальной ориентации)
<code>TB_PAGEDOWN</code>	3	Нажата клавиша « <i>Page Down</i> » (<code>VK_NEXT</code>) или щелчок мышью на линейке правее или ниже ползунка — ползунок сдвигается на величину «страницы»

Код уведомления	Значение	Описание
TB_THUMBPOSITION	4	Пользователь отпустил кнопку мыши после перемещения ползунка (это сообщение следует всегда после сообщения TB_THUMBTRACK)
TB_THUMBTRACK	5	Ползунок перемещается с помощью мыши
TB_TOP	6	Нажата клавиша «Home» (VK_HOME) — ползунок устанавливается в крайнее левое (верхнее) положение, соответствующее значению wMin
TB_BOTTOM	7	Нажата клавиша «End» (VK_END) — ползунок устанавливается в крайнее правое (нижнее) положение, соответствующее значению wMax

Часто приложение может обойтись без обработки этих сообщений, так как с помощью управляющего сообщения TBM_GETPOS (см. ниже) можно легко узнать текущую позицию ползунка, и в большинстве случаев этого оказывается достаточно.

Управляющие сообщения

Для управления регулятором предусмотрено несколько десятков сообщений, символические коды которых начинаются с префикса TBM_. В табл. 8.15 приведены некоторые из этих сообщений.

Таблица 8.15. Сообщения для управления элементом Slider

Сообщение	wParam	lParam	Описание
TBM_GETPOS	0	0	Возвращает текущую позицию ползунка
TBM_SETPOS	fRedraw	newPos	Устанавливает новую позицию ползунка. Если fRedraw равно TRUE, регулятор перерисовывается после этого
TBM_SETRANGE	fRedraw	MAKELPARAM (wMin, wMax)	Устанавливает диапазон регулятора
TBM_SETTICFREQ	wFreq	0	Устанавливает шаг меток
TBM_SETLINESIZE	0	nLineSize	Устанавливает размер «строки»
TBM_SETPAGESIZE	0	nPageSize	Устанавливает размер «страницы»

Приложение TrackBar

Для демонстрации применения регуляторов разработаем приложение TrackBar, которое является клоном приложения ModelessDlg, рассмотренного в главе 7. Приложение ModelessDlg позволяло изменять цвет фона окна с помощью полос прокрутки. В нашем новом приложении вместо полос прокрутки будут применены элементы управления Slider.

Создайте новый проект с именем TrackBar. Скопируйте из папки проекта ModelessDlg (см. листинг 7.6) в папку проекта TrackBar файлы ModelessDlg.cpp, ModelessDlg.rc и resource.h, скорректировав имена первых двух файлов заменой подстроки ModelessDlg на TrackBar. Также скопируйте из папки проекта ToolTip (листинг 8.4) файлы KWndEx.cpp и KWndEx.h. Все скопированные файлы нужно добавить в состав проекта. Также к настройкам проекта на вкладке Link следует добавить библиотеку comctl32.lib.

Откройте вкладку ResourceView в окне Workspace. В списке ресурсов откройте папку Dialog и вызовите редактор диалоговых окон двойным щелчком мыши на эле-

менте IDD_MODELESS. Удалите все элементы управления с формы диалога. Разместите три надписи и три элемента управления Slider, как показано на рис. 8.13.

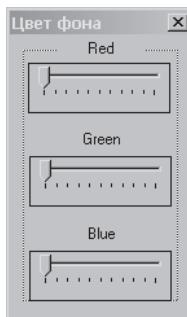


Рис. 8.13. Форма диалога после размещения элементов управления

Для элементов управления установите следующие атрибуты:

Элемент	ID	Caption	Свойства
Static text	IDC_STATIC_RED	Red	По умолчанию
Static text	IDC_STATIC_GREEN	Green	По умолчанию
Static text	IDC_STATIC_BLUE	Blue	По умолчанию
Slider	IDC_SLIDER_RED	—	Стиль меток — Bottom/Right. Установлены флажки Tick marks, Auto ticks, Border
Slider	IDC_SLIDER_GREEN	—	
Slider	IDC_SLIDER_BLUE	—	Остальные свойства — по умолчанию

Отредактируйте исходный код в файле TrackBar.cpp, чтобы он соответствовал листингу 8.7.

Листинг 8.7. Проект TrackBar

```
////////////////////////////////////////////////////////////////
// TrackBar.cpp
#include <windows.h>
#include <stdio.h>
#include <commctrl.h>

#include "KWndEx.h"
#include "resource.h"

enum UserMsg { UM_CHANGE = WM_USER+1 };

HWND hModelessDlg;
RECT rcWork; // прямоугольник рабочей области
int rgb[3]; // интенсивность для R-, G-, B-компонентов цвета

BOOL CALLBACK ModelessDlgProc(HWND, UINT, WPARAM, LPARAM);
void AdjustDlgPlace(HWND, HWND);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
=====
```

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;

    KWndEx mainWnd("TrackBar", hInstance, nCmdShow, WndProc,
        NULL, 100, 100, 400, 270);

    while (GetMessage(&msg, NULL, 0, 0)) {
        if (!IsDialogMessage(hModelessDlg, &msg)) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
    return msg.wParam;
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    /* Здесь такой же текст, как в листинге 7.6 */
}

//=====
BOOL CALLBACK ModelessDlgProc(HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    static HWND hSliderRed;
    static HWND hSliderGreen;
    static HWND hSliderBlue;
    static HWND hStaticRed;
    static HWND hStaticGreen;
    static HWND hStaticBlue;
    char text[100];

    switch (uMsg) {
    case WM_INITDIALOG:

        // Определяем дескрипторы элементов управления Slider
        hSliderRed = GetDlgItem(hDlg, IDC_SLIDER_RED);
        hSliderGreen = GetDlgItem(hDlg, IDC_SLIDER_GREEN);
        hSliderBlue = GetDlgItem(hDlg, IDC_SLIDER_BLUE);
        hStaticRed = GetDlgItem(hDlg, IDC_STATIC_RED);
        hStaticGreen = GetDlgItem(hDlg, IDC_STATIC_GREEN);
        hStaticBlue = GetDlgItem(hDlg, IDC_STATIC_BLUE);

        // Инициализация элементов управления Slider
        SendMessage(hSliderRed, TBM_SETRANGE, TRUE, MAKELONG(0, 255));
        SendMessage(hSliderRed, TBM_SETTICFREQ, 32, 0);
        SendMessage(hSliderGreen, TBM_SETRANGE, TRUE, MAKELONG(0, 255));
        SendMessage(hSliderGreen, TBM_SETTICFREQ, 32, 0);
        SendMessage(hSliderBlue, TBM_SETRANGE, TRUE, MAKELONG(0, 255));
        SendMessage(hSliderBlue, TBM_SETTICFREQ, 32, 0);

        // Вывод фона окна для начального состояния регуляторов
        SendMessage(hDlg, WM_HSCROLL, 0, 0);
    }
}
```

продолжение ⇨

Листинг 8.7 (продолжение)

```

    return TRUE;

    case WM_HSCROLL:
        // Получаем текущие позиции элементов управления Slider
        rgb[0] = SendMessage(hSliderRed, TBM_GETPOS, 0, 0);
        sprintf(text, "Red = %d", rgb[0]);
        SetWindowText(hStaticRed, text);

        rgb[1] = SendMessage(hSliderGreen, TBM_GETPOS, 0, 0);
        sprintf(text, "Green = %d", rgb[1]);
        SetWindowText(hStaticGreen, text);

        rgb[2] = SendMessage(hSliderBlue, TBM_GETPOS, 0, 0);
        sprintf(text, "Blue = %d", rgb[2]);
        SetWindowText(hStaticBlue, text);

        // Сообщение родительскому окну об изменении цвета фона
        SendMessage(GetParent(hDlg), UM_CHANGE, 0, 0);
        break;
    }
    return FALSE;
}

//=====
void AdjustDlgPlace(HWND hParent, HWND hDlg) {
    /* Здесь такой же текст, как в листинге 7.6 */
}
/////////////////////////////////////////////////////////////////

```

В блоке обработки сообщения **WM_INITDIALOG** всем трем регуляторам для установки диапазона и шага меток посылаются управляющие сообщения **TBM_SETRANGE** и **TBM_SETTICFREQ**.

В блоке обработки сообщения **WM_HSCROLL** мы узнаем текущую позицию для всех трех ползунков, отправляя трижды сообщение **TBM_GETPOS**. Полученные значения **rgb[0]**, **rgb[1]** и **rgb[2]** используются для вывода текстовой информации в элементы управления **hStaticRed**, **hStaticGreen** и **hStaticBlue**, а также для формирования цвета фона окна. Цвет фона окна устанавливается оконной процедурой **WndProc**, когда она получает сообщение **UM_CHANGE**. Именно поэтому далее в коде программы следует отправка сообщения:

```
SendMessage(GetParent(hDlg), UM_CHANGE, 0, 0);
```

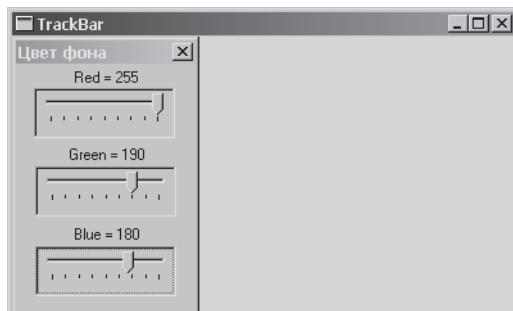


Рис. 8.14. Окно приложения TrackBar

Код обработки сообщения UM_CHANGE, находящийся в теле оконной процедуры WndProc, здесь не приведен, поскольку все тело процедуры повторяет текст, заимствованный из листинга 7.6.

На рис. 8.14 показан вид работающего приложения.

В настоящий момент ползунки регуляторов задают кремовый цвет фона окна.

Счетчик и поле с прокруткой

Элемент управления *счетчик* (Spin) реализован как две кнопки со стрелками, с помощью которых пользователь может увеличивать или уменьшать некоторое числовое значение. Значение, связанное со счетчиком, называется его *текущей позицией*.

Кроме этого счетчик можно ассоциировать с другим элементом управления, называемым *приятельским окном* (*buddy window*). Чаще всего таким окном является окно редактирования. Комбинацию счетчика с окном редактирования называют также полем с прокруткой. Поле с прокруткой воспринимается пользователем как единый элемент управления. Содержимое окна редактирования в таком элементе отображает текущую позицию счетчика (рис. 8.15).



Рис. 8.15. Поле с прокруткой

Не имея ассоцииированного с ним окна, счетчик функционирует как упрощенный вариант полосы прокрутки. Например, в элементе управления Tab control счетчик используется для осуществления доступа к дополнительным вкладкам (рис. 8.16).

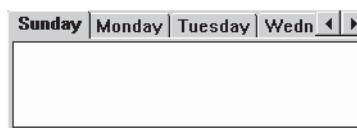


Рис. 8.16. Элемент управления Spin в правом верхнем углу элемента управления Tab control

Создание счетчика

Счетчик можно создать несколькими способами. В первом способе используется вызов функции CreateWindowEx с указанием оконного класса UPDOWN_CLASS. Во втором способе вызывается функция CreateUpDownControl, создающая счетчик и одновременно определяющая его минимальную, максимальную и текущую позиции, а также его приятельское окно.

Третий способ создания счетчика — при помощи редактора диалоговых окон на этапе визуального проектирования формы диалога. Элемент управления Spin выбирается с помощью мыши на панели инструментов Controls и просто помещается в нужное место формы диалога.

Затем надо вызвать окно свойств Spin Properties и на вкладке General в поле ID указать идентификатор элемента управления. На вкладке Styles, показанной на рис. 8.17, можно задать дополнительные свойства счетчика.

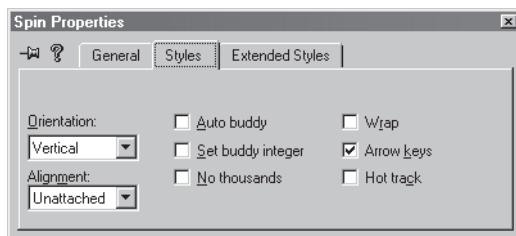


Рис. 8.17. Свойства элемента Spin на вкладке Styles

Список Orientation предназначен для выбора ориентации элемента управления. По умолчанию используется значение Vertical, но можно также установить значение Horizontal.

Список Alignment позволяет выбрать один из трех стилей размещения счетчика:

Alignment	Описание размещения
Unattached	Счетчик располагается рядом с ассоциированным окном (так, как вы поместили его на форму диалога)
Left	Счетчик располагается в левой части ассоциированного окна (уменьшая тем самым его клиентскую область)
Right	Счетчик располагается в правой части ассоциированного окна (уменьшая тем самым его клиентскую область)

Флажки на вкладке Styles позволяют задать дополнительные параметры:

Флажок	Интерпретация
Auto buddy	Автоматический выбор в качестве ассоциированного окна ближайшего предыдущего элемента управления в файле описания ресурсов
Set buddy integer	Вместе с предыдущим флажком определяет синхронную работу счетчика и ассоциированного окна: любое изменение позиции счетчика сразу отображается в ассоциированном окне. Аналогично при вводе в ассоциированное окно допустимого целого числа оно устанавливает новую позицию счетчика
No thousands	Если этот флажок выключен, в изображение десятичного числа после каждого трех цифр вставляется пробел, если включен — пробел не вставляется
Wrap	По умолчанию текущая позиция счетчика не изменяется, если пользователь пытается перейти максимальное (при увеличении) или минимальное (при уменьшении) значение. Если установлен данный флажок, то счетчик работает как циклический. После максимального значения текущим становится минимальное, и наоборот
Arrow keys	Поддерживается управление счетчиком с помощью клавиш «стрелка вверх» и «стрелка вниз» (независимо от ориентации элемента управления)

Таким образом, при создании поля с прокруткой необходимо всегда выполнять два правила. Во-первых, элемент управления Spin нужно поместить на форму диалога сразу вслед за размещением приятельского окна редактирования. Во вторых, надо установить флажки Auto buddy и Set buddy integer. Отметка флажка Arrow keys установлена по умолчанию, и эту установку следует сохранить.

Следует отметить высокий «интеллект» поля с прокруткой. Если пользователь введет в окно редактирования числовое значение больше максимально допустимого или меньше минимально допустимого, то позиция счетчика будет зафиксирована на соответствующей границе диапазона. Правда, пользователь узнает об этом только после очередного воздействия на стрелки счетчика.

При попытке ввести в окно редактирования нецифровые символы счетчик сохраняет текущую позицию.

Сообщения от поля с прокруткой

При нажатии одной из стрелок элемента управления Spin посыпает своему родительскому окну сообщение WM_VSCROLL или WM_HSCROLL (в зависимости от ориентации счетчика), в котором младшее слово параметра wParam содержит код SB_THUMBPOSITION. Кроме того, счетчик посыпает уведомляющее сообщение UDN_DELTAPOS в форме сообщения WM_NOTIFY.

Обычно в приложении нет необходимости обрабатывать все сообщения. Часто бывает достаточно получить текущую позицию счетчика, обрабатывая сообщение WM_VSCROLL или WM_HSCROLL. Это можно сделать, отправив элементу Spin управляющее сообщение UDM_GETPOS.

При непосредственном клавиатурном вводе нового числа в окно редактирования элемент Edit Box посыпает родительскому окну сообщение WM_COMMAND с кодом уведомления EN_CHANGE. Если вы хотите, чтобы приложение немедленно отреагировало на изменившуюся текущую позицию счетчика (а он отслеживает все изменения в приятельском окне), то предусмотрите обработку этого сообщения. Пример такой обработки приведен ниже в приложении Spinner.

Управляющие сообщения

Для управления счетчиком предусмотрено более двадцати сообщений, символические коды которых начинаются с префикса UDM_. В табл. 8.16 приведены некоторые сообщения, чаще всего используемые в работе.

Таблица 8.16. Сообщения для управления элементом Spin

Сообщение	wParam	lParam	Описание
UDM_GETPOS	0	0	Возвращает текущую позицию счетчика
UDM_SETPOS	0	MAKELPARAM (nPos, 0)	Устанавливает новую позицию счетчика
UDM_SETBASE	nBase	0	Устанавливает систему счисления. nBase должно быть равно 10 или 16. По умолчанию установлена десятичная система счисления
UDM_SETRANGE	0	MAKELPARAM (nUpper, nLower)	Устанавливает минимальную и максимальную позиции для счетчика (в интервале -0x7FFF..0x7FFF)

Если после создания счетчика не определить его диапазон при помощи сообщения UDM_SETRANGE, то будет использоваться диапазон по умолчанию со значениями nLower=100 и nUpper=0. В этом случае «стрелка вверх» вызывает уменьшение значения счетчика, а «стрелка вниз» — увеличение значения. Чем обусловлен такой оригинальный диапазон по умолчанию, в котором максимум меньше, чем минимум, в справочных материалах MSDN не поясняется.

Сообщения, перечисленные в табл. 8.16, работают с 16-разрядной версией элемента управления Spin.

Библиотека Comctl32.dll, начиная с версии 4.71, поддерживает 32-разрядную версию счетчика, который управляет сообщениями, приведенными в табл. 8.17.

Таблица 8.17. Сообщения для управления 32-разрядной версией элемента Spin

Сообщение	wParam	lParam	Описание
UDM_GETPOS32	0	(LPBOOL) pfError	Возвращает текущую позицию счетчика. pfError — указатель на булеву переменную, которая получает значение FALSE, если значение успешно получено, и TRUE — в случае ошибки
UDM_SETPOS32	0	nPos	Устанавливает новую позицию счетчика
UDM_SETRANGE32	iLow	iHigh	Устанавливает минимальную и максимальную позиции для счетчика в интервале от –0x7FFFFFFF до 0x7FFFFFFF

Приложение Spinner

Для демонстрации применения поля с прокруткой разработаем приложение Spinner, которое является модификацией приложения TrackBar, рассмотренного выше. Приложение TrackBar позволяло изменять цвет фона окна с помощью регуляторов.

В нашем новом приложении вместо каждого регулятора будет применено поле с прокруткой.

Создайте новый проект с именем Spinner. Скопируйте из папки проекта TrackBar (см. листинг 8.7) в папку проекта Spinner файлы с расширениями .cpp, .h и .rc, скорректировав их имена заменой подстроки TrackBar на Spinner. Добавьте скопированные файлы в состав проекта. Также к настройкам проекта на вкладке Link надо добавить библиотеку comctl32.lib. Затем откройте вкладку ResourceView в окне Workspace. В списке ресурсов откройте папку Dialog и вызовите редактор диалоговых окон двойным щелчком мыши на элементе IDD_MODELESS.

Удалите элементы управления Slider с формы диалога. Разместите вместо них три пары элементов (элемент Edit Box и рядом элемент Spin), как показано на рис. 8.18. В каждой паре указанных элементов счетчик помещается на форму диалога сразу после установки приятельского окна редактирования.

Для расположенных элементов управления установите следующие атрибуты:

Элемент	ID	Свойства
Edit Box	IDC_EDIT_RED	Align text — Centered. Остальные свойства — по умолчанию
Edit Box	IDC_EDIT_GREEN	
Edit Box	IDC_EDIT_BLUE	
Spin	IDC_SPIN_RED	Alignment — Right. Установлены флажки Auto buddy, Set buddy integer. Остальные свойства — по умолчанию
Spin	IDC_SPIN_GREEN	
Spin	IDC_SPIN_BLUE	

Приведите текст файла Spinner.cpp к виду, соответствующему листингу 8.8.

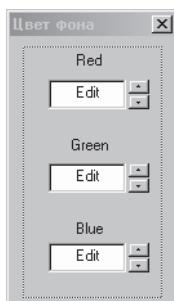


Рис. 8.18. Форма диалога после размещения элементов управления

Листинг 8.8. Проект Spinner

```
////////////////////////////////////////////////////////////////
// Spinner.cpp
#include <windows.h>
#include <stdio.h>
#include <commctrl.h>
#include "KWndEx.h"
#include "resource.h"

enum UserMsg { UM_CHANGE = WM_USER+1 };

HWND hModelessDlg;
RECT rcWork; // прямоугольник рабочей области
int rgb[3]; // интенсивность для R-, G-, B-компонентов цвета

BOOL CALLBACK ModelessDlgProc(HWND, UINT, WPARAM, LPARAM);
void AdjustDlgPlace(HWND, HWND);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====================================================
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;

    KWndEx mainWnd("Spinner", hInstance, nCmdShow, WndProc,
        NULL, 100, 100, 400, 260);

    while (GetMessage(&msg, NULL, 0, 0)) {
        if (!IsDialogMessage(hModelessDlg, &msg)) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
    return msg.wParam;
}

//=====================================================
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    /* Здесь такой же текст, как в листинге 7.6 */
}

//=====================================================
BOOL CALLBACK ModelessDlgProc(HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    static HWND hSpinRed;
    static HWND hSpinGreen;
    static HWND hSpinBlue;
    static HWND hStaticRed;
    static HWND hStaticGreen;
    static HWND hStaticBlue;

    switch (uMsg) {
    case WM_INITDIALOG:
        // Определяем дескрипторы элементов управления Spin
        hSpinRed = GetDlgItem(hDlg, IDC_SPIN_RED);
        hSpinGreen = GetDlgItem(hDlg, IDC_SPIN_GREEN);
        hSpinBlue = GetDlgItem(hDlg, IDC_SPIN_BLUE);
```

продолжение ↗

Листинг 8.8 (продолжение)

```

hStaticRed = GetDlgItem(hDlg, IDC_STATIC_RED);
hStaticGreen = GetDlgItem(hDlg, IDC_STATIC_GREEN);
hStaticBlue = GetDlgItem(hDlg, IDC_STATIC_BLUE);

// Инициализация элементов управления Spin
SendMessage(hSpinRed, UDM_SETRANGE, TRUE, MAKELPARAM(255, 0));
SendMessage(hSpinGreen, UDM_SETRANGE, TRUE, MAKELPARAM(255, 0));
SendMessage(hSpinBlue, UDM_SETRANGE, TRUE, MAKELPARAM(255, 0));
return TRUE;

case WM_VSCROLL:
// Получаем текущие позиции элементов управления Spin
rgb[0] = SendMessage(hSpinRed, UDM_GETPOS, 0, 0);
rgb[1] = SendMessage(hSpinGreen, UDM_GETPOS, 0, 0);
rgb[2] = SendMessage(hSpinBlue, UDM_GETPOS, 0, 0);

// Сообщение родительскому окну об изменении цвета фона
SendMessage(GetParent(hDlg), UM_CHANGE, 0, 0);
break;

case WM_COMMAND:
// Обновление фона основного окна после ввода
// через приятельское окно редактирования
if (HIWORD(wParam) == EN_CHANGE)
    SendMessage(hDlg, WM_VSCROLL, 0, 0);
break;
}
return FALSE;
}

//=====
void AdjustDlgPlace(HWND hParent, HWND hDlg) {
/* Здесь такой же текст, как в листинге 7.6 */
}
=====
```

Использование поля с прокруткой, как видите, очень просто. Обратите внимание на обработку сообщения **WM_COMMAND** с кодом уведомления **EN_CHANGE**. Мы отправляем диалоговому окну сообщение **WM_VSCROLL**, чтобы приложение немедленно отреагировало на изменившееся вследствие изменения текста в приятельском окне состояние счетчика.

На рис. 8.19 показано окно работающего приложения.

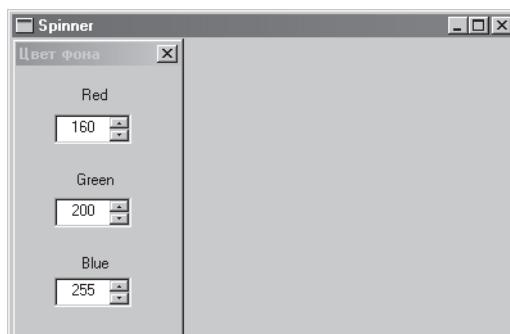


Рис. 8.19. Приложение Spinner. С помощью полей прокрутки задан голубой цвет фона окна

9

Многозадачность

Эта глава содержит краткое изложение поддержки многозадачности в Win32 API. Более подробную информацию вы можете найти в книге [5] или в справочных материалах MSDN.

Объекты ядра

Современные процессоры позволяют выполнять программы в одном из двух режимов: *Kernel* (режим ядра) или *User* (режим пользователя). В режиме ядра приложению, кроме всего прочего, разрешено выделять память для других приложений. Поэтому изменить важные данные в памяти компьютера, например, управляющие распределением памяти, может только приложение, работающее в режиме ядра.

Архитектура операционной системы Windows NT/2000 состоит из двух основных частей: привилегированной подсистемы режима ядра (*privileged kernel mode part*) и непривилегированной подсистемы пользовательского режима (*nonprivileged user mode part*).

Подсистема ядра содержит следующие компоненты:

- HAL (Hardware Abstraction Layer) — уровень, абстрагирующий другие компоненты от аппаратных различий, зависимых от платформы.
- Микроядро (MicroKernel), которое отвечает за планирование потоков, переключение задач, обработку прерываний и исключительных ситуаций, много-процессорную синхронизацию.
- Драйверы устройств — драйверы оборудования, файловой системы и сетевой поддержки, реализующие пользовательские функции ввода-вывода.
- Управление окнами и графическая подсистема реализуют функции графического интерфейса.
- Исполнительная часть отвечает за управление памятью, управление процессами и потоками, безопасность, ввод-вывод данных и взаимодействия между процессами.

Объекты ядра используются системой и приложениями для управления самыми разнообразными ресурсами, например процессами, потоками, файлами, семафорами, событиями и многими другими.

Каждый объект ядра — это блок памяти, выделенный ядром и доступный только ему. В блоке размещается структура данных, поля которой содержат информацию об объекте. Некоторые поля (например, дескриптор защиты и счетчик количества пользователей) присутствуют во всех объектах, но большая их часть специфична для объектов конкретного типа. Приложение не имеет прямого доступа к структурам данных, представляющим собой объекты ядра. Оперировать объектами ядра приложение может только через специальные функции Windows.

Когда вы вызываете функцию, создающую объект ядра (например, `CreateFile`), она возвращает дескриптор созданного объекта. Этот дескриптор может быть использован любым потоком вашего процесса. Обычно значения дескрипторов объектов ядра действительны только в адресном пространстве процесса, их создавшего. Поэтому все попытки использования такого дескриптора в другом процессе приводят к ошибкам. Исключение составляют объекты ядра, предназначенные для синхронизации или обмена данными между процессами. Более подробно о них будет сказано ниже.

Объекты принадлежат ядру, а не процессу. Другими словами, если процесс создает какой-либо объект ядра, а затем завершает свою работу, то объект ядра может быть и не разрушен. В большинстве случаев объект все же разрушается, но если этим объектом пользуется другой процесс, то ядро не позволит разрушить объект до тех пор, пока второй процесс не откажется от него.

Ядру известно, сколько процессов использует конкретный объект ядра, поскольку в каждом объекте есть *счетчик пользователей*. В момент создания объекта ядра счетчику присваивается единичное значение. Когда к этому объекту обращается другой процесс, значение счетчика увеличивается на единицу. А когда какой-то процесс завершается, счетчики всех используемых им объектов ядра уменьшаются на единицу. Как только счетчик пользователей объекта обнуляется, ядро уничтожает этот объект.

Многие объекты ядра находятся всегда в одном из двух состояний: *свободном состоянии* (*signaled state*) либо *несвободном состоянии* (*nonsignaled state*). Переход из одного состояния в другое осуществляется по правилам, определенным Microsoft для каждого из объектов ядра. Эти состояния могут использоваться так называемыми *wait-функциями* для синхронизации выполнения потоков.

Объекты ядра можно защитить с помощью дескриптора защиты (*security descriptor*). Он содержит информацию о том, кто создал объект и кто имеет права на доступ к нему. Дескрипторы защиты обычно используют при написании серверных приложений. Создавая клиентское приложение, можно просто игнорировать это свойство объектов ядра.

Почти все функции, создающие объекты ядра, принимают в качестве аргумента указатель на структуру `SECURITY_ATTRIBUTES`. В большинстве случаев на месте соответствующего аргумента можно указать значение `NULL`, и тогда объект создается с защитой по умолчанию. В этом случае создатель объекта и любой член группы администраторов получают полный доступ к объекту, а все прочие процессы к объекту не допускаются.

Вне зависимости от того, как был создан объект ядра, после окончания работы с ним его нужно закрыть вызовом функции `CloseHandle`:

```
BOOL CloseHandle(HANDLE hObject);
```

Эта функция сначала проверяет таблицу дескрипторов данного процесса, чтобы убедиться, что процесс имеет доступ к объекту `hObject`. Если доступ разрешен,

то система получает адрес структуры данных объекта `hObject` и уменьшает в ней счетчик количества пользователей. Как только счетчик обнулится, ядро удаляет объект из памяти.

Если же дескриптор неверен, то функция `CloseHandle` возвращает значение `FALSE`, а функция `GetLastError` — код `ERROR_INVALID_HANDLE`.

Перед самым возвратом управления функция `CloseHandle` удаляет соответствующую запись из таблицы дескрипторов. После этого дескриптор `hObject` считается недоступным для данного процесса, и его нельзя более использовать. Но если счетчик пользователей этого объекта не обнулен, то объект остается в памяти. Это означает, что объект используется другими процессами. Когда и остальные процессы завершат свою работу с этим объектом, тоже вызвав функцию `CloseHandle`, он будет разрушен.

Ранее указывалось, что объекты ядра используются только в рамках процесса, их создавшего. Но все же иногда возникает необходимость в совместном использовании объектов ядра *несколькими* процессами, например, в следующих ситуациях:

- объект ядра «проекция файла» позволяет двум процессам, исполняемым на одной машине, совместно использовать одни и те же блоки памяти;
- почтовые ящики и именованные каналы дают возможность программам обмениваться данными с процессами, исполняемыми на других машинах в сети;
- мьютексы, семафоры и события позволяют синхронизировать потоки, исполняемые в разных процессах, чтобы одно приложение могло уведомить другое приложение об окончании той или иной операции.

Win32 API предоставляет три механизма, позволяющие процессам использовать одни и те же объекты ядра: а) наследование дескриптора объекта в дочернем процессе; б) именованные объекты; в) дублирование дескрипторов объектов. Второй из указанных механизмов основан на использовании совпадающих имен для разделяемых объектов. Именно он будет применен в программных примерах, рассматриваемых ниже в разделе «Обмен данными между процессами».

Процессы и потоки

При запуске приложения операционная система Windows создает процесс. *Процесс (process)* — это совокупность ресурсов, необходимых для выполнения программы. Процесс владеет виртуальным адресным пространством, выполняемым кодом, данными, дескрипторами необходимых объектов и иными ресурсами. Однако сам по себе процесс не выполняется. Вместо этого он запускает единственный поток, который часто называют *первичным потоком (primary thread)*. Если процесс имеет только один первичный поток, то, фактически, понятия «процесс» и «поток» совпадают. Первичный поток может создавать другие потоки, те, в свою очередь, новые потоки и т. д.

Поток (thread) — это основная выполняемая единица, для которой операционная система выделяет процессорное время. Каждый поток работает со своим контекстом. *Контекст потока (thread context)* — это структура, содержащая зна-

чения всех регистров процессора. Кроме того, поток имеет доступ ко всем ресурсам своего процесса, включая память, открытые файлы и другие ресурсы.

Обычно приложение содержит только один процесс, поэтому термины «программа» и «процесс» часто используются как синонимы. В то же время любой поток процесса может создать дочерний процесс, выполняющийся одновременно с родительским процессом.

Планирование потоков

Чтобы все потоки работали, операционная система выделяет каждому из них определенное процессорное время. Тем самым создается иллюзия одновременного выполнения потоков. Разумеется, для многопроцессорных систем возможен истинный параллелизм.

Каждый поток может находиться в одном из трех состояний, показанных на рис. 9.1.

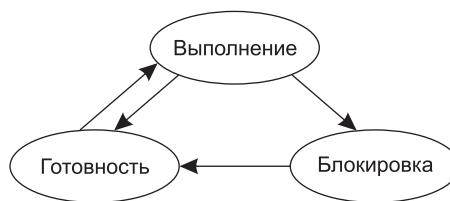


Рис. 9.1. Диаграмма состояний потока

В однопроцессорной системе в любой момент времени только один поток может находиться в состоянии выполнения. Все остальные потоки находятся либо в состоянии готовности, либо в состоянии блокировки.

С самого начала поток попадает в очередь готовых к выполнению потоков. Через какое-то время операционная система выделяет в его распоряжение центральный процессор, и поток переходит в состояние выполнения. В системе Windows реализована система вытесняющего планирования на основе приоритетов. Это означает, что освободившийся процессор продолжает обслуживать тот поток из очереди, который обладает наибольшим приоритетом. О правилах назначения приоритетов мы поговорим чуть позже.

Выбранный для выполнения поток работает в течение некоторого периода, называемого *квантом*. Windows оперирует квантом потока не как отрезком времени, а как целым числом. Обычно поток стартует со значением кванта, равным 6 — для Windows 2000 Professional или 36 — для Windows 2000 Server.

Каждый раз, когда возникает прерывание от системного таймера, из кванта выполняющегося потока вычитается фиксированное значение 3, и так продолжается до тех пор, пока значение кванта не достигнет нуля. Поэтому под управлением Windows 2000 Professional поток будет выполняться в течение двух интервалов системного таймера, а под управлением Windows 2000 Server — в течение 12 интервалов.

Интервал (или период) системного таймера обычно равен 10 мс или около 15 мс, в зависимости от аппаратной платформы. Точное значение этого интервала

можно получить с помощью функции `GetSystemTimeAdjustment` (см. главу 10). Например, для моего компьютера с процессором Intel Celeron CPU 2,0 ГГц период срабатывания системного таймера равен 15,625 мс. При таком периоде кванту потока соответствует временной интервал $15,625 \cdot 2 = 31,25$ мс.

Когда после очередного прерывания квант потока становится равным нулю, Windows переводит поток в состояние готовности и ищет в системе поток с самым высоким приоритетом, находящимся в состоянии готовности. Если в состоянии готовности находятся несколько потоков с приоритетом не ниже предыдущего выполняемого потока, то следующий поток из очереди будет переведен в состояние выполнения и начнет функционировать. Таким образом, потоки с одинаковым уровнем приоритета обслуживаются в циклическом порядке. Впрочем, при отсутствии других претендентов предыдущий поток может получить еще один квант.

Однако выполняемый поток не всегда полностью использует свой квант. Его выполнение может быть прервано при ненулевом кванте в двух ситуациях:

- когда появился в состоянии готовности другой поток с более высоким приоритетом; при этом текущий поток вытесняется и переводится в состояние готовности;
- текущему потоку потребовался какой-либо системный ресурс (или объект ядра), который в настоящий момент времени является занятым; в этом случае поток переводится в состояние блокировки (ожидания события).

Выбрав новый поток, операционная система переключает *контекст*. Эта операция заключается в сохранении содержимого регистров процессора для вытесненного потока и загрузке контекста для выбранного потока.

Классы приоритетов процесса и приоритеты потоков

Windows поддерживает 32 приоритета (от 0 до 31) — чем больше номер, тем выше приоритет. Приоритет потока складывается из двух составляющих: *класса приоритета процесса*, его создавшего, и *относительного приоритета потока* внутри этого класса.

Классы приоритетов процессов приведены в табл. 9.1.

Таблица 9.1. Классы приоритетов

Класс	Флаг в функции <code>CreateProcess</code>	Базовый уровень
Idle	<code>IDLE_PRIORITY_CLASS</code>	4
Below normal	<code>BELOW_NORMAL_PRIORITY_CLASS</code>	6
Normal	<code>NORMAL_PRIORITY_CLASS</code>	8
Above normal	<code>ABOVE_NORMAL_PRIORITY_CLASS</code>	10
High	<code>HIGH_PRIORITY_CLASS</code>	13
Realtime	<code>REALTIME_PRIORITY_CLASS</code>	24

Классы *Below normal* и *Above normal* стали использоваться, начиная с Windows 2000. Класс *Idle* назначается процессу, который должен простоять в случае активности других процессов, например для приложения — хранителя экрана.

Процессам, запускаемым пользователем, присваивается класс *Normal*. Это самые многочисленные процессы в системе. Как правило, они являются интерактивными, то есть требуют постоянного взаимодействия с пользователем, как, например, графические или текстовые редакторы. Процессы класса *Normal* делятся на процессы *переднего плана* (*foreground*) и *фоновые* (*background*). Для процесса, с которым пользователь в данный момент работает, то есть для процесса переднего плана, уровень приоритета поднимается на две единицы. Это повышает комфортабельность общения пользователя с прикладной программой.

Создавать процессы, относящиеся к классу *High*, следует с большой осторожностью. Если поток с классом приоритета *High* занимает процессор достаточно долго, то в это время другие потоки вообще не получат доступа к процессору. Обычно с классом *High* работают некоторые системные процессы, которые большую часть времени ожидают какого-либо события, например, *winlogon.exe*. Если в вашем приложении какая-то подзадача требует быстрой реакции на некоторое событие, то вы можете повышать класс приоритета процесса до значения *High* именно на тот период, когда решается эта подзадача, а затем возвращать его к значению *Normal*. Для изменения класса приоритета процесса во время работы приложения может применяться функция *SetPriorityClass*.

Практически никогда вы не должны использовать класс приоритета *Realtime*, поскольку в этом случае ваше приложение будет прерывать системные потоки, управляющие мышью, клавиатурой и дисковыми операциями. Система будет фактически парализована. Только в особых случаях, когда программа взаимодействует непосредственно с аппаратурой или решаются короткие подзадачи, для которых нужно гарантировать отсутствие прерываний, класс приоритета *Realtime* может быть кратковременно использован.

По умолчанию создаваемый поток получает *базовый приоритет* в соответствии с классом своего процесса. После создания потока его приоритет может изменяться как операционной системой, так и приложением с помощью функции *SetThreadPriority*. В табл. 9.2 приведены относительные приоритеты потоков.

Таблица 9.2. Относительные приоритеты потоков

Относительный приоритет	Флаг в функции <i>SetThreadPriority</i>	Описание
Idle	THREAD_PRIORITY_IDLE	Для процессов класса <i>Realtime</i> приоритет потока равен 16, для процессов остальных классов равен 1
Lowest	THREAD_PRIORITY_LOWEST	Приоритет потока меньше базового приоритета на 2
Below normal	THREAD_PRIORITY_BELOW_NORMAL	Приоритет потока меньше базового приоритета на 1
Normal	THREAD_PRIORITY_NORMAL	Приоритет потока равен базовому приоритету
Above normal	THREAD_PRIORITY_ABOVE_NORMAL	Приоритет потока больше базового приоритета на 1
Highest	THREAD_PRIORITY_HIGHEST	Приоритет потока больше базового приоритета на 2
Time critical	THREAD_PRIORITY_TIME_CRITICAL	Для процессов класса <i>Realtime</i> приоритет потока равен 31, для процессов остальных классов равен 15

Управление процессами

Самый распространенный способ начала процесса — это осуществить запуск приложения в Проводнике (Explorer), либо в меню Пуск (Start), либо набрав название программы в командной строке. Кроме того, Win32 API содержит несколько функций, которые можно использовать для создания и управления процессами.

Использование функции CreateProcess

Функция CreateProcess создает новый процесс и его первичный поток. Она имеет следующий прототип:

```
BOOL CreateProcess(
    LPCTSTR lpApplicationName, // имя исполняемого файла
    LPTSTR lpCommandLine,     // команда строка
    LPSECURITY_ATTRIBUTES processAttributes, // атрибуты доступа к процессу
    LPSECURITY_ATTRIBUTES threadAttributes, // атрибуты доступа к потоку
    BOOL bInheritHandles, // флаг наследования дескрипторов
    DWORD dwCreationFlags, // флаги создания и флаги класса приоритета
    LPVOID lpEnvironment, // указатель на параметры настройки окружения
    LPCTSTR lpCurrentDirectory, // путь к текущему каталогу
    LPSTARTUPINFO lpStartupInfo, // указатель на структуру STARTUPINFO
    LPPROCESS_INFORMATION lpProcessInformation // указатель на структуру
                                                // PROCESS_INFORMATION
);
```

Имя исполняемого файла можно задать в первом или втором параметре.

Параметр lpCommandLine позволяет указать полную командную строку, используемую функцией CreateProcess при создании нового процесса. Разбирая эту строку, функция полагает, что первый компонент в ней представляет собой имя исполняемого файла. Если в этом имени расширение не указано, она считает его .exe. Далее функция приступает к поиску заданного файла и делает это в следующем порядке:

1. Текущий каталог вызывающего процесса.
2. Системный каталог Windows.
3. Основной каталог Windows.
4. Каталоги, перечисленные в переменной окружения PATH.

Конечно, если в имени файла указан полный путь доступа, то система сразу обращается туда и не просматривает эти каталоги.

Чаще всего параметру lpApplicationName передается значение NULL, а имя исполняемого файла содержится в параметре lpCommandLine.

Перед вызовом функции CreateProcess вы должны определить две структурные переменные типа STARTUPINFO и PROCESS_INFORMATION:

```
STARTUPINFO si;
PROCESS_INFORMATION pi;
```

Адреса этих структур передаются в двух последних параметрах функции CreateProcess.

Элементы структуры STARTUPINFO задают атрибуты отображения нового процесса. Надо сказать, что большинство приложений порождает процессы с атрибутами по умолчанию. Но и в этом случае вы должны инициализировать все поля структуры STARTUPINFO хотя бы нулевыми значениями, а в поле cb занести размер этой структуры.

Если запуск нового процесса осуществлен успешно, то функция `CreateProcess` возвращает значение `TRUE` и помимо этого заполняет поля структуры `PROCESS_INFORMATION`, которая определена следующим образом:

```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess; // дескриптор нового процесса
    HANDLE hThread; // дескриптор первичного потока нового процесса
    DWORD dwProcessId; // идентификатор нового процесса
    DWORD dwThreadId; // идентификатор первичного потока нового процесса
} PROCESS_INFORMATION;
```

Описание остальных параметров функции `CreateProcess` можно найти в справочных материалах MSDN. Для этих параметров часто используются значения по умолчанию.

Когда поток в приложении вызывает функцию `CreateProcess`, система создает объект ядра «процесс» с начальным значением счетчика его пользователей, равным единице. Затем система создает для нового процесса виртуальное адресное пространство¹ и загружает в него код и данные как для исполняемого файла, так и для любых DLL, если они используются в работе. Далее система формирует объект ядра «поток» со счетчиком, равным единице, для первичного потока нового процесса.

Первичный поток начинает с исполнения стартового кода из библиотеки C/C++, который в конечном счете вызывает функцию `WinMain`, `wWinMain`, `main` или `wmain`² в запускаемой программе.

Завершение процесса

Процесс можно завершить четырьмя способами:

- ❑ Входная функция первичного потока, например `WinMain`, возвращает управление (предпочтительный способ).
- ❑ Один из потоков процесса вызывает функцию `ExitProcess`.
- ❑ Поток другого процесса вызывает функцию `TerminateProcess` (нежелательный способ).
- ❑ Все потоки процесса завершаются по своей воле. Но это случается очень редко.

Рекомендуется проектировать приложение так, чтобы его процесс завершался только после возврата управления функцией первичного потока. Это единственный способ, который гарантирует корректную очистку всех ресурсов, принадлежащих первичному потоку. При таком завершении любые объекты C++, созданные данным потоком, уничтожаются соответствующими деструкторами. Система освобождает память, которую занимал стек потока, и устанавливает код завершения процесса, который и возвращает входная функция.

Вы можете также завершить процесс, вызвав функцию `ExitProcess`. В справочных материалах MSDN этот способ указан как рекомендуемый. В то же время Дж. Рихтер указывает [5], что возможны ситуации, когда при данном способе

¹ Более подробно о виртуальном адресном пространстве говорится в разделе «Виртуальная память. Адресное пространство процесса».

² С символа «w» начинаются имена Unicode-версий входных функций первичного потока. Функции `main` и `wmain` используются в консольных приложениях.

завершения процесса не для всех объектов C++ будут вызваны деструкторы. Конечно, операционная система и в этом случае корректно очистит все ресурсы, выделенные процессу. Но при этом весьма вероятна утечка памяти или других ресурсов.

Вызов функции `TerminateProcess` тоже завершает процесс. Главное отличие этой функции от `ExitProcess` заключается в том, что ее может вызвать любой поток и завершить при этом любой процесс. Пользуйтесь функцией `TerminateProcess` лишь в крайнем случае, когда иным способом завершить процесс не удается. Процесс не получает абсолютно никаких уведомлений, что он завершается, и приложение не может выполнить очистку ресурсов или предотвратить свое неожиданное завершение. При этом теряются все данные, которые программа не успела переписать из памяти на диск. Но операционная система и в этом случае освобождает все принадлежавшие процессу ресурсы.

Четвертая ситуация может возникнуть, если все потоки вызвали `ExitThread` или они были закрыты другими потоками, вызвавшими функцию `TerminateThread`. Обнаружив, что в процессе нет исполняющихся потоков, операционная система немедленно завершает его. Код завершения процесса приравнивается к коду завершения последнего потока.

В случае завершения процесса системой выполняются следующие действия.

- Выполнение всех потоков в процессе прекращается.
- Все User- и GDI-объекты, созданные процессом, уничтожаются, а объекты ядра закрываются, если их не использует другой процесс.
- Объект ядра «процесс» переходит в свободное, или незанятое (*signaled*), состояние.
- Счетчик пользователей объекта ядра «процесс» уменьшается на единицу.

Запуск обособленных дочерних процессов

В большинстве случаев приложение создает другие процессы как *обособленные* (*detached processes*). Это значит, что после создания и запуска нового процесса родительскому процессу нет нужды взаимодействовать с ним или ждать, пока он закончит работу. Именно так и действует `Explorer`. Это приложение запускает для пользователя новые процессы, а после этого более не следит за их судьбой.

Приведенное ниже приложение `CreateMyProcess` демонстрирует, как можно вызвать из вашей программы стандартное приложение Windows Калькулятор (`calc.exe`).

Создавая приложение `CreateMyProcess`, добавьте к нему ресурс меню с идентификатором `IDR_MENU1`. Меню должно содержать один пункт с именем `Create process` и идентификатором `IDM_CREATE_PROCESS`. Скопируйте также в папку проекта `CreateMyProcess` файлы `KWnd.h` и `KWnd.cpp` из листинга 1.2 и добавьте их в состав проекта.

Листинг 9.1. Проект `CreateMyProcess`

```
//////////  
// CreateMyProcess.cpp  
#include <windows.h>  
#include "resource.h"  
#include "KWnd.h"  
  
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);  
=====
```

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("CreateMyProcess", hInstance, nCmdShow, WndProc,
        MAKEINTRESOURCE(IDR_MENU1), 100, 100, 400, 300);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return (msg.wParam);
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;

    STARTUPINFO si;
    static PROCESS_INFORMATION pi;
    BOOL success;

    switch (uMsg)
    {
    case WM_COMMAND:
        switch (LOWORD(wParam))
        {
        case IDM_CREATE_PROCESS:
            ZeroMemory( &si, sizeof(si) );
            si.cb = sizeof(si);
            success = CreateProcess( NULL, "calc.exe", NULL, NULL, FALSE,
                0, NULL, NULL, &si, &pi);
            if (!success)
                MessageBox(hWnd, "Error of CreateProcess", NULL, MB_OK);
            break;

        default:
            break;
        }

    case WM_PAINT:
        hDC = BeginPaint(hWnd, &ps);
        EndPaint(hWnd, &ps);
        break;

    case WM_DESTROY:
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
        PostQuitMessage(0);
        break;

    default:
        return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }
    return 0;
}
///////////
```

После запуска приложения `CreateMyProcess` пользователь может вызвать приложение Калькулятор, выполнив команду меню `Create process`. После этого оба приложения будут работать независимо друг от друга. Вы можете пользоваться по-переменно и тем и другим приложением, а также в любой последовательности прекращать их выполнение.

Управление потоками

Каждый поток начинает выполнение с некоторой входной функции. В первичном потоке используется одна из функций: `WinMain`, `wWinMain`, `main` или `wmain`. Если вы хотите создать вторичный поток, в нем тоже должна быть входная функция, которая выглядит примерно так:

```
DWORD WINAPI ThreadFunc(PVOID pvParam) {
    DWORD dwResult = 0;
    ...
    return dwResult;
}
```

Функция потока может выполнять любые задачи. Рано или поздно она закончит свою работу и вернет управление. В этот момент поток остановится, память, отведенная под его стек, будет освобождена, а счетчик пользователей его объекта ядра «поток» уменьшится на единицу. Когда счетчик обнулится, этот объект ядра будет разрушен.

В отличие от входной функции первичного потока, имеющей одно из предопределенных имен: `WinMain`, `wWinMain`, `main` или `wmain`, — функции других потоков могут иметь произвольные имена.

Функция `CreateThread`

Для создания потока используется функция `CreateThread`:

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES threadAttributes, // атрибуты доступа
    DWORD dwStackSize,                      // размер стека
    LPTHREAD_START_ROUTINE lpStartAddress, // адрес функции потока
    LPVOID lpParameter,                   // параметр функции потока
    DWORD dwCreationFlags,                // флаги потока
    LPDWORD lpThreadId                  // идентификатор потока
);
```

Как и при работе с функцией `CreateProcess`, для многих параметров можно задавать значения по умолчанию (0 или `NULL`). Третий параметр не может иметь значение по умолчанию, ему всегда передается адрес функции потока. Четвертый параметр часто используется для организации взаимосвязи вызывающего потока с дочерним потоком (листинг 9.2).

При каждом вызове функции `CreateThread` система создает объект ядра «поток» с начальным значением счетчика его пользователей, равным единице. Система выделяет память под стек потока из адресного пространства процесса. Новый поток выполняется в адресном пространстве того же процесса, что и родительский поток.

Завершение потока можно организовать четырьмя способами:

- функция потока возвращает управление (предпочтительный способ);

- ❑ поток самоуничтожается вызовом функции `ExitThread` (рекомендация такая же, как и для функции `ExitProcess`);
- ❑ один из потоков данного или стороннего процесса вызывает функцию `Terminate Thread` (нежелательный способ);
- ❑ завершается процесс, содержащий данный поток (тоже нежелательно).

В случае завершения потока сначала уничтожаются все User-объекты, принадлежащие потоку, а именно окна и ловушки (*hooks*). После этого объект ядра «поток» переходит в свободное состояние, а счетчик пользователей объекта ядра «поток» уменьшается на единицу.

Функция `Sleep`

Поток может попросить у системы не выделять ему процессорное время на определенный период, вызвав функцию `Sleep`:

```
VOID Sleep(DWORD dwMilliseconds);
```

Эта функция приостанавливает выполнение потока на `dwMilliseconds` миллисекунд. При использовании данной функции следует учитывать несколько дополнительных аспектов.

- ❑ Вызывая функцию `Sleep`, поток добровольно отказывается от остатка выделенного ему кванта времени.
- ❑ Система прекращает выделять потоку процессорное время на период, *примерно равный* заданному. Вопрос точности, с которой функция обеспечит затребованную задержку, подробно рассматривается в главе 10.
- ❑ Если параметр `dwMilliseconds` равен нулю, то текущий поток уступает оставшуюся часть своего кванта другому потоку, обладающему равным с ним приоритетом и готовому к выполнению. Однако система может снова запустить текущий поток, если других планируемых потоков с тем же приоритетом нет.

В главе 10 (раздел «Программирование задержек в исполнении кода») приводятся программные эксперименты для уточнения характеристик функции `Sleep`.

Пример многопоточного приложения

Приведенное ниже приложение `CreateMyThreads` демонстрирует создание и параллельную работу трех потоков.

Листинг 9.2. Проект `CreateMyThreads`

```
///////////////////////////////
// CreateMyThreads.cpp
#include <windows.h>
#include <string>
using namespace std;

#include "KWnd.h"

enum UserMsg { UM_THREAD_DONE = WM_USER+1 };

struct ThreadManager {
    ThreadManager(string _name) : name(_name) { nValue = 0; }
    HWND hwndParent;
```

продолжение ↗

Листинг 9.2 (продолжение)

```
string name;
int nValue;
};

ThreadManager tm_A(string("Поток А"));
ThreadManager tm_B(string("Поток В"));
ThreadManager tm_C(string("Поток С"));

DWORD WINAPI ThreadFuncA(LPVOID);
DWORD WINAPI ThreadFuncB(LPVOID);
DWORD WINAPI ThreadFuncC(LPVOID);

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("CreateMyThreads", hInstance, nCmdShow, WndProc,
        NULL, 100, 100, 400, 160);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return (msg.wParam);
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;

    static HANDLE hThreadA, hThreadB, hThreadC;
    static char text[100];
    ThreadManager* pTm;
    static int y = 0;

    switch (uMsg)
    {
    case WM_CREATE:
        tm_A.hwndParent = hWnd;
        hThreadA = CreateThread(NULL, 0, ThreadFuncA, &tm_A, 0, NULL);
        if (!hThreadA)
            MessageBox(hWnd, "Error of create hThreadA", NULL, MB_OK);

        tm_B.hwndParent = hWnd;
        hThreadB = CreateThread(NULL, 0, ThreadFuncB, &tm_B, 0, NULL);
        if (!hThreadB)
            MessageBox(hWnd, "Error of create hThreadB", NULL, MB_OK);

        tm_C.hwndParent = hWnd;
        hThreadC = CreateThread(NULL, 0, ThreadFuncC, &tm_C, 0, NULL);
    }
}
```

```
if (!hThreadC)
    MessageBox(hWnd, "Error of create hThreadC", NULL, MB_OK);
break;

case UM_THREAD_DONE:
    pTm = (ThreadManager*)wParam;
    sprintf(text, "%s: count = %d", pTm->name.c_str(), pTm->nValue);
    y += 30;
    InvalidateRect(hWnd, NULL, FALSE);
break;

case WM_PAINT:
    hdc = BeginPaint(hWnd, &ps);
    TextOut(hdc, 20, y, text, strlen(text));
    EndPaint(hWnd, &ps);
break;

case WM_DESTROY:
    CloseHandle(hThreadA);
    CloseHandle(hThreadB);
    CloseHandle(hThreadC);
    PostQuitMessage(0);
break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}

return 0;
}

//=====
DWORD WINAPI ThreadFuncA(LPVOID lpv)
{
    ThreadManager* pTm = (ThreadManager*)lpv;
    int count = 0;

    for (int i = 0; i < 100000000; ++i) count++;
    pTm->nValue = count;
    SendMessage(pTm->hwndParent, UM_THREAD_DONE, (WPARAM)pTm, 0);

    return 0;
}

//=====
DWORD WINAPI ThreadFuncB(LPVOID lpv)
{
    ThreadManager* pTm = (ThreadManager*)lpv;
    int count = 0;

    for (int i = 0; i < 50000000; ++i) count++;
    pTm->nValue = count;
```

Листинг 9.2 (продолжение)

```
SendMessage(pTm->hwndParent, UM_THREAD_DONE, (WPARAM)pTm, 0);

    return 0;
}

//=====
DWORD WINAPI ThreadFuncC(LPVOID lpv)
{
    ThreadManager* pTm = (ThreadManager*)lpv;
    int count = 0;

    for (int i = 0; i < 20000; ++i) count++;
    pTm->nValue = count;
    SendMessage(pTm->hwndParent, UM_THREAD_DONE, (WPARAM)pTm, 0);

    return 0;
}
///////////
```

В приложении определена структура `ThreadManager`, объекты которой `tm_A`, `tm_B` и `tm_C` используются для взаимосвязи дочерних потоков с первичным потоком. Адреса этих объектов передаются в качестве четвертого параметра при вызовах функции `CreateThread`.

В блоке обработки сообщения `WM_CREATE` создаются три дочерних потока с дескрипторами `hThreadA`, `hThreadB` и `hThreadC`.

Функции потоков `ThreadFuncA`, `ThreadFuncB`, `ThreadFuncC` работают по одному и тому же сценарию. В каждой из них есть локальный счетчик `count`, инкрементируемый в цикле `for`. Разница только в числе повторений цикла: 100 000 000, 50 000 000 и 20 000 раз соответственно. После завершения цикла каждая функция посыпает окну первичного потока пользовательское сообщение `UM_THREAD_DONE`.

Получив сообщение `UM_THREAD_DONE`, функция `WndProc` выводит в свое окно информацию о завершившемся потоке.

На рис. 9.2 показан результат запуска приложения `CreateMyThreads`.

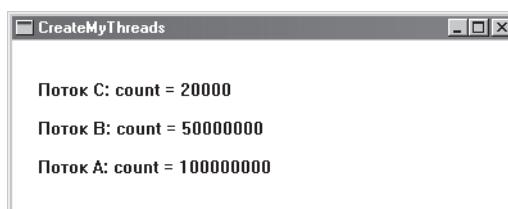


Рис. 9.2. Результат выполнения программы `CreateMyThreads`

Хотя потоки запущены почти одновременно (порядок запуска: поток А, поток В, поток С), быстрее всех завершается поток С, затем поток В и последним — поток А. В этой последовательности информация о них и выводится в главное окно приложения.

В заключение стоит сделать одно замечание об использовании функции `CreateThread`. Дж. Рихтер предостерегает о возможных проблемах в работе приложения, если оно использует функции стандартной библиотеки С/C++ и создает поток вызовом функции `CreateThread`. В этом случае рекомендуется создавать поток

вызовом функции `_beginthreadex` из библиотеки Visual C++. У функции `_beginthreadex` тот же список параметров, что и у `CreateThread`, но их имена и типы несколько различаются.

Чтобы упростить замену вызовов `CreateThread` вызовами `_beginthreadex`, объя-
вите с помощью оператора `typedef` указатель на функцию

```
typedef unsigned (_stdcall *PTHREAD_START) (void*)
```

Теперь для создания потока А в рассмотренном выше приложении вы можете использовать следующую инструкцию:

```
hThreadA = (HANDLE)_beginthreadex(NULL, 0, (PTHREAD_START)ThreadFuncA,  
(void*)&tm_A, 0, NULL);
```

В случае применения функции `_beginthreadex` обязательно свяжите ваш про-
ект с многопоточной версией библиотеки С/C++, иначе будут возникать ошибки
компиляции. Для этого надо выполнить команду меню `Project ▶ Settings` и в по-
явившемся диалоговом окне `Project Settings` перейти на вкладку `C/C++`. В списке
`Category` выберите строку `Code Generation`, а в списке `Use run-time library` — одну из
многопоточных версий библиотеки, например `Debug Multithreaded`.

Взаимодействие потоков через глобальную переменную

Как только вы разобьете приложение на несколько потоков, вам придется решать
проблемы, о существовании которых вы даже не подозревали, занимаясь програм-
мированием в однопоточном режиме.

Например, простая операция считывания и записи в общую глобальную пе-
ременную из нескольких потоков может быть источником ошибок в работе
программы. В качестве примера рассмотрим приложение, приведенное в лис-
tinge 9.3.

Листинг 9.3. Проект BadCount

```
//////////  
// BadCount.cpp  
#include <windows.h>  
#include <stdio.h>  
#include "KWnd.h"  
  
#define N 50000000  
long g_counter = 0;  
  
DWORD WINAPI ThreadFunc(LPVOID);  
void IncCounter();  
void DecCounter();  
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);  
=====  
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,  
LPSTR lpCmdLine, int nCmdShow)  
{  
    MSG msg;  
    KWnd mainWnd("BadCount", hInstance, nCmdShow, WndProc,  
    NULL, 100, 100, 400, 100);  
  
    while (GetMessage(&msg, NULL, 0, 0)) {  
        TranslateMessage(&msg);
```

продолжение ↗

Листинг 9.3 (продолжение)

```
DispatchMessage(&msg);
    }
    return (msg.wParam);
}
//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    HANDLE hThread;
    char text[100];

    switch (uMsg)
    {
        case WM_CREATE:
            hThread = CreateThread(NULL, 0, ThreadFunc, NULL, 0, NULL);
            if (!hThread)
                MessageBox(hWnd, "Error of CreateThread", NULL, MB_OK);

            DecCounter();

            WaitForSingleObject(hThread, INFINITE);
            InvalidateRect(hWnd, NULL, TRUE);
            break;

        case WM_PAINT:
            hDC = BeginPaint(hWnd, &ps);
            sprintf(text, "g_counter = %d", g_counter);
            TextOut(hDC, 20, 20, text, strlen(text));
            EndPaint(hWnd, &ps);
            break;

        case WM_DESTROY:
            PostQuitMessage(0);
            break;

        default:
            return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }

    return 0;
}
//=====
DWORD WINAPI ThreadFunc(LPVOID lpv)
{
    IncCounter();
    return 0;
}
//=====
void IncCounter()
{
    for (int i = 0; i < N; ++i) ++g_counter;
}
//=====
void DecCounter()
{
    for (int i = 0; i < N; ++i) --g_counter;
}
//////////
```

Первичный поток создает с помощью `CreateThread` дочерний поток, имеющий входную функцию `ThreadFunc`. В теле функции `ThreadFunc` вызывается функция `IncCounter`, которая выполняет в цикле N раз операцию инкремента для глобального счетчика `g_counter`.

Запустив дочерний поток и не дожидаясь окончания его выполнения, первичный поток вызывает функцию `DecCounter`. Функция `DecCounter` выполняет в цикле N раз операцию декремента для глобального счетчика `g_counter`.

После возврата из `DecCounter` первичный поток ждет окончания работы дочернего потока с помощью функции `WaitForSingleObject`. Вызов функции

```
WaitForSingleObject(hThread, INFINITE);
```

обеспечивает приостановку выполнения потока на неограниченное время (`INFINITE`), пока не освободится объект ядра `hThread`.

После этого, вызывая функцию `InvalidateRect`, первичный поток заставляет Windows сформировать сообщение `WM_PAINT`. Обрабатывая это сообщение, функция `WndProc` выводит значение переменной `g_counter` в главное окно приложения. Нетрудно догадаться, что правильным результатом выполнения этого приложения должно быть значение `g_counter = 0`.

При маленьких значениях N так и происходит. Но вот для значения 50 000 000 эта программа дает самые разные результаты на разных компьютерах. Эксперименты проводились в основном на компьютерах, имеющих процессор Intel Celeron и Intel Pentium. Заметим, что при тактовой частоте 2 ГГц функция `IncCounter` так же, как и функция `DecCounter`, требует для своего выполнения примерно 150 мс. При такой продолжительности выполнения каждый поток будет прерываться не менее пяти раз, отдавая процессор другому потоку.

На всех компьютерах, имеющих процессор с технологией HyperThreading, был получен ненулевой результат счетчика `g_counter`. Для компьютеров с процессором без поддержки HyperThreading результат был иногда ненулевым, иногда нулевым.

Чем же вызвано искажение результата работы программы? — Прерыванием потока в тот момент, когда процессор не полностью завершил очередную операцию! Рассмотрим это чуть подробнее.

Допустим, что функция `DecCounter` первичного потока приступила к очередному вычитанию единицы из счетчика. Процессор, реализуя эту операцию, скопировал значение глобальной переменной `g_counter` в свой регистр. Предположим, что это значение равно 100 000. Далее процессор успел вычесть единицу, но не успел переписать новое значение 99 999 обратно в глобальную переменную. В этот момент система отбирает процессор у первичного потока, сохранив контекст потока, и передает его на использование дочернему потоку. Предположим, что функция `IncCounter` дочернего потока успела 50 000 раз добавить в счетчик единицу, так что переменная `g_counter` стала равна 150 000. В этот момент квант дочернего потока истек, и *после окончания очередной операции* система возвращает процессор первичному потоку. При этом система восстанавливает контекст первичного потока с *незавершенной операцией*. Завершая эту операцию, процессор переписывает значение 99 999 из своего регистра в глобальную переменную `g_counter`. В описанной ситуации квант работы дочернего потока пошел наスマрку — его результаты утеряны!

Итак, мы показали, что одновременное использование несколькими потоками общей глобальной переменной без должной синхронизации может быть источником ошибок.

Синхронизация

Кроме проблем, связанных с использованием общих глобальных переменных, у потоков могут быть также проблемы доступа к общим системным ресурсам. Потоки должны взаимодействовать друг с другом в следующих ситуациях:

- ❑ совместно используя разделяемый ресурс (чтобы не разрушить его);
- ❑ когда нужно уведомить другие потоки о завершении каких-либо операций.

Примитив синхронизации — это объект, который помогает управлять многопоточным приложением. Основными типами примитивов синхронизации в Windows 2000 являются:

- ❑ атомарные операции API-уровня;
- ❑ критические секции;
- ❑ события;
- ❑ ожидаемые таймеры;
- ❑ семафоры;
- ❑ мьютексы.

Каждый из указанных типов применяется в определенных ситуациях.

Атомарный доступ и семейство Interlocked-функций

Большая часть синхронизации потоков связана с *атомарным доступом* (*atomic access*) — монопольным захватом ресурса обращающимся к нему потоком. Win32 API предоставляет несколько функций для реализации взаимно блокированных операций. Все Interlocked-функции работают корректно только при условии, что их аргументы выровнены по границе двойного слова (DWORD).

Функция `InterlockedIncrement`, имеющая прототип

```
LONG InterlockedIncrement(LPLONG lpAddend);
```

инкрементирует 32-разрядную переменную, адрес которой задается параметром `lpAddend`. Функция возвращает новое значение указанной переменной.

Функция `InterlockedDecrement` определена аналогично функции `InterlockedIncrement`, но она декрементирует 32-разрядную переменную.

Пара функций

```
LONG InterlockedExchange(LPLONG lpTarget, LONG Value);
```

```
PVOID InterlockedExchangePointer(PVOID* ppvTarget, PVOID pvValue);
```

монопольно заменяет текущее значение переменной типа `LONG`, адрес которой передается в первом параметре, значением, передаваемым во втором параметре. В 32-разрядном приложении обе функции работают с 32-разрядными значениями. В 64-разрядной программе первая функция оперирует 32-разрядными значениями, а вторая — 64-разрядными. Обе функции возвращают исходное значение переменной.

Следующая функция добавляет к значению переменной, адрес которой передается в первом параметре, значение, передаваемое во втором параметре:

```
LONG InterlockedExchangeAdd(LPLONG lpAddend, LONG Increment);
```

Еще две функции выполняют операцию сравнения и присваивания по результату сравнения:

```
LONG InterlockedCompareExchange(LPLONG lpDestination, LONG Exchange,  
    LONG Comparand);  
PVOID InterlockedCompareExchangePointer(PVOID* ppvDestination,  
    PVOID pvExchange, PVOID pvComparand);
```

Если значение переменной, адрес которой передается в первом параметре, совпадает со значением, передаваемым в третьем параметре, то оно заменяется значением, передаваемым во втором параметре. В 32-разрядном приложении обе функции работают с 32-разрядными значениями. В 64-разрядной программе первая функция оперирует 32-разрядными значениями, а вторая — 64-разрядными. Обе функции возвращают исходное значение переменной, заданной первым параметром.

Вернемся к нашему приложению `BadCount` (см. листинг 9.3), работающему некорректно из-за одновременного доступа к общей глобальной переменной `g_counter` из разных потоков. Имея на вооружении функции с атомарным доступом, совсем несложно заставить приложение работать правильно. Для этого нужно в реализации функции `IncCounter` заменить инструкцию

```
++g_counter;
```

вызовом следующей функции:

```
InterlockedIncrement(&g_counter);
```

Аналогично, в реализации функции `DecCounter` необходимо операцию

```
--g_counter;
```

заменить следующим вызовом:

```
InterlockedDecrement(&g_counter);
```

Проверьте, как будет работать на вашем компьютере программа `BadCount` после указанных изменений в ее исходном тексте. Модифицированное приложение, по-видимому, достойно и нового имени `GoodCount`!

Критические секции

Критическая секция (critical section) — это небольшой участок кода, который должен использоваться только одним потоком одновременно. Если в одно время несколько потоков попытаются получить доступ к критическому участку, то контроль над ним будет предоставлен только одному из потоков, а все остальные будут переведены в состояние ожидания до тех пор, пока участок не освободится.

Для использования критической секции необходимо определить переменную типа `CRITICAL_SECTION`:

```
CRITICAL_SECTION cs;
```

Поскольку эта переменная должна находиться в области видимости для каждого использующего ее потока, обычно она объявляется как глобальная. Эту переменную следует инициализировать до ее первого применения с помощью функции `InitializeCriticalSection`:

```
InitializeCriticalSection(&cs);
```

Чтобы завладеть критическим участком, поток должен вызвать функцию `EnterCriticalSection`:

```
EnterCriticalSection(&cs);
```

Если критический участок не используется в данный момент другим потоком, он обозначается системой как занятый, и поток немедленно продолжает выполнение. Если критический участок уже используется, то поток блокируется до тех пор, пока участок не будет освобожден.

После вызова `EnterCriticalSection` следуют инструкции, принадлежащие критическому участку.

Конец критического участка обозначается вызовом функции `LeaveCriticalSection`:
`LeaveCriticalSection(&cs);`

Как только поток получает контроль над критическим участком, доступ других потоков к этому участку блокируется. При этом очень важно, чтобы время выполнения критического участка было минимальным. Это позволит добиться наилучших результатов работы приложения.

Если критический участок больше не нужен, используемые им ресурсы освобождаются вызовом функции `DeleteCriticalSection`.

Кроме перечисленных функций Windows NT/2000 предоставляет функцию `TryEnterCriticalSection`, вызывая которую, поток пытается захватить критический участок:

```
BOOL bAcquired = TryEnterCriticalSection(&cs);
if (bAcquired) {
    // Запись в защищенные переменные
    . .
}
else {
    // Контроль над критическим участком недопустим
    . .
}
```

Если критический участок доступен, то поток становится его «владельцем» и осуществляет запись в защищенные переменные. Если участок недоступен, то поток не блокируется, как в случае применения функции `EnterCriticalSection`, а занимается другой работой, за что отвечает ветвь `else`.

Работая с критическими секциями или применяя `Interlocked`-функции, программа не переключается в режим ядра. Поэтому эти виды синхронизации называют *синхронизацией в пользовательском режиме*. Она отличается высокой скоростью реализации. Главным недостатком такой синхронизации является невозможность ее применения для потоков, принадлежащих разным процессам.

Описываемые ниже примитивы синхронизации основаны на использовании объектов ядра. Но сначала мы должны рассмотреть так называемые *wait*-функции.

Wait-функции

Многие объекты ядра могут находиться либо в свободном (*signaled state*), либо в занятом состоянии (*nonsignaled state*). К таким объектам относятся:

- процессы;
- потоки;
- задания;
- файлы;
- консольный ввод;

- ❑ уведомления об изменении файлов;
- ❑ события;
- ❑ ожидаемые таймеры;
- ❑ семафоры;
- ❑ мьютексы.

Wait-функции позволяют потоку в любой момент приостановиться и ждать освобождения какого-либо объекта ядра. Важным свойством функций этого семейства является то, что они не тратят процессорное время, пока ждут освобождения объекта или наступления тайм-аута.

Функция **WaitForSingleObject**

Функция имеет следующий прототип:

```
DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMillisecond);
```

Когда поток вызывает эту функцию, параметр *hHandle* идентифицирует объект ядра, поддерживающий состояние «свободен–занят». Параметр *dwMillisecond* задает тайм-аут (*time out*) — интервал времени, спустя которое функция возвращает управление, даже если объект остается в занятом состоянии. Если параметр *dwMillisecond* имеет нулевое значение, то функция только проверяет состояние объекта и возвращает управление немедленно. Константа *INFINITE* в качестве значения *dwMillisecond* задает бесконечное значение тайм-аута.

Возвращаемым значением функции чаще всего является одна из следующих констант:

Константа	Интерпретация
<i>WAIT_OBJECT_0</i>	Контролируемый объект ядра перешел в свободное состояние
<i>WAIT_TIMEOUT</i>	Истек интервал тайм-аута, а контролируемый объект ядра остается в занятом состоянии
<i>WAIT_FAILED</i>	Функция завершилась с ошибкой. Для получения дополнительной информации об ошибке нужно использовать функцию <i>GetLastError</i>

Вот пример обработки кода возврата функции *WaitForSingleObject*:

```
DWORD dw = WaitForSingleObject(hProcess, 5000);
switch (dw) {
    case WAIT_OBJECT_0:
        // код обработки для завершившегося процесса
        break;

    case WAIT_TIMEOUT:
        // код обработки, если процесс не завершился в течение 5000 мс
        break;

    case WAIT_FAILED:
        // функция завершилась с ошибкой.
        break;
}
```

Будьте осторожны, используя константу *INFINITE* в качестве второго параметра функции *WaitForSingleObject*. Если объект *hHandle* так и не перейдет в свободное состояние, то вызывающий поток никогда не проснеться. Единственное «утешение» заключается в том, что тратить драгоценное процессорное время он при этом не будет.

Функция WaitForMultipleObjects

Функция имеет следующий прототип:

```
DWORD WaitForMultipleObjects(DWORD nCount, CONST HANDLE* lpHandles,  
    BOOL fWaitAll, DWORD dwMilliseconds);
```

Она работает так же, как и функция `WaitForSingleObject`, но при этом позволяет ждать освобождения сразу нескольких объектов или какого-то одного объекта из заданного списка.

Параметр `nCount` определяет количество интересующих вас объектов ядра. Его значение должно быть в пределах от 1 до `MAXIMUM_WAIT_OBJECTS`. В заголовочных файлах Windows эта константа имеет значение 64. Параметр `lpHandles` содержит указатель на массив дескрипторов объектов ядра. В массиве могут содержаться дескрипторы объектов разных типов.

Функция `WaitForMultipleObjects` приостанавливает поток и заставляет его ждать освобождения либо всех заданных объектов ядра, либо одного из них. Параметр `fWaitAll` определяет поведение функции. Значение `TRUE` задает режим ожидания освобождения всех указанных объектов, а `FALSE` — только одного из них. В последнем случае код возврата функции содержит информацию о том, какой именно объект освободился.

Возвращаемое функцией значение сообщает, почему возобновилось выполнение вызвавшего ее потока. Значения `WAIT_TIMEOUT` и `WAIT_FAILED` интерпретируются по аналогии с функцией `WaitForSingleObject`. Если параметр `fWaitAll` равен `TRUE` и все объекты перешли в свободное состояние, то функция возвращает значение `WAIT_OBJECT_0`. Если же `fWaitAll` имеет значение `FALSE`, то функция возвращает управление, как только освобождается любой из объектов. При этом ее код возврата лежит в интервале от `WAIT_OBJECT_0` до `WAIT_OBJECT_0 + nCount - 1`. Иными словами, если из кода возврата вычесть константу `WAIT_OBJECT_0`, то вы получите индекс освободившегося объекта в массиве `lpHandles`.

Вот пример обработки кода возврата функции `WaitForMultipleObjects`:

```
HANDLE hp[3];  
hp[0] = hProcess0;  
hp[1] = hProcess1;  
hp[2] = hProcess2;  
DWORD dw = WaitForMultipleObjects (3, hp, FALSE, 4000);  
switch (dw) {  
    case WAIT_OBJECT_0 + 0:  
        // завершился процесс с дескриптором hp[0]  
        break;  
    case WAIT_OBJECT_0 + 1:  
        // завершился процесс с дескриптором hp[1]  
        break;  
    case WAIT_OBJECT_0 + 2:  
        // завершился процесс с дескриптором hp[2]  
        break;  
  
    case WAIT_TIMEOUT:  
        // ни один из объектов не освободился в течение 4000 мс  
        break;  
    case WAIT_FAILED:  
        // функция завершилась с ошибкой.  
        break;  
}
```

Побочные эффекты успешного ожидания

Успешный вызов функции `WaitForSingleObject` или `WaitForMultipleObjects` на самом деле меняет состояние некоторых объектов ядра. Успешным считается такой вызов, который завершается освобождением соответствующего объекта или объектов. При этом функция возвращает значение `WAIT_OBJECT_0` или значение, являющееся смещением относительно `WAIT_OBJECT_0`. Вызов считается неудачным, если возвращается значение `WAIT_TIMEOUT` или `WAIT_FAILED`. В этом случае состояние каких-либо объектов не меняется.

Пусть, например, поток вызвал функцию `WaitForSingleObject` и ждет освобождения объекта «событие с автосбросом» (объекты-события рассматриваются в следующем разделе). Когда объект переходит в свободное состояние, функция обнаруживает это и возвращает значение `WAIT_OBJECT_0`. Но перед самым возвратом из функции объект-событие будет переведен в занятое состояние. Это и есть побочный эффект успешного ожидания.

Объекты ядра «событие с автосбросом» ведут себя подобным образом, потому что таково одно из правил, определенных Microsoft для объектов этого типа. Другие объекты дают иные побочные эффекты, а некоторые — вообще никаких. К последним относятся объекты ядра «процесс» и «поток».

События

Событие (event) — самая простая разновидность объектов ядра. Оно содержит счетчик количества пользователей и две булевые переменные. Одна переменная указывает тип данного объекта-события, а другая — его состояние.

События просто уведомляют об окончании какой-либо операции. Объекты-события бывают двух типов: со *сбросом вручную* (*manual-reset events*) или с *автосбросом* (*auto-reset events*). Первые события позволяют возобновить выполнение сразу нескольких ждущих потоков, а вторые — только одного потока.

Объект ядра «событие» создается функцией `CreateEvent`, имеющей следующий прототип:

```
HANDLE CreateEvent(
    LPSECURITY_ATTRIBUTES eventAttributes, // атрибуты доступа
    BOOL bManualReset, // тип сброса
    BOOL bInitialState, // начальное состояние
    LPCTSTR pszName // имя объекта
);
```

Параметр `bManualReset` определяет тип объекта-события. Значение `TRUE` создает событие со сбросом вручную, а значение `FALSE` — событие с автосбросом.

Параметр `bInitialState` определяет начальное состояние события — свободное (`TRUE`) или занятое (`FALSE`).

Параметр `pszName` содержит указатель на С-строку, в которой указывается имя объекта. Если `pszName` имеет значение `NULL`, то создается *неименованный объект*.

Например, следующий вызов функции `CreateEvent` из процесса А создает событие с автосбросом и именем `EventName`:

```
HANDLE hEvent = CreateEvent(NULL, FALSE, FALSE, "EventName");
```

На самом деле все обстоит немножко сложнее. При таком вызове система проверяет, не существует ли уже объект ядра с таким именем. Если подобный объект существует, то ядро проверяет тип этого объекта. Допустим, что некий процесс В

уже создал ранее объект-событие с таким же именем `EventName`. В этом случае система проверяет права доступа процесса A к этому объекту. Если с правами доступа все в порядке, то в таблице дескрипторов процесса A создается новая запись с дескриптором `hEvent`. То есть процесс A получает свой дескриптор уже существующего именованного объекта-события, а счетчик пользователей этого объекта увеличивается на единицу.

Предположим, что имеет место другая ситуация, когда некий процесс B уже создал ранее объект ядра с таким же именем, но другого типа, например семафор. Тогда функция `CreateEvent` вернет значение `NULL`, а если после этого вызвать функцию `GetLastError`, то она вернет код ошибки, равный 6 (`ERROR_INVALID_HANDLE`).

Наконец, самая простая ситуация — когда объект ядра с таким именем (`EventName`) в момент вызова функции `CreateEvent` не существует. Тогда действительно будет создан новый объект ядра «событие» с именем `EventName`.

Так что будьте аккуратны с именованием объектов ядра и учитывайте, что *пространство имен* объектов ядра является общим для объектов всех типов.

Объекты-события могут разделяться разными процессами. Допустим, что некий процесс A создал новое событие `hEvent` с именем `EventName`. Потоки из других процессов могут получить доступ к этому объекту несколькими способами:

- вызовом функции `CreateEvent` с передачей в параметре `pszName` такого же имени (эта ситуация только что рассматривалась);
- наследованием дескриптора;
- применением функции `DuplicateHandle`;
- вызовом функции `OpenEvent` с передачей в параметре `pszName` такого же имени.

Функция `OpenEvent` имеет следующий прототип:

```
HANDLE OpenEvent(
    DWORD dwDesiredAccess, // требуемый доступ
    BOOL bInheritHandle,   // опция наследования
    LPCTSTR pszName       // имя объекта
);
```

Параметр `dwDesiredAccess` определяет требуемый доступ к объекту-событию. Его значением может быть любая комбинация следующих флагов:

Флаг доступа	Описание
<code>EVENT_ALL_ACCESS</code>	Все возможные флаги доступа к объекту-событию
<code>EVENT_MODIFY_STATE</code>	Разрешено использование дескриптора объекта в функциях <code>SetEvent</code> и <code>ResetEvent</code>
<code>SYNCHRONIZE</code>	Разрешено использование дескриптора объекта в любых <code>wait</code> -функциях

Функция `OpenEvent` завершится с ошибкой, если атрибуты доступа, заданные для объекта с именем `pszName` при вызове функции `CreateEvent`, не будут разрешать доступ, указанный параметром `dwDesiredAccess`.

Параметр `bInheritHandle` определяет, разрешено ли наследование данного объекта при создании дочерних процессов. Последний параметр `pszName` указывает на имя существующего объекта-события.

Если функция завершается успешно, то возвращаемое значение содержит дескриптор существующего объекта-события. Если возникла ошибка, то функция возвращает значение `NULL`. При помощи функции `GetLastError` можно уточнить причину возникновения ошибки.

Создав событие или получив дескриптор существующего события, вы можете управлять его состоянием с помощью функции

```
BOOL SetEvent(HANDLE hEvent);
```

которая переводит объект `hEvent` в свободное состояние, или с помощью функции

```
BOOL ResetEvent(HANDLE hEvent);
```

которая переводит его в занятое состояние.

Для событий с автосбросом действует следующее правило. Когда ожидание потоком освобождения события успешно завершается, этот объект автоматически сбрасывается в занятое состояние. Для событий со сбросом вручную автоматического сбрасывания не происходит, поэтому для возврата в занятое состояние необходимо вызвать функцию `ResetEvent`.

Пример использования объектов-событий приводится в листингах 9.4 и 9.5.

Семафоры

Объекты ядра «семафор» (*semaphore*) используются для учета ресурсов. Кроме счетчика числа пользователей семафор содержит два 32-битных значения со знаком. Одно из них определяет максимальное количество ресурсов, контролируемых семафором, а другое используется как счетчик текущего числа ресурсов.

Объект ядра «семафор» создается вызовом функции `CreateSemaphore`:

```
HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTES semaphoreAttributes, // атрибуты доступа
    LONG lInitialCount, // текущее количество доступных ресурсов
    LONG lMaximumCount, // максимальное количество ресурсов
    LPCTSTR pszName // имя объекта
);
```

Например, в результате следующего вызова:

```
HANDLE hSem = CreateSemaphore(NULL, 0, 5, "MySemaphore");
```

будет создан именованный объект-семафор с максимальным числом ресурсов, равным пяти, и изначально нулевым количеством доступных ресурсов.

Поток может увеличить значение счетчика текущего числа доступных ресурсов на величину `lReleaseCount`, вызывая функцию `ReleaseSemaphore`:

```
BOOL ReleaseSemaphore(
    HANDLE hSemaphore, // дескриптор семафора
    LONG lReleaseCount, // приращение количества доступных ресурсов
    LPVOID lpPreviousCount // предыдущее значение счетчика ресурсов
);
```

По адресу, указанному в последнем параметре, функция записывает предыдущее значение счетчика ресурсов. Если оно вас не интересует, то просто передайте в параметре `lpPreviousCount` значение `NULL`.

Как обычно, любой процесс может получить свой («процессозависимый») дескриптор существующего объекта «семафор», вызвав функцию `OpenSemaphore`:

```
HANDLE OpenSemaphore (
    DWORD dwDesiredAccess, // требуемый доступ
    BOOL bInheritHandle, // параметр наследования
    LPCTSTR pszName // имя объекта
);
```

Параметр dwDesiredAccess определяет требуемый уровень доступа к объекту-семафору. Его значением может быть любая комбинация следующих флагов:

Флаг доступа	Описание
SEMAPHORE_ALL_ACCESS	Все возможные флаги доступа к объекту-семафору
SEMAPHORE_MODIFY_STATE	Разрешено использование дескриптора объекта в функции ReleaseSemaphore
SYNCHRONIZE	Разрешено использование дескриптора объекта в любых wait-функциях

Если функция завершается успешно, то возвращаемое значение содержит дескриптор существующего объекта-семафора. Если возникла ошибка, функция возвращает значение NULL. При помощи функции GetLastError можно уточнить причину возникновения ошибки.

Поток получает доступ к ресурсу, вызывая одну из wait-функций и передавая ей дескриптор семафора, охраняющего этот ресурс. Wait-функция проверяет у семафора значение счетчика текущего числа ресурсов. Если оно больше нуля (то есть семафор свободен), то значение счетчика уменьшается на единицу, а вызывающий поток продолжает выполнение. Очень важно, что эта операция выполняется для семафора на уровне атомарного доступа. Иначе говоря, пока wait-функция не вернет управление, операционная система не позволит прервать эту операцию никакому другому потоку.

Если wait-функция определяет, что счетчик текущего числа ресурсов равен нулю (семафор занят), то система переводит вызывающий поток в состояние ожидания. Когда другой поток увеличит значение этого счетчика, система вспомнит о ждущем потоке и снова начнет выделять ему процессорное время. А он, захватив ресурс, уменьшит значение счетчика на единицу.

Мьютексы

Объекты ядра «мьютексы» (*mutex*) гарантируют потокам взаимоисключающий доступ к единственному ресурсу. Кроме счетчика пользователей мьютекс содержит счетчик рекурсии и переменную, в которой запоминается идентификатор потока-владельца. Мьютексы ведут себя подобно критическим секциям. Однако если критические секции являются объектами пользовательского режима, то мьютексы — это объекты ядра. Поэтому они позволяют синхронизировать доступ к ресурсу нескольких потоков из разных процессов.

Для использования объекта-мьютекса один из процессов должен сначала создать его вызовом функции CreateMutex:

```
HANDLE CreateMutex (
    LPSECURITY_ATTRIBUTES mutexAttributes, // атрибуты доступа
    BOOL bInitialOwner, // флаг наличия потока-владельца
    LPCTSTR pszName // имя объекта
);
```

Параметр bInitialOwner определяет начальное состояние мьютекса. Если он имеет значение FALSE, то объект-мьютекс не принадлежит ни одному из потоков и поэтому находится в свободном состоянии. При этом его идентификатор потока-владельца и счетчик рекурсии равны нулю. Если же в этом параметре передается значение TRUE, то идентификатор потока-владельца в мьютексе приравнива-

ется идентификатору вызывающего потока, а счетчик рекурсии получает единичное значение. Если идентификатор потока-владельца не равен нулю, мьютекс считается занятым.

Разумеется, любой процесс может получить свой («процессозависимый») дескриптор существующего объекта «мьютекс», вызвав функцию `OpenMutex`:

```
HANDLE OpenMutex(DWORD dwDesiredAccess, BOOL bInheritHandle,
    LPCTSTR pszName);
```

Значением параметра `dwDesiredAccess` может быть любая комбинация флагов:

Флаг доступа	Описание
<code>MUTEX_ALL_ACCESS</code>	Все возможные флаги доступа к объекту-мьютексу
<code>SYNCHRONIZE</code>	Разрешено использование дескриптора объекта в любых wait-функциях, а также при вызове <code>ReleaseMutex</code>

Если функция завершается успешно, то возвращаемое значение содержит дескриптор существующего объекта-мьютекса. В случае возникновения ошибки функция возвращает значение `NULL`.

Для получения доступа к разделяемому ресурсу поток вызывает одну из wait-функций и передает ей дескриптор мьютекса, охраняющего этот ресурс. Wait-функция проверяет у мьютекса идентификатор потока-владельца. Если идентификатор равен нулю, то ресурс свободен и вызывающий поток может продолжить выполнение. В этом случае перед возвратом из wait-функции идентификатор потока-владельца в мьютексе становится равным идентификатору вызывающего потока, а счетчику рекурсии присваивается единичное значение.

Если wait-функция определяет, что идентификатор потока-владельца не равен нулю, то вызывающий поток переходит в состояние ожидания. Система запоминает это и, когда идентификатор потока-владельца обнуляется (а перед этим обнуляется и счетчик рекурсии), записывает в него идентификатор ожидающего потока, а счетчику рекурсии присваивает единичное значение. После этого ожидающий поток вновь становится планируемым. Все проверки и изменения состояния объекта-мьютекса выполняются на уровне атомарного доступа.

Как только разделяемый ресурс перестает быть нужным, поток, владеющий мьютексом, должен его освободить, вызвав функцию `ReleaseMutex`. Функция `ReleaseMutex` уменьшает значение счетчика рекурсии в мьютексе на единицу. Если счетчик рекурсии становится равным нулю, то обнуляется также идентификатор потока-владельца.

Система допускает рекурсивный многократный захват мьютекса одним потоком. Это происходит, если, уже владея мьютексом и не освободив его, поток вызывает wait-функцию с дескриптором этого же мьютекса. В этом случае счетчик рекурсии в мьютексе каждый раз увеличивается на единицу. Для освобождения мьютекса количество вызовов потоком функции `ReleaseMutex` должно совпадать с количеством предыдущих захватов мьютекса.

А что будет, если поток, владеющий мьютексом, завершится, не успев его освободить? В таком случае система считает, что произошел отказ от мьютекса, и автоматически переводит его в свободное состояние. Если этот мьютекс ждут другие потоки, то система выбирает один из них и позволяет ему захватить мьютекс. Тогда wait-функция возвращает потоку значение `WAIT_ABANDONED` вместо `WAIT_OBJECT_0`. Это позволяет понять, что мьютекс освобожден некорректно.

Обычно программы после возврата из wait-функции специально не проверяют возвращаемое значение на `WAIT_ABANDONED`, поскольку такое завершение потоков происходит очень редко. Но на стадии отладки приложения может оказаться полезным обнаружение ситуаций с некорректным освобождением мьютексов.

Обмен данными между процессами

Потоки одного процесса не имеют доступа к адресному пространству другого процесса. Однако существуют специализированные механизмы для передачи данных между процессами. Обмен данными между процессами (*Interprocess Communication, IPC*) имеет долгую историю развития вместе с развитием вычислительной техники. Последним впечатляющим примером технологической эволюции в области обмена информацией является создание всемирной паутины (World Wide Web).

Приведем краткий список механизмов IPC, встроенных в Windows:

- ❑ Буфер обмена (clipboard) — одна из самых примитивных и хорошо известных форм IPC. Основная его задача состоит в поддержке обмена данными между программами по желанию и под контролем пользователя.
- ❑ Библиотеки динамической компоновки. Когда в рамках DLL объявляется переменная, то ее можно сделать разделяемой (shared). Все процессы из DLL, обращающиеся к такой переменной, будут использовать одно и то же место в физической памяти.
- ❑ Сообщение `WM_COPYDATA`, которое применяется для передачи участка памяти другому процессу.
- ❑ Разделяемая память (shared memory) реализуется при помощи объектов ядра «проекция файла». Этот механизм основан на отображении файлов на оперативную память.
- ❑ Протокол динамического обмена данными (Dynamic Data Exchange, DDE), определяющий все основные функции для обмена данными между приложениями. DDE широко использовался до тех пор, пока Microsoft не приняла решение использовать OLE, которую затем переименовали в ActiveX, в качестве основной технологии взаимодействия программ.
- ❑ Удаленный вызов процедур (Remote Procedure Call, RPC), строго говоря, не является механизмом IPC. Это скорее, технологическая оболочка, расширяющая возможности традиционных механизмов IPC. Благодаря RPC сеть становится совершенно прозрачной как для сервера, так и для клиента.
- ❑ ActiveX является универсальной технологией, и одним из ее применений является обмен данными между процессами. Специально для этой цели Microsoft определила стандартный интерфейс `IDataObject`. А для обмена данными по сети используется Distributed Component Object Model (DCOM), которую можно рассматривать как объединение ActiveX и RPC.
- ❑ Каналы (pipes) — мощная технология обмена данными. В общем случае канал можно представить в виде трубы, соединяющей два процесса, через кото-

рую идет непрерывный поток данных. Каналы делятся на анонимные (*anonymous pipes*) и именованные (*named pipes*). Анонимные каналы используются довольно редко. Именованные каналы, которые поддерживаются только в системах WinNT/2000, передают произвольные данные и могут работать через сеть. В последнее время вместо именованных каналов все чаще используют сокеты.

- Сокеты (sockets) — очень важная технология, так как именно она отвечает за обмен данными в сети Интернет. Сокеты также часто используются в крупных локальных сетях. Сокеты можно рассматривать как «разъемы», представляющие собой абстракцию конечных точек коммуникационной линии, которая соединяет два приложения. Windows содержит достаточно мощный API для работы с сокетами.
- Почтовые слоты (mailslots) — это механизм одностороннего IPC. Если приложению известно имя слота, то оно может помещать туда сообщения, а приложение, которое является владельцем этого слота, может их оттуда извлекать и обрабатывать. Основное преимущество этого способа заключается в возможности передавать сообщения по локальной сети сразу нескольким компьютерам за одну операцию, если, конечно, несколько приемников имеют почтовые слоты с одним и тем же именем.
- Microsoft Message Queue (MSMQ) — обеспечивает посылку сообщений между приложениями с помощью очереди сообщений. В отличие от других форм IPC, эта технология позволяет посыпать сообщения процессу, который в данное время недоступен, например, если приложение не запущено, сервер вышел из строя или сетевой канал связи перегружен. Механизм MSMQ ставит сообщение в очередь до тех пор, пока не появится возможность переслать его адресату.

В данной книге будут рассмотрены только два механизма обмена данными между процессами: а) обмен с помощью разделяемой памяти (или объекта ядра «проекция файла»); б) обмен при помощи сообщения WM_COPYDATA.

Но сначала — несколько слов об архитектуре памяти Win32.

Виртуальная память. Адресное пространство процесса

Каждому процессу операционная система выделяет собственное *виртуальное адресное пространство*. В Win32 его размер составляет 4 Гбайт, что определяется разрядностью регистра команд. Соответственно, 32-битный указатель может быть любым числом в интервале от 0x00000000 до 0xFFFFFFFF. Таким образом, адресуется 4 294 967 296 значений, что как раз и перекрывает указанный диапазон памяти.

Верхняя половина этого пространства, то есть адреса от 0x80000000 до 0xFFFFFFFF, резервируется за операционной системой, а нижняя половина почти вся доступна процессу.

Виртуальное адресное пространство процесса доступно всем потокам этого процесса. С другой стороны, потоки одного процесса не имеют доступа к адресному пространству других процессов.

Физическая память и страничный файл

Так как на современных компьютерах оперативная память RAM имеет размеры, по крайней мере, на порядок меньше 4 Гбайт, то система имитирует задекларированную память за счет дискового пространства. При этом на диске создается *страничный файл* (*page file*), который вместе с физической памятью RAM образует виртуальную память, доступную всем процессам. Другое название страничного файла — *файл подкачки* (*swap file*).

Процессор сам управляет отображением виртуальных адресов из машинных команд в эквивалентные физические адреса в ходе выполнения команды. Осуществляя это отображение, процессор оперирует *страницами памяти* размером 4 Кбайта. Этот же размер страниц используется Windows для управления виртуальной памятью.

Теперь посмотрим, что происходит, когда поток пытается получить доступ к блоку данных в адресном пространстве своего процесса. В этом случае возможны три ситуации:

- Данные, к которым обращается поток, находятся в оперативной памяти. Тогда процессор проецирует виртуальный адрес данных на физический, и поток получает доступ к этим данным.
- Данные отсутствуют в оперативной памяти, но размещены где-то в страничном файле. Попытка доступа к данным генерирует прерывание, называемое *Page Fault* (Ошибка страницы). Тогда система начинает искать свободную страницу в оперативной памяти. Если такой страницы нет, то система вынуждена освободить одну из занятых страниц. Если занятая страница не модифицировалась, она просто освобождается. В ином случае она сначала копируется из оперативной памяти в страничный файл. После этого система отыскивает в страничном файле запрошенный блок данных, загружает этот блок на свободную страницу оперативной памяти и, наконец, проецирует виртуальный адрес данных на соответствующий адрес в физической памяти.
- Иногда из-за программной ошибки или сбоя в аппаратной части требуемая страница отсутствует и в оперативной памяти, и в страничном файле. Тогда система генерирует ошибку *Invalid Page Fault*, и работающее приложение закрывается.

Архитектура интерфейсов (API) управления памятью

Диспетчер виртуальной памяти (Virtual Memory Manager — VMM) является составной частью ядра операционной системы. Он отображает виртуальные адреса на физические, используя механизм подкачки страниц памяти (*page swapping*). Кроме того, он предоставляет прикладным программам различные интерфейсы (API) для работы с виртуальной памятью:

- *Virtual Memory API* — набор функций, позволяющих приложению работать с виртуальным адресным пространством. Например, функции *VirtualAlloc* и *VirtualFree* позволяют процессу получать страницы из памяти или возвращать их системе.
- *Memory Mapped File API* — набор функций, позволяющих работать с файлами, проецируемыми в память. Это новый механизм, предоставляемый Windows для работы с файлами и взаимодействия процессов.

- *Heap Memory API* – набор функций, позволяющих работать с динамически распределяемыми областями памяти (кучами).

Файлы данных, проецируемые в память

Проецирование файла данных в адресное пространство процесса предоставляет разработчику мощный механизм работы с файлами. Спроектировав файл на адресное пространство процесса, программа получает возможность работать с ним, как с массивом. Это очень удобно при манипуляциях с большими потоками данных.

Для проецирования файла в память необходимо выполнить три операции:

1. Создать объект ядра «файл», идентифицирующий дисковый файл, который вы хотите использовать как проецируемый в память (функция `CreateFile`).
2. Создать объект ядра «проекция файла» при помощи функции `CreateFileMapping`. При этом используется дескриптор файла, возвращенный функцией `CreateFile`.
3. Указать отображение объекта «проекция файла» или его части на адресное пространство процесса (функция `MapViewOfFile`).

Закончив работу с проецируемым в память файлом, тоже следует выполнить три операции:

1. Отменить отображение на адресное пространство процесса объекта «проекция файла» (функция `UnmapViewOfFile`).
2. Закрыть объект ядра «проекция файла».
3. Закрыть объект ядра «файл».

Описанную технологию можно проиллюстрировать следующим фрагментом кода:

```
HANDLE hFile, hFileMapping;
PVOID pArray;
hFile = CreateFile("File Name", ... );
hFileMapping = CreateFileMapping(hFile, ... );
CloseHandle(hFile) ;
pArray = MapViewOfFile(hFileMapping, ... );
CloseHandle(hFileMapping) ;
/*
 * Работаем с файлом, как с массивом pArray */
UnmapViewOfFile(pArray);
```

В этом примере «закрывающие» операции выполняются сразу после использования соответствующего дескриптора объекта. Это уменьшает вероятность утечки ресурсов.

Использование проекции файла для реализации разделяемой памяти

Самый низкоуровневый механизм совместного использования данных в одной системе – проецирование файла в память. На нем так или иначе базируются все другие механизмы разделения данных. Поэтому, если вы хотите получить максимальное быстродействие с минимумом издержек, лучше всего применять именно проецирование.

Совместное использование данных в этом случае происходит по следующей схеме. Несколько процессов проецируют в память представления одного и того же объекта «проекция файла», то есть делят одни и те же страницы физической памяти. Поэтому, когда один процесс записывает данные в представление общего объекта «проекция файла», эти изменения немедленно отражаются в других процессах. Но для этого все процессы должны использовать одинаковое имя объекта «проекция файла».

В предыдущем разделе рассматривалось проецирование представления файла, размещенного на диске. Но для целей обмена данными между разными процессами хранение этих данных на диске было бы очень неудобным. К счастью, Win32 API предусматривает возможность проецирования файлов непосредственно на физическую память из страничного файла, а не из специально создаваемого дискового файла.

Этот способ даже проще стандартного, рассмотренного в предыдущем разделе. Во-первых, не нужно вызывать функцию `CreateFile`. Вы просто вызываете функцию `CreateFileMapping` и передаете значение `INVALID_HANDLE_VALUE` (или константу `-1`) в параметре `hFile`. Но при вызове функции `CreateFileMapping` следует передать в последнем ее параметре С-строку, содержащую имя этого объекта. Тогда другие процессы, если им понадобится доступ к разделяемой памяти, смогут вызвать функцию `OpenFileMapping` и передать ей то же самое имя.

Учтите, что если разделяемая память используется в режиме записи данных более чем одним потоком, то вы должны предусмотреть синхронизацию работы этих потоков. Пример такой синхронизации при помощи объектов-событий рассматривается чуть ниже.

Когда работа с объектом «проекция файла» завершена, процесс должен вызвать функцию `CloseHandle`. Как только все дескрипторы объекта будут закрыты, система освободит память, переданную из страничного файла.

Модель «клиент-сервер»

Чтобы показать применение механизмов обмена между процессами с помощью разделяемой памяти (файла, проецируемого в память) и с помощью сообщения `WM_COPYDATA`, мы разработаем две программы, имитирующую функции сервера и клиента.

Клиентом называется объект, запрашивающий доступ к службе или ресурсу. *Сервер* — это объект, выполняющий некую службу или обладающий ресурсом.

Клиент и сервер могут работать на одной и той же машине, используя локальные механизмы коммуникации, или на разных машинах, применяя для связи сетевые средства.

Поведение клиента и сервера асимметрично. Процесс-сервер инициализируется и переходит в состояние ожидания запросов от возможных клиентов. Как правило, процесс-клиент запускается в интерактивном режиме и посыпает запросы серверу. Сервер исполняет полученный запрос, причем это может подразумевать диалог с клиентом, а может — и нет. Затем сервер вновь переходит в состояние ожидания запросов от других клиентов.

В рассматриваемом ниже примере клиентом и сервером являются приложения `ClientApp` и `ServerApp`. Рисунок 9.3 показывает, как осуществляется взаимосвязь между этими приложениями через локальные механизмы коммуникации.

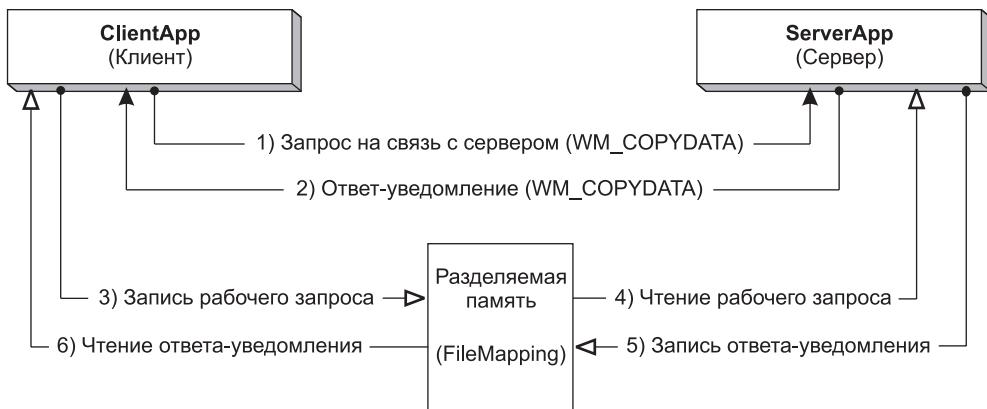


Рис. 9.3. Взаимодействия клиента и сервера

Хотя на рисунке изображен только один клиент, на самом деле их может быть несколько. Максимально возможное число клиентов определяется «мощностью» или пропускной способностью сервера.

Сценарий взаимодействия клиента и сервера предусматривает начальную фазу, в которой клиент посылает серверу запрос на связь и получает ответ-уведомление как признак установки связи. После этого начинается рабочая фаза, в которой с некоторой периодичностью клиент записывает свой рабочий запрос в разделяемую память и ждет ответа от сервера. Этот ответ тоже записывается в разделяемую память. Синхронизация взаимодействия клиента и сервера осуществляется через объекты ядра «события». Таким образом, в рабочей фазе повторяются действия, помеченные на рис. 9.3 номерами 3—6.

Обмен данными с помощью сообщения WM_COPYDATA

Системное сообщение WM_COPYDATA является, вероятно, самым простым методом обмена данными между процессами.

Чтобы послать любое сообщение при помощи функции `SendMessage`, необходимо знать дескриптор окна, которому оно посыпается. Обычно этот дескриптор получают вызовом функции `FindWindow`, например:

```
hwndServer = FindWindow(NULL, "ServerApp");
```

С помощью этой инструкции клиентское приложение узнает дескриптор основного окна серверного приложения, имеющего заголовок `ServerApp`.

Прежде чем посыпать сообщение WM_COPYDATA, вы должны определить структурную переменную типа `COPYDATASTRUCT`. Эта структура объявлена следующим образом:

```
typedef struct tagCOPYDATASTRUCT {
    ULONG_PTR dwData;
    DWORD cbData;
    PVOID lpData;
} COPYDATASTRUCT, *PCOPYDATASTRUCT;
```

В поле `dwData` можно записать 32-битное число, которое будет передано приложению-получателю.

Поле `lpData` может содержать указатель на данные, передаваемые приложению-получателю. Если `lpData` не равно `NULL`, то поле `cbData` должно содержать размер в байтах данных, на которые указывает `lpData`.

В зависимости от потребностей приложения вы можете использовать оба указанных поля (`dwData` и `lpData`) или только одно из них.

Когда вы посыпаете сообщение `WM_COPYDATA` с помощью функции `SendMessage`, адрес переменной типа `COPYDATASTRUCT` должен передаваться в параметре `lParam`.

Например, следующие инструкции в клиентском приложении подготавливают запрос в виде С-строки `request` и посыпают его серверному приложению, которое имеет заголовок окна `ServerApp`:

```
COPYDATASTRUCT cds;
char request[80];
sprintf(request, "ClientApp");
cds.lpData = &request;
cds.cbData = strlen(request);
hwndServer = FindWindow(NULL, "ServerApp");
SendMessage(hwndServer, WM_COPYDATA, (WPARAM)hWnd, (LPARAM)&cds);
```

Программа, которой адресуется сообщение `WM_COPYDATA`, должна рассматривать данные, на которые указывает `lpData`, как данные только для чтения. Указатель `lpData` является корректным только в процессе обработки сообщения `WM_COPYDATA`. Если принимающее приложение намерено использовать полученные данные в дальнейшей своей работе, то оно должно скопировать их в локальный буфер.

Данные, размещенные по адресу `lpData`, не должны содержать никаких указателей, поскольку в адресном пространстве принимающего приложения эти указатели будут интерпретироваться неверно.

Но пора от теории перейти к практике и показать, наконец, нечто работающее.

Приложение ServerApp

Исходный код приложения `ServerApp` приведен в листинге 9.4.

Листинг 9.4. Проект ServerApp

```
///////////
// ServerApp.cpp
#include <windows.h>
#include <stdio.h>
#include "KWhd.h"

#define MAX_STR_SIZE 80
#define MAX_N 10 // максимальное количество клиентов
int N = 0; // текущее количество клиентов

HANDLE hEvtRecToServ[MAX_N]; // массив событий "рабочий запрос от клиента"
char eventName[MAX_N][MAX_STR_SIZE + 8]; // массив имен событий

HANDLE hEvtServIsExist; // событие "Сервер уже запущен"
HANDLE hEvtServIsFree; // событие "Сервер свободен"
HANDLE hEvtServIsDone; // событие "Сервер выполнил рабочий запрос"
HANDLE hFileMap = NULL; // объект "проекция файла"

struct ThreadManager {
    HWND hWndParent;
};
```

```

ThreadManager tm; // структура для связи с потоком

DWORD WINAPI ThreadFunc(LPVOID);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("ServerApp", hInstance, nCmdShow, WndProc,
        NULL, 100, 100, 450, 100);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return (msg.wParam);
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;

    COPYDATASTRUCT* pCds;
    COPYDATASTRUCT cds;
    static char request[MAX_STR_SIZE];
    char suffix[8];

    HWND hwndClient;
    static HANDLE hThread;
    HWND* pHwnd = &hWnd;

    int i;

    switch (uMsg)
    {
    case WM_CREATE:
        hEvtServIsExist = OpenEvent(EVENT_ALL_ACCESS, FALSE, "Server");
        if (hEvtServIsExist) {
            MessageBox(hWnd, "Сервер уже выполняется. 2 экземпляр запрещен.",
                "ServerApp", MB_OK | MB_ICONSTOP | MB_SYSTEMMODAL);
            PostQuitMessage(0);
        }
        else
            hEvtServIsExist = CreateEvent(NULL, FALSE, FALSE, "Server");
        break;

    case WM_COPYDATA: // прием запроса от клиента на связь с сервером

        if (N == MAX_N) {
            MessageBox(hWnd, "Сервер перегружен. В доступе отказано",
                "ServerApp", MB_OK | MB_ICONSTOP | MB_SYSTEMMODAL);
            break;
        }
        // Извлечение сообщения
        pCds = (COPYDATASTRUCT*)lParam;
    }
}

```

продолжение ↗

Листинг 9.4 (продолжение)

```
strncpy(request, (char*)pCds->lpData, pCds->cbData);
strcpy(eventName[N], request);
sprintf(suffix, "_%d", N);
strcat(eventName[N], suffix);

// Создание события hEvtRecToServ[N] - "запрос N-го клиента"
hEvtRecToServ[N] = CreateEvent(NULL, FALSE, FALSE, eventName[N]);

if (!N) {
    // Инициализация массива hEvtRecToServ
    for (i = 0; i < MAX_N; ++i)
        hEvtRecToServ[i] = hEvtRecToServ[0];

    // Создание файла, проецируемого в память
    hFileMap = CreateFileMapping(INVALID_HANDLE_VALUE, NULL,
        PAGE_READWRITE, 0, 4 * 1024, "SharedData");
    // Создание событий "Сервер свободен" и "Сервер выполнил рабочий запрос"
    hEvtServIsFree = CreateEvent(NULL, FALSE, TRUE, "ServerIsFree");
    hEvtServIsDone = CreateEvent(NULL, FALSE, FALSE, "ServerIsDone");
    // Запуск потока для обработки запросов клиентов
    tm.hwndParent = hWnd;
    hThread = CreateThread(NULL, 0, ThreadFunc, &tm, 0, NULL);
}

// Отправка обратного сообщения клиенту
hwndClient = FindWindow(NULL, request);
cds.dwData = N;
cds.lpData = &eventName[N];
cds.cbData = strlen(eventName[N]);
SendMessage(hwndClient, WM_COPYDATA, (WPARAM)hWnd, (LPARAM)&cds);

if (N < MAX_N) N++;
InvalidateRect(hWnd, NULL, TRUE);
break;

case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);
    if (N) {
        char text[40];
        sprintf(text, "Количество обслуживаемых клиентов: %d", N);
        TextOut(hDC, 10, 20, text, strlen(text));
    }
    EndPaint(hWnd, &ps);
    break;

case WM_DESTROY:
    CloseHandle(hFileMap);
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
//=====
DWORD WINAPI ThreadFunc(LPVOID lpv)
{
```

```

ThreadManager* pTm = (ThreadManager*)lpv;
HWND hParent = pTm->hwndParent;
static PVOID pView;
char suffix[40];
int k;
char text[100];
DWORD dw;

while (1) {
    // Ожидание "рабочего запроса" от клиента
    dw = WaitForMultipleObjects(MAX_N, hEvtRecToServ, FALSE, INFINITE);
    switch (dw) {
    case WAIT_FAILED:
        MessageBox(hParent, "Ошибка вызова WaitForMultipleObjects",
                   "ServerApp", MB_OK);
        return 0;

    default:
        k = dw - WAIT_OBJECT_0; // индекс клиента
        // Отображаем проекцию файла на адресное пространство процесса
        pView = MapViewOfFile(hFileMap, FILE_MAP_WRITE, 0, 0, 0);
        // Извлекаем содержание запроса в буфер text
        strcpy(text, (PTSTR)pView);
        // Добавляем к нему "суффикс", содержащий имя клиента
        sprintf(suffix, " - %s\\0", eventName[k]);
        strcat(text, suffix);
        // Помещаем сформированную запись обратно в проекцию файла
        strcpy((PTSTR)pView, text);
        UnmapViewOfFile(pView);
        // Освобождаем событие hEvtServIsDone
        SetEvent(hEvtServIsDone);
        break;
    }
}
return 0;
}
/////////////////////////////////////////////////////////////////

```

После старта сервер ждет первого запроса на связь от возможного клиента. Этот запрос поступает в виде сообщения **WM_COPYDATA**. В коде программы **ClientApp**, который приводится ниже, можно увидеть, что клиент посыпает в этом сообщении С-строку, содержащую его имя (которое совпадает с заголовком главного окна приложения). Сервер извлекает эту строку **request** и добавляет к ней суффикс, содержащий символ подчеркивания и номер запроса N. Модифицированная строка запоминается в элементе массива **eventName[N]**. Затем сервер создает объект-событие с дескриптором **hEvtRecToServ[N]** и именем **eventName[N]**.

Если поступивший запрос является первым и N имеет нулевое значение, то сервер выполняет ряд инициализирующих действий:

- ❑ Все элементы массива **hEvtRecToServ** заполняются значениями дескриптора **hEvtRecToServ[0]**. Если этого не сделать, то функция **WaitForMultipleObjects**, вызываемая позже для массива событий **hEvtRecToServ**, работать не будет.
- ❑ Создается объект «проекция файла» с дескриптором **hFileMap** и именем **SharedData**. При этом размер файла, задаваемый четвертым и пятым параметрами функции **CreateFileMapping**, равен 4 Кбайт. Эта величина должна быть кратна размеру

страницы памяти. А третий параметр определяет, что проекция файла будет использоваться и для записи, и для чтения.

- ❑ Создаются объекты-события `hEvtServIsFree` и `hEvtServIsDone`.
- ❑ Запускается вторичный поток с входной функцией `ThreadFunc`. Этот поток предназначен для обработки «рабочих запросов» клиентов.

Завершая обработку принятого сообщения `WM_COPYDATA`, сервер отправляет имя `eventName[N]` обратно клиенту, используя для этого все то же сообщение `WM_COPYDATA`. Это сообщение является ответом-уведомлением для клиента (см. рис. 9.3).

Алгоритм функции потока `ThreadFunc` очень прост. Как только сервер обнаруживает освобождение какого-либо события из массива `hEvtRecToServ`, он определяет индекс клиента и приступает к обработке «рабочего запроса». Обратите внимание на то, что после записи «ответа» в разделяемую память поток вызывает функцию `SetEvent`, чтобы перевести событие `hEvtServIsDone` в свободное состояние. Как мы увидим ниже, клиентское приложение будет ждать этого события, чтобы извлечь ответ сервера из разделяемой памяти.

Приложение ClientApp

Исходный код приложения `ClientApp` приведен в листинге 9.5.

Создав приложение, добавьте к нему ресурс меню с идентификатором `IDR_MENU1`. Меню должно содержать один пункт с именем `Link to server` и идентификатором `IDM_LINK`.

Листинг 9.5. Проект ClientApp

```
////////////////////////////////////////////////////////////////////////
// ClientApp.cpp
#include <windows.h>
#include <stdio.h>
#include "resource.h"
#include "KWnd.h"

#define MAX_LEN_REQUEST 80
#define MAX_LEN_EVTNAME 88
#define TIMER_ID 1
#define TIMER_PERIOD 1000

HANDLE hEvtRecToServ; // событие "Запрос к серверу от клиента"
HANDLE hEvtServIsFree; // событие "Сервер свободен"
HANDLE hEvtServIsDone; // событие "Сервер выполнил запрос"

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====================================================
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("ClientApp", hInstance, nCmdShow, WndProc,
        MAKEINTRESOURCE(IDR_MENU1), 100, 100, 450, 150);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

```
    return (msg.wParam);
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;

    COPYDATASTRUCT* pCds;
    COPYDATASTRUCT cds;
    static char request[MAX_LEN_REQUEST];
    static char eventName[MAX_LEN_EVTNAME];
    static char text[100];
    static char msgSended[256];
    static char msgReceived[256];

    HWND hWndServer;
    static BOOL isLinkToServer;
    static BOOL bServerIsDone = FALSE;

    static HANDLE hFileMap; // объект "проекция файла"
    static PVOID pView;
    DWORD dw0, dw1;
    static int count = 0;

    switch (uMsg)
    {
        case WM_CREATE:
            isLinkToServer = FALSE;
            msgSended[0] = 0;
            msgReceived[0] = 0;
            break;

        case WM_COMMAND:
            switch (LOWORD(wParam))
            {
                case IDM_LINK: // Отправка запроса на связь с сервером
                    sprintf(request, "ClientApp");
                    cds.lpData = &request;
                    cds.cbData = strlen(request);

                    hWndServer = FindWindow(NULL, "ServerApp");
                    if (!hWndServer)
                        MessageBox(hWnd, "Ошибка связи!", "ClientApp", MB_OK);
                    break;
                case WM_COPYDATA:
                    SendMessage(hWndServer, WM_COPYDATA, (WPARAM)hWnd,
                               (LPARAM)&cds);
                    break;
            }
            break;

        case WM_COPYDATA: // Получение ответа от сервера
            pCds = (COPYDATASTRUCT*)lParam;
            // Имя разделяемого объекта-события
            strncpy(eventName, (char*)pCds->lpData, pCds->cbData);
    }
}
```

продолжение ↗

Листинг 9.5 (продолжение)

```
// Дескрипторы разделяемых объектов-событий
hEvtRecToServ = OpenEvent(EVENT_ALL_ACCESS, FALSE, eventName);
hEvtServIsFree = OpenEvent(EVENT_ALL_ACCESS, FALSE, "ServerIsFree");
hEvtServIsDone = OpenEvent(EVENT_ALL_ACCESS, FALSE, "ServerIsDone");

// Открыть файл, проецируемый в память
hFileMap = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE,
    "SharedData");
SetWindowText(hWnd, eventName);
isLinkToServer = TRUE;
InvalidateRect(hWnd, NULL, TRUE);

SetTimer(hWnd, TIMER_ID, TIMER_PERIOD, NULL);
break;

case WM_TIMER:
// Отправка "рабочих запросов" и ожидание ответа от сервера

// Ожидание освобождения сервера
dw0 = WaitForSingleObject(hEvtServIsFree, TIMER_PERIOD / 2);
switch (dw0) {
case WAIT_OBJECT_0:
    // Отображаем проекцию файла на адресное пространство процесса
    pView = MapViewOfFile(hFileMap, FILE_MAP_WRITE, 0, 0, 0);
    // Записываем содержание "рабочего запроса"
    sprintf((PTSTR)pView, "%d\0", ++count);
    // Сообщение для вывода в окно приложения
    sprintf(msgSended, "Запрос к серверу: \t\t%s", (PTSTR)pView);
    UnmapViewOfFile(pView);

    InvalidateRect(hWnd, NULL, FALSE);
    SetEvent(hEvtRecToServ); // освобождаем событие hEvtRecToServ

    // Ожидание события "Сервер выполнил запрос"
    dw1 = WaitForSingleObject(hEvtServIsDone, TIMER_PERIOD / 2);
    switch (dw1) {
    case WAIT_OBJECT_0:
        // Опять отображаем проекцию файла
        pView = MapViewOfFile(hFileMap, FILE_MAP_READ, 0, 0, 0);
        // Извлекаем содержание ответа сервера
        sprintf(msgReceived, "Ответ от сервера: \t\t%s",
            (PTSTR)pView);
        UnmapViewOfFile(pView);
        bServerIsDone = TRUE;
        // освобождаем событие "Сервер свободен"
        SetEvent(hEvtServIsFree);
        InvalidateRect(hWnd, NULL, FALSE);
        break;
    case WAIT_TIMEOUT: return 0;
    case WAIT_FAILED:
        MessageBox(hWnd, "Ошибка ожидания hEvtServIsDone",
            "ClientApp", MB_OK);
        return 0;
    }
}

case WAIT_TIMEOUT: return 0;
```

```

case WAIT_FAILED:
    MessageBox(hWnd, "Ошибка ожидания hEvtServIsFree",
               "ClientApp", MB_OK);
    return 0;
}

break;

case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);

    if (isLinkToServer) {
        sprintf(text, "Установлена связь с сервером через событие %s",
               eventName);
        TextOut(hDC, 20, 20, text, strlen(text));
        TabbedTextOut(hDC, 20, 40, msgSended, strlen(msgSended), 0,
                      NULL, 20);
    }
    if (bServerIsDone) {
        bServerIsDone = FALSE;
        TabbedTextOut(hDC, 20, 60, msgReceived, strlen(msgReceived), 0,
                      NULL, 20);
    }
    EndPaint(hWnd, &ps);
    break;

case WM_DESTROY:
    UnmapViewOfFile(pView);
    CloseHandle(hFileMap);
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
/////////////////////////////////////////////////////////////////

```

Когда приложение запущено, работа клиента начинается после того, как пользователь выполнит команду меню **Link to server**. Обрабатывая эту команду (**case IDM_LINK**), программа посыпает серверу сообщение **WM_COPYDATA**, содержащее указатель на строку **request**.

Получив ответ-уведомление от сервера также через сообщение **WM_COPYDATA**, клиент выполняет следующие действия:

- ❑ извлекает имя разделяемого объекта-события **eventName**;
- ❑ открывает разделяемые объекты-события, получая дескрипторы **hEvtRecToServ**, **hEvtServIsFree**, **hEvtServIsDone** (событие **hEvtRecToServ** в дальнейшем клиент использует, чтобы информировать сервер о готовности «рабочего запроса»);
- ❑ открывает существующий объект «проекция файла» с именем **SharedData** (сервер должен быть запущен раньше клиента);
- ❑ заменяет текст заголовка своего окна строкой **eventName**;
- ❑ запускает стандартный таймер с периодом **TIMER_PERIOD**.

Далее работа программы управляется прерываниями от таймера. Обрабатывая сообщения WM_TIMER, приложение проверяет с помощью функции WaitForSingleObject, свободен ли сервер. Если да, то в разделяемую память записывается «рабочий запрос», представляющий собой С-строку с номером запроса. Если нет, то приложение ждет освобождения сервера.

После успешной записи «рабочего запроса» в разделяемую память программа переводит в свободное состояние объект-событие hEvtRecToServ. Именно это событие ожидает сервер. Затем клиент переходит к ожиданию события hEvtServIsDone, которое свидетельствует о том, что сервер выполнил запрос. Как только указанное событие освобождается, полученный от сервера ответ извлекается из разделяемой памяти и выводится в окно приложения.

На рис. 9.4 показаны в работе сервер ServerApp и три клиента ClientApp.

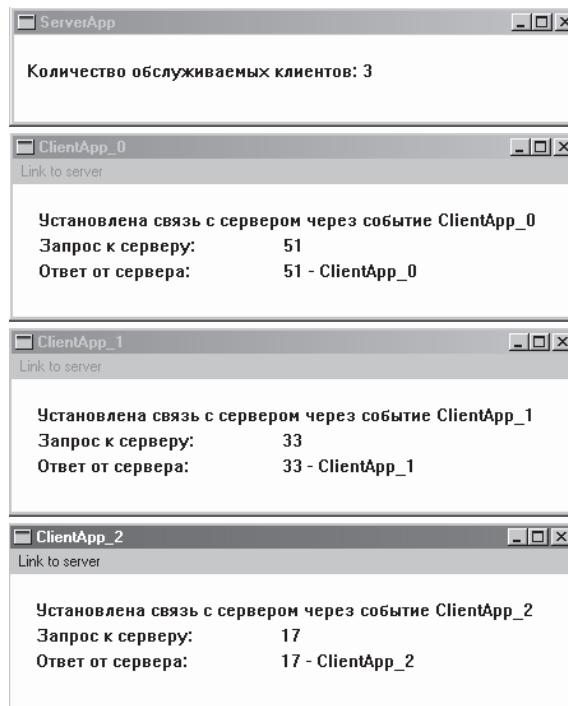


Рис. 9.4. Сервер ServerApp обслуживает трех клиентов ClientApp

Заметим, что с целью сокращения объема текста в приведенных программах опущены те проверки кодов возврата, которые должны производиться после вызова функций Create... и Open.... В реальных приложениях вы, конечно, должны предусмотреть подобные проверки.

Не забывайте освобождать ресурсы

В рассмотренном выше примере сервер обрабатывает запросы клиентов по очень упрощенной технологии. На практике могут применяться более сложные методы обслуживания клиентов. Пусть, например, ваше приложение спроектировано в рас-

чете на предоставление сервисных услуг N клиентам, а реализация каждой услуги требует создания отдельного процесса. По всей видимости, вы объявили массивы структур `STARTUPINFO` и `PROCESS_INFORMATION`:

```
# define N 1000
STARTUPINFO si[N];
PROCESS_INFORMATION pi[N];
int iProc = -1; // индекс обслуживающего процесса
```

Теперь, получив запрос от клиента, сервер может создать отдельный процесс для его обработки:

```
iProc++;
ZeroMemory( &pi[iProc], sizeof(pi[iProc]) );
ZeroMemory( &si[iProc], sizeof(si[iProc]) );
si[iProc].cb = sizeof(si);
CreateProcess( NULL, "ServProcess.exe", NULL, NULL, FALSE,
0, NULL, NULL, &si[iProc], &pi[iProc]);
```

Здесь `iProc` — текущий индекс для создаваемого процесса. Мы опускаем подробности, связанные с управлением этим индексом. Например, при достижении максимально допустимого значения $N-1$ с ним надо что-то делать (в простейшем алгоритме следующим значением `iProc` должен быть 0).

Но в дальнейшем сервер должен проверять состояние запущенных ранее процессов и в случае их завершения сразу же освобождать задействованные ресурсы. Такую проверку можно осуществить, например, с помощью следующего кода, который вызывается по прерыванию от таймера:

```
DWORD dwExitCode;
for (i = 0; i < N; ++i) {
    if (pi[i].hProcess) {
        GetExitCodeProcess (pi[i].hProcess, &dwExitCode);
        if (dwExitCode != STILL_ACTIVE) {
            CloseHandle(pi[i].hThread);
            CloseHandle(pi[i].hProcess);
            pi[i].hProcess = 0;
        }
    }
}
```

Обработка ошибок здесь опущена. Вызов функции `GetExitCodeProcess` позволяет определить, завершился ли процесс с дескриптором `pi[i].hProcess`? Если да, то корректное освобождение ресурсов обеспечивается вызовами функции `CloseHandle` для дескриптора основного потока и дескриптора завершившегося процесса.

Когда многопоточность реально полезна?

«Потоки — вещь невероятно полезная, когда ими пользуются с умом» — это цитата из Джеки Рихтера [5]. В то же время бездумное применение многопоточности в любой программе может привести к получению совершенно неожиданных результатов. Например, вы решили повысить производительность приложения за счет распараллеливания вычислений и сделали для этого несколько потоков. Возможно, что на многопроцессорной машине программа действительно станет работать быстрее, но на обычном однопроцессорном персональном компьютере

ее характеристики ухудшатся, так как система будет тратить процессорное время на переключение между потоками.

Приведем несколько примеров известных приложений, работающих в много-поточном режиме:

- ❑ Текстовые процессоры принимают ввод от пользователя, проверяют его на орфографические ошибки и печатают в фоновом режиме.
- ❑ Windows Explorer может копировать файлы из одной папки в другую и одновременно предоставляет пользователю другие функции. Например, можно просматривать содержимое других папок или запускать любое другое приложение.
- ❑ Веб-браузеры способны взаимодействовать с серверами в фоновом режиме. Благодаря этому пользователь может перейти на другой сайт, не дожидаясь, когда будет получена вся информация с текущего ресурса.

Во всех этих приложениях имеется разделение на несколько относительно независимых задач, которые могут выполняться разными потоками, имеющими различный приоритет. В примере с веб-браузером выделение ввода-вывода в отдельный поток обеспечивает « отзывчивость » пользовательского интерфейса приложения даже при интенсивной передаче данных.

Потоки, используемые в приложениях Win32, могут быть двух типов:

- ❑ потоки пользовательского интерфейса (*user-interface threads*);
- ❑ рабочие потоки (*worker threads*).

Потоки первого типа позволяют работать с ними при помощи механизма сообщений и имеют в своем составе оконную процедуру. Типичным примером такого потока является первичный поток Windows-приложения, содержащий кроме входной функции `WinMain` еще и оконную функцию `WndProc`.

Потоки второго типа, напротив, сообщения обрабатывать не могут и применяются обычно для фонового выполнения задач. В качестве примера можно привести поток с входной функцией `ThreadFunc` в рассмотренном выше приложении `ServerApp`.

Чаще всего приложению достаточно иметь один поток пользовательского интерфейса для взаимодействия с пользователем. Необходимость использования вторичных рабочих потоков определяется, исходя из решаемой задачи. Например, если ваша программа должна принимать в фоновом режиме данные из двух СОМ-портов, стоит организовать два рабочих потока для решения этих задач.

Несколько потоков пользовательского интерфейса в одном процессе можно обнаружить в таких приложениях, как Windows Explorer. Оно создает отдельный поток для каждого окна папки. Кроме параллельного решения различных задач, о котором говорилось выше, такая архитектура повышает надежность приложения. Если какая-то ошибка в Explorer приводит к краху одного из его потоков, то прочие потоки остаются работоспособны.

Не забывайте, что если вы решили сделать ваше приложение многопоточным, то необходимо очень тщательно продумать вопросы синхронизации работы потоков.

В заключение следует привести еще одну цитату из Джейфри Рихтера: «...Многопоточность следует использовать разумно».

10

Таймеры и время

При решении некоторых задач программа должна отслеживать текущее время или выполнять какие-либо действия с определенной периодичностью. Например, эта проблема возникает в приложениях, имитирующих аппаратуру, работающую в реальном масштабе времени, в игровых и мультимедийных приложениях, а также при проведении различных тестов. Кроме того, иногда требуется отладить критичные ко времени исполнения фрагменты кода, для чего нужен «хронометр» с высокой разрешающей способностью.

Win32 API содержит как функции для измерения текущего времени, так и функции для создания виртуальных таймеров — устройств, извещающих приложение об истечении заданного интервала времени. Для успешного применения этих программных средств необходимо учитывать их разрешающую способность и потенциальную точность измерения.

Важно понимать, что многозадачная операционная система Windows не является системой реального времени, поэтому любой виртуальный таймер в Windows не может гарантировать какой бы то ни было *фактической точности* отсчета временного интервала. Ведь в любой момент времени система может прервать выполнение вашего приложения, чтобы дать возможность поработать другому приложению (простой, вызванный прерыванием, чаще всего длится от 1 до 30 мс¹). Вероятность таких прерываний тем ниже, чем меньше рассматриваемый временной интервал и чем меньше других программ работает одновременно с вашим приложением. В то же время, как показывают эксперименты, мультимедийный таймер Windows обеспечивает вполне приемлемую фактическую точность отсчета временных интервалов для многих задач.

В этой главе кроме функций Win32 API мы рассмотрим также использование ассемблерной команды `rdtsc` для реализации «хронометра» с высокой разрешающей способностью.

Время Windows

Время Windows — это количество миллисекунд, прошедших с момента старта операционной системы. Этот формат времени поддерживается для обратной совме-

¹ Указанная здесь нижняя граница весьма приблизительна и относится к тому случаю, когда системный диспетчер потоков выделяет очередной квант этому же приложению.

стимости с 16-разрядными версиями Windows. Время Windows хранится в виде 32-разрядного целого числа без знака, которое сбрасывается в нулевое значение после того, как Windows проработает примерно 49,7 дней.

Операционная система управляет временем Windows через прерывания системного таймера, добавляя к текущему значению времени Windows приращение, равное периоду работы системного таймера. Кроме того, система периодически синхронизирует время Windows с показаниями часов реального времени, то есть с *системным временем*, рассматриваемым ниже.

Системный таймер Windows – это программное устройство, находящееся под управлением операционной системы (в отличие от аппаратного таймера, с которым работали программы под управлением MS-DOS).

Обычно период прерываний системного таймера составляет 10 или около 15 мс в зависимости от аппаратной платформы. Точное значение этого периода, называемое также *разрешением* системного таймера, можно получить с помощью функции `GetSystemTimeAdjustment`, имеющей следующий прототип:

```
BOOL GetSystemTimeAdjustment(PDWORD lpTimeAdjustment, PDWORD lpTimeIncrement,  
PBOOL lpTimeAdjustmentDisabled);
```

Эта функция предоставляет информацию, относящуюся к синхронизации системного времени и времени Windows. При этом значение, возвращаемое через второй параметр, как раз равно периоду прерывания системного таймера, выраженному в 100-наносекундных единицах.

Например, для компьютера, на котором тестировались программы, приводимые в данной книге (с процессором Intel Celeron CPU 2,0 ГГц и операционной системой Microsoft Windows 2000), разрешение системного таймера, полученное с помощью функции `GetSystemTimeAdjustment`, равно 15,625 мс.

Для получения текущего значения времени Windows предназначена функция `GetTickCount`. Функция возвращает число миллисекунд, прошедших с момента старта системы. Точность этого измерения определяется разрешающей способностью системного таймера.

Системное время

Системное время в Windows содержит информацию о текущих дате и времени и представляет собой так называемое UTC-время (Universal Time Coordinated). Время в формате UTC основывается на среднем времени по Гринвичу. Системное время может быть получено при помощи функции `GetSystemTime`:

```
VOID GetSystemTime(LPSYSTEMTIME lpSystemTime);
```

Функция записывает результат в структуру типа `SYSTEMTIME`, адрес которой задается параметром `lpSystemTime`. Структура типа `SYSTEMTIME` содержит поля для года, месяца, дня недели, дня, часов, минут, секунд и миллисекунд.

Так как системное время отсчитывается по Гринвичу, то, скорее всего, оно не совпадает с местным временем, которое отображается на панели задач. Получить значение местного времени можно при помощи функции `GetLocalTime`, которая возвращает информацию в том же формате, что и функция `GetSystemTime`. Если вы считаете, что ваше приложение может изменять системное время, то это можно осуществить при помощи вызова функции `SetSystemTime` или `SetLocalTime`. В неко-

торых случаях может оказаться полезной пара функций, работающих с информацией о часовом поясе, — `GetTimeZoneInformation` и `SetTimeZoneInformation`.

Системное время считывается с часов *реального времени*, встроенных в компьютер и имеющих автономное питание, в момент запуска Windows. Затем операционная система обеспечивает приращения системного времени, используя прерывания системного таймера, аналогично управлению временем Windows. Таким образом, точность измерения времени с помощью `GetSystemTime` также определяется разрешением системного таймера.

Измерение малых временных интервалов

Для оценки разрешающей способности программных средств работы со временем, предоставляемых операционной системой, очень важно иметь какой-то инструмент, позволяющий измерять время выполнения некоторого участка кода (в пределе — одной команды ассемблера или одной инструкции языка C++).

Функции считывания текущего времени `GetTickCount` и `GetSystemTime`, с которыми мы уже познакомились, теоретически, могут быть использованы для профилировки¹ некоторого фрагмента программы. Например, можно использовать следующий фрагмент кода:

```
int t1 = GetTickCount();
// ... профилируемый участок кода
int t2 = GetTickCount();
int elapsedTime = t2 - t1;
```

Но разрешение этих функций определяется разрешением системного таймера и для нашего компьютера, например, равно 15,625 мс. Это очень большая величина, так как при тактовой частоте процессора порядка 2 ГГц за это время могут быть выполнены сотни тысяч машинных команд.

Для более точных измерений малых интервалов следует использовать либо счетчик монитора производительности (`QueryPerformanceCounter`), либо счетчик меток реального времени, доступ к которому реализован при помощи ассемблерной команды `rdtsc`.

Использование счетчика монитора производительности

Win32 API содержит две функции для работы с высокоточным счетчиком монитора производительности. Функция `QueryPerformanceFrequency` возвращает количество приращений в секунду для этого счетчика:

```
BOOL QueryPerformanceFrequency(LARGE_INTEGER* lpFrequency);
```

Если счетчик монитора производительности не поддерживается системой, то функция вернет нулевое значение. Если счетчик все же поддерживается, то его частота записывается в 64-разрядную переменную типа `LARGE_INTEGER` по адресу

¹ Профилировкой называют измерение производительности как всей программы в целом, так и отдельных ее фрагментов.

`lpFrequency`. Тип `LARGE_INTEGER` определен как объединение структуры из двух 32-разрядных полей `LowPart`, `HighPart` и 64-разрядного поля `QuadPart`:

```
typedef union _LARGE_INTEGER {
    struct {
        DWORD LowPart;
        LONG HighPart;
    };
    LONGLONG QuadPart;
} LARGE_INTEGER;
```

Если компилятор не имеет встроенной поддержки 64-разрядных знаковых целых величин, то можно использовать поля `LowPart` (младшие 32 разряда) и `HighPart` (старшие 32 разряда).

Вторая функция, `QueryPerformanceCounter`, возвращает текущее значение счетчика монитора производительности:

```
BOOL QueryPerformanceCounter(LARGE_INTEGER* lpPerformanceCount);
```

записывая его по адресу `lpPerformanceCount`.

Для профилировки некоторого участка кода необходимо определить переменные

```
LARGE_INTEGER freq;
LARGE_INTEGER count_t1;
LARGE_INTEGER count_t2;
```

После этого следует получить значение частоты счетчика `freq`:

```
QueryPerformanceFrequency(&freq);
```

Сама профилировка выглядит следующим образом:

```
QueryPerformanceCounter(&count_t1);
// ... профилируемый участок кода
QueryPerformanceCounter(&count_t2);
double dt = count_t2.QuadPart - count_t1.QuadPart;
double elapsedTime = 1000 * dt / freq.QuadPart;
// время выполнения в миллисекундах
```

Какова разрешающая способность счетчика монитора производительности? Она определяется его частотой `freq.QuadPart`. Наши эксперименты на разных компьютерах показали, что независимо от тактовой частоты процессора частота `freq.QuadPart` равна 3 579 545 тиков в секунду. Таким образом, промежуток времени между двумя тиками равен 1/3579545 с, что составляет 0,279 мкс или 279 нс.

Сравните эту разрешающую способность (0,279 мкс) с разрешающей способностью функции `GetTickCount` (15 625 мкс), чтобы почувствовать разницу.

Однако в процессорах семейства Intel Pentium¹ появилась возможность еще более точного измерения малых интервалов времени, основанная на использовании счетчика меток реального времени.

Использование команды RDTSC

Начиная с Pentium III, процессоры этого семейства содержат доступный для программистов *счетчик меток реального времени TSC* (Time Stamp Counter). Это

¹ А также его клонов, таких, как, например, Intel Celeron, AMD, VIA и т. д.

64-разрядный регистр, содержимое которого инкрементируется с каждым тактом процессорного ядра. Каждый раз при аппаратном сбросе (сигналом RESET) отсчет в счетчике TSC начинается с нуля. Разрядность регистра обеспечивает отсчет времени без переполнения в течение сотен лет.

Команда `rdtsc` (read time stamp counter) возвращает количество тактов с момента запуска процессора, помещая результат в пару регистров общего назначения `EDX:EAX`. Функция на языке C++ может использовать эту команду следующим образом:

```
unsigned __int64 GetCycleCount(void) {
    _asm rdtsc
}
```

Если ваш компилятор не «понимает» команды `rdtsc`, используйте ее машинное представление:

```
_asm _emit 0x0F
_asm _emit 0x31
```

Так как частота работы у разных процессоров может различаться и колеблется в настоящее время от 60 МГц до 3 ГГц, то для измерения временных интервалов счетчик нужно отградуировать с помощью стандартных функций измерения времени ОС, например при помощи функции `Sleep`. Эта функция будет рассматриваться в разделе «Программирование задержек в исполнении кода».

Следующий фрагмент кода иллюстрирует механизм градуировки счетчика TSC:

```
unsigned __int64 t_start;
unsigned __int64 t_stop;
t_start = GetCycleCount();
Sleep(1000);
t_stop = GetCycleCount() - t_start - overhead;
double nCyclePer1microSec = t_stop / 1000000.;
```

Здесь функция `Sleep` приостанавливает выполнение потока на 1000 мс. Показания счетчика TSC фиксируются определенной выше функцией `GetCycleCount` перед вызовом функции `Sleep` и после возврата из нее. Разница этих показаний, запоминаемая в переменной `t_stop`, показывает количество машинных тактов, прошедших за одну секунду. Переменная `overhead` учитывает «накладные расходы», связанные с выполнением функции `GetCycleCount`, и должна быть вычислена заблаговременно. Получив величину `t_stop`, можно вычислять различные калибровочные коэффициенты, например коэффициент `nCyclePer1microSec`, определяющий, сколько тактов содержится в одной микросекунде.

После вычисления калибровочных коэффициентов профилировка некоторого участка программы осуществляется следующим образом:

```
t_start = GetCycleCount();
// ... профилируемый участок кода
t_stop = GetCycleCount() - t_start - overhead;
double elapsedTime = t_stop / nCyclePer1microSec; // время выполнения в микросекундах
```

Однако мы до сих пор не пояснили, что скрывается за таинственной поправочной величиной `overhead`.

Прежде чем сделать это, обратим ваше внимание на то, что профилировка небольших фрагментов программы сопряжена с рядом серьезных и не всегда очевидных проблем, незнание которых может привести к грубым ошибкам. Подробно

эти вопросы рассматриваются в книге К. Касперски [8]. Дело в том, что основу архитектуры любого процессора составляет *конвейер*, на котором выполняются микрооперации. Конвейер позволяет существенно повысить быстродействие процессора. Но для конвейерной системы такое понятие, как «время выполнения одной команды», весьма условно. Конвейер принято характеризовать двумя параметрами: «пропускной способностью» и «латентностью».

Пропускная способность — это количество инструкций, выполняемых за один такт при условии их непрерывного продвижения по конвейеру. *Латентность* — это полное время прохождения одной команды по конвейеру. Продвижение машинных инструкций по конвейеру сопряжено с рядом принципиальных трудностей: то не готовы операнды, то занято исполнительное устройство, — и в каждом таком случае конвейер простояивает. При выполнении команд ветвления ситуация еще более неопределенная, так как конвейер заполняется микрооперациями в предположении, что вычисления пойдут по одной из ветвей. Впоследствии это предположение может оказаться ложным, и тогда процессор вынужден вернуться к точке ветвления.

В лучшем случае время выполнения одной инструкции определяется пропускной способностью конвейера, а в худшем — его латентностью. Длина конвейера в современных процессорах колеблется от 12 до 36 стадий. Опустив другие подробности, обсуждаемые в издании [8], приведем заключительную рекомендацию. Минимальный промежуток времени, которому еще можно верить при использовании команды `rdtsc`, составляет по меньше мере от пятидесяти до ста тактов.

Другая тонкость, которую надо учитывать при профилировке, — это режим компиляции. Напомним, что в среде Visual Studio 6.0 этот режим определяется конфигурацией проекта. Возможны две конфигурации: *отладочная* (Win32 Debug) и *выпускная* (Win32 Release). Так вот, если вы занимаетесь профилировкой, то используйте только выпускную конфигурацию! Иначе результаты будут искажены, поскольку в код отладочной конфигурации компилятор вставляет дополнительные отладочные инструкции.

Вернемся к вопросу определения поправки *overhead*, с помощью которой учитываются «накладные расходы» в реализации функции `GetCycleCount`. Дело в том, что если выполнить фрагмент кода

```
t_start = GetCycleCount();
// профилируемый участок кода отсутствует
t_stop = GetCycleCount() - t_start;
```

то мы получим *ненулевое* значение `t_stop`. Казалось бы, полученное значение и надо присвоить переменной *overhead*. Но при многократных испытаниях приведенного фрагмента обнаруживается, что результаты его выполнения почти всегда будут разными. Так, для нашего процессора были получены значения 88, 100, 84, 320. Такое непостоянство, по-видимому, связано с теми особенностями функционирования конвейера, о которых говорилось выше. При этом ясно, что минимальное значение, в данном случае 84, и есть истинное значение, так как никакие побочные эффекты не могут привести к ускорению работы программы.

Таким образом, для правильного определения величины *overhead* необходимо повторить указанный выше фрагмент несколько раз и выбрать наименьшее значение. Эксперименты показали, что достаточно сделать десять таких замеров для достижения устойчивого результата.

На точность градуировки счетчика TSC влияет еще один фактор. Это погрешность отсчета эталонного интервала времени с помощью функции `Sleep`. Как будет показано в следующем разделе¹, наибольшей точности отсчета можно достичь, если предварительно выполнить две инструкции:

```
timeBeginPeriod(1);
Sleep(20);
```

а уже после этого выполнять код для градуировки счетчика, приведенный выше.

Функция `timeBeginPeriod(1)`, согласно спецификации в MSDN, устанавливает разрешение системного мультимедийного таймера, равное 1 мс. И как показали эксперименты, этот вызов влияет также и на работу функции `Sleep`. После окончания градуировки в указанном режиме необходимо вызвать функцию `timeEndPeriod(1)`, чтобы освободить системные ресурсы, используемые для поддержания высокоточного разрешения.

Для применения функций `timeBeginPeriod` и `timeEndPeriod` необходимо добавить в начало файла следующую директиву:

```
#include <Mmsystem.h>
```

а также подключить к проекту мультимедийную библиотеку `winmm.lib`².

Изложенные выше соображения учтены при разработке класса `KTimer`, объекты которого могут использоваться как высокоточные «хронометры». Исходный текст класса `KTimer` приведен в листинге 10.1. За основу взят код одноименного класса из [6], но здесь он доработан, чтобы повысить точность калибровки и предоставить разработчику более удобный сервис.

В файле `main.cpp` рассматриваемого приложения осуществляется тестирование класса `KTimer`.

Листинг 10.1. Проект KTimer

```
///////////
// KTimer.h
#ifndef KTIMER_H
#define KTIMER_H

#pragma warning(disable : 4035)
typedef enum { Msec, Usec, Ns_100, Ns_10, Nsec } TimeUnit;
static int et[10];
inline unsigned __int64 GetCycleCount(void) {
    _asm    rdtsc
}

class KTimer {
    __int64 t_start;
    __int64 t_stop;
    int overhead;
    // коэффициенты-делители
    double nCyclePer1nanoSec;
    double nCyclePer10nanoSec;
    double nCyclePer100nanoSec;
```

продолжение ↗

¹ «Программирование задержек в исполнении кода».

² Для Visual Studio 6.0 нужно использовать команду меню Project ▶ Settings..., перейти на вкладку Link, после чего ввести имя в текстовое поле Object/library modules.

Листинг 10.1. (продолжение)

```
double nCyclePer1microSec;
double nCyclePer1milliSec;
double divizor; // текущий делитель
public:
    KTimer(void) {
        // Калибровка overhead
        overhead = 0;
        for (int i = 0; i < 10; ++i) {
            t_start = GetCycleCount();
            et[i] = GetTick();
        }
        overhead = et[0];
        for (i = 1; i < 10; ++i)
            if (et[i] < overhead) overhead = et[i];

        // Калибровка делителей
        timeBeginPeriod(1);
        Sleep(20);

        Start();
        Sleep(1000);
        t_stop = GetCycleCount() - t_start - overhead;

        nCyclePer1nanoSec = t_stop / 1000000000.;
        nCyclePer10nanoSec = t_stop / 100000000.;
        nCyclePer100nanoSec = t_stop / 10000000.;
        nCyclePer1microSec = t_stop / 1000000.;
        nCyclePer1milliSec = t_stop / 1000.;
        timeEndPeriod(1);
        divizor = nCyclePer1milliSec; // делитель по умолчанию
    }

    void SetUnit(TimeUnit unit) {
        switch (unit) {
            case Msec: divizor = nCyclePer1milliSec; break;
            case Usec: divizor = nCyclePer1microSec; break;
            case Nsec: divizor = nCyclePer1nanoSec; break;
            case Ns_10: divizor = nCyclePer10nanoSec; break;
            case Ns_100: divizor = nCyclePer100nanoSec; break;
        }
    }

    inline void Start(void) { t_start = GetCycleCount(); }

    inline int GetTick(void) {
        return (int)(GetCycleCount() - t_start - overhead);
    }

    inline double GetTime(void) {
        t_stop = GetCycleCount() - t_start - overhead;
        return t_stop / divizor;
    }

    inline int GetTickPerUsec(void) { return (int)nCyclePer1microSec; }
    inline void uDelay(int uSec) {
        int tElapsed = 0;
        SetUnit(Usec);
```

```

Start();
while ( tElapsed < uSec)
    tElapsed = (int)GetTime();
}

int GetOverhead() { return overhead; }

#endif /* KTIMER_H */
////////////////////////////////////////////////////////////////
// Main.cpp
#include <windows.h>
#include <Mmsystem.h>
#include <fstream>
#include <iomanip>
using namespace std;

#include "KWnd.h"
#include "KTimer.h"

KTimer timer1;
KTimer timer2;
KTimer timer3;

#define N 5000000
int dTicks[N]; // число тактов счетчика TSC при профилировке нулевого участка
double grossWorkQuant[N]; // кванты времени работы приложения (брутто)
ofstream flog;
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
=====

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("Тестирование класса KTimer", hInstance, nCmdShow, WndProc);
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    int i, No = 0;
    char text[200];
    double networkQuant; // квант времени работы приложения (нетто)
    double sleepQuant; // квант времени "отдыха" приложения
    double totalTime; // общее время выполнения цикла for

    switch (uMsg)
    {
    case WM_CREATE:
        flog.open("flog.txt");
        timer3.Start();

```

продолжение ↗

Листинг 10.1. (продолжение)

```

timer2.Start();

for (i = 0; i < N; ++i) {
    timer1.Start();
    // ... ноль инструкций на профилируемом участке
    dTicks[i] = timer1.GetTick();
    if (dTicks[i] > 0) {
        grossWorkQuant[i] = timer2.GetTime();
        timer2.Start();
    }
}
totalTime = timer3.GetTime();
flog << "Отклонения измерений dTicks от ожидаемого значения 0\n";
flog << "-----\n";
flog << "      No      i      dTicks      sleepQuant      workQuant\n";
flog << "              (мс)          (мс)\n";
flog << "-----\n";
flog.flags(ios::fixed);
for (i = 0; i < N; ++i) {
    if (dTicks[i]) {
        No++;
        sleepQuant = (double)dTicks[i] / timer1.GetTickPerUsec() / 1000.;
        netWorkQuant = grossWorkQuant[i] - sleepQuant;
        flog << setw(8) << No << setw(10) << i << setw(10)
        << dTicks[i] << setw(12) << setprecision(5) << sleepQuant
        << setw(10) << setprecision(3) << netWorkQuant << endl;
    }
}
flog << "-----\n";
flog << "Общее время выполнения:  " << totalTime << " мс" << endl;
flog.close();

InvalidateRect(hWnd, NULL, TRUE);
break;

case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);
    sprintf(text, "overhead = %d,    CPU speed: %d Mhz\n",
            timer1.GetOverhead(), timer1.GetTickPerUsec());
    TextOut(hDC, 20, 10, text, strlen(text));
    EndPaint(hWnd, &ps);
    break;

case WM_DESTROY:
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
/////////////////////////////////////////////////////////////////

```

После ознакомления с текстом класса KTimer, размещенного в файле KTimer.h, обратите внимание на использование объектов этого класса. Вы можете объявить объект **timer**:

```
KTimer timer;
```

а затем осуществлять профилировку участка кода следующим образом:

```
timer.Start();
// ... профилируемый участок
double elapsedTime = timer.GetTime();
```

По умолчанию объект `timer` настроен на измерение времени в миллисекундах. Вы можете изменить разрешение хронометра, вызвав предварительно (до вызова метода `Start`) метод `SetUnit`. Этот метод принимает в качестве параметра одну из констант перечисляемого типа `TimeUnit`, которые соответствуют единицам измерения 1 мс, 1 мкс, 100 нс, 10 нс, 1 нс. Заметим, что при тактовой частоте процессора порядка 2 ГГц устанавливать единицы измерения менее 100 нс нецелесообразно (см. изложенные выше замечания о профилировке программ на конвейерных системах).

Если вместо метода `GetTime` применить метод `GetTick`, то будет реализована профилировка, результаты измерения которой выражены в тактах счетчика TSC. Этот метод может оказаться полезным для проведения экспериментальных исследований взаимодействия Windows и пользовательского приложения.

Класс `KTimer` содержит также метод `uDelay`, реализующий задержку в исполнении кода, заданную в микросекундах. Но эта тема будет обсуждаться в следующем разделе.

А теперь обратимся к коду файла `main.cpp`, написанному для тестирования класса `KTimer`. В глобальной области видимости определены три таймера класса `KTimer`: `timer1`, `timer2` и `timer3`. Работа с таймерами происходит в оконной процедуре при обработке сообщения `WM_CREATE`. Попутно здесь же осуществляется эксперимент по наблюдению за многозадачностью, реализуемой операционной системой Windows.

Идея эксперимента довольно проста. В теле цикла `for`, выполняемом `N` раз, находятся инструкции для профилировки программного участка *нулевой* длины. Профилировка выполняется вызовом методов `timer1.Start()` и `timer1.GetTick()`. Если `GetTick` возвращает нулевое значение, значит, «все нормально» и приложению никто не мешает спокойно работать. Если же `dTicks[i] > 0`, это свидетельствует о вмешательстве Windows. Понятно, что система переключилась на обслуживание других задач. Величина `dTicks[i]` в этом случае характеризует квант времени, в течение которого наше приложение «отдыхает». С помощью таймера `timer2` в этот момент фиксируется грубая оценка `grossWorkQuant[i]` для предыдущего кванта времени, в течение которого приложение проработало. Оценка считается грубой из-за того, что эта величина на самом деле представляет собой сумму «рабочего» кванта и кванта «отдыха». Общее время `totalTime` выполнения цикла `for` фиксируется с помощью таймера `timer3`.

После завершения цикла результаты измерений, сохраненные в массивах `dTicks` и `grossWorkQuant`, обрабатываются, чтобы их можно было сохранить в файле протокола `flog.txt`. В файл выводятся только те измерения, в которых значение `dTicks[i]` отличается от нуля. Каждая строка протокола содержит порядковый номер строки, номер прохода в цикле `for`, количество тактов счетчика TSC для профилировки «нулевого» участка, продолжительность «кванта отдыха» приложения и продолжительность «кванта работы» `netWorkQuant`.

Эксперименты с приложением `KTimer` на нашем компьютере показали, что при значениях `N`, не превышающих 100 000 (при общем времени `totalTime` около 9 мс), Windows *не прерывает* работу приложения и в массиве `dTicks` не фиксируется ни одного ненулевого значения¹.

¹ Учитите, что для чистоты эксперимента во время работы приложения `KTimer` рекомендуется воздержаться от действий с клавиатурой или мышью, а также не запускать других программ.

Результаты испытаний программы для $N = 5\ 000\ 000$ (пять миллионов) выглядят примерно так, как показано в листинге 10.2.

Листинг 10.2. Содержимое файла flog.txt после прогона программы с $N = 5\ 000\ 000$

Отклонения измерений dTicks от ожидаемого значения 0				
No	i	dTicks	sleepQuant (мс)	workQuant (мс)
1	82410	709236	0.35128	7.302
2	435817	3977324	1.96995	30.884
3	589957	46776	0.02317	13.646
4	770080	772092	0.38241	15.588
5	770977	352	0.00017	0.078
6	1299726	39176	0.01940	46.395
7	1479784	793520	0.39303	15.590
8	2357842	128812	0.06380	77.717
9	2471843	16792	0.00832	9.917
10	3068285	328824	0.16286	52.490
11	3772097	39672	0.01965	62.321
12	3952201	691372	0.34243	15.589
13	4128673	280	0.00014	15.379
14	4480734	37272	0.01846	31.130
15	4660754	6874012	3.40466	15.591
16	4801332	481832	0.23865	12.205
17	4978235	460724	0.22819	15.370

Общее время выполнения: 446.926 мс

Конечно, от прогона к прогону получаются разные результаты измерений, но общая качественная картина, представленная в листинге 10.2, сохраняется.

Представленный класс KTimer можно использовать в качестве измерительного инструмента в любом приложении. Для этого достаточно скопировать и подключить к вашему проекту файл KTimer.h, после чего нужно добавить директиву #include "KTimer.h" в исходный текст тех файлов, где используются объекты этого класса. Также необходимо подключить к проекту мультимедийную библиотеку winmm.lib.

Программирование задержек в исполнении кода

При создании реальных программ часто возникает необходимость «притормозить» исполнение кода в том или ином месте программы. В качестве простейшего примера можно привести игровую программу, перемещающую некий графический объект по экрану с помощью цикла while. Предположим, что вы отладили вашу программу на компьютере с процессором, имеющим тактовую частоту 500 МГц. Что произойдет, если эта программа будет исполняться на более мощном компьютере с тактовой частотой процессора 2 ГГц? Если не предусмотреть специальных мер, то объект будет двигаться гораздо быстрее, и не исключено, что играть станет совершенно невозможно. Типичное решение проблемы заключается в определении характеристик процессора и добавлении программируемых задержек в таких циклах. Другой пример, с которым мы столкнулись совсем недавно, —

использование программируемой задержки для калибровки счетчика меток реального времени в классе KTimer.

Использование функции Sleep

Самый простой способ обеспечения задержки реализуется вызовом функции `Sleep`:

```
VOID Sleep(DWORD dwMilliseconds);
```

Функция приостанавливает выполнение потока на то количество миллисекунд, которое указано в качестве параметра.

Вопрос точности, с которой функция обеспечит затребованную задержку, в MSDN не освещается. Чтобы разобраться с этим, создадим тестовую программу, код которой приведен в листинге 10.3.

Листинг 10.3. Проект SleepTest¹

```
////////////////////////////////////////////////////////////////////////
// SleepTest.cpp
#include <windows.h>
#include <stdio.h>
#include <Mmsystem.h>

#include "KWnd.h"
#include "KTimer.h"

KTimer timer;
#define SLEEP_TIME 1
#define N 100

double realTimeInterval[N];
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
=====

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("SleepTest", hInstance, nCmdShow, WndProc);
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
```

продолжение ↗

¹ Для нормальной компиляции и сборки проекта не забудьте подключить к проекту мультимедийную библиотеку `winmm.lib` (см. предыдущий раздел).

Листинг 10.3. (продолжение)

```
char text[200];
int i;
double minT, maxT, amount;

switch (uMsg)
{
case WM_CREATE:

    //timeBeginPeriod(1);    // #1

    Sleep(20);              // #2
    for (i = 0; i < N; ++i) {
        timer.Start();
        Sleep (SLEEP_TIME);
        realTimeInterval[i] = timer.GetTime();
    }

    //timeEndPeriod(1);      // #3
    InvalidateRect(hWnd, NULL, TRUE);
    break;

case WM_PAINT:
    hDC = BeginPaint(hWnd, &ps);
    sprintf(text, "Фактическая величина задержки для SLEEP_TIME = %d", SLEEP_TIME);
    TextOut(hDC, 0, 20, text, strlen(text));
    x = 15; y = 40;
    for (i = 0; i < 20; ++i) {
        sprintf(text, "%.2f\\0", realTimeInterval[i]);
        TextOut(hDC, x, y, text, strlen(text));
        y += 16;
    }

    // Определяем min и max значения
    minT = maxT = amount = realTimeInterval[0];
    for (i = 1; i < N; ++i) {
        if (realTimeInterval[i] < minT)
            minT = realTimeInterval[i];
        if (realTimeInterval[i] > maxT)
            maxT = realTimeInterval[i];
        amount += realTimeInterval[i];
    }
    sprintf(text, "minT = %.2f,  maxT = %.2f, average = %.2f\\0",
           minT, maxT, amount / N);
    TextOut(hDC, x, y, text, strlen(text));
    EndPaint(hWnd, &ps);
    break;

case WM_DESTROY:
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}//////////
```

В программе производится профилировка функции Sleep с помощью объекта timer класса KTimer. Профилирующий код находится в блоке обработки сообщения WM_CREATE.

Испытания повторяются N раз с помощью цикла for. Результаты измерений в миллисекундах сохраняются в массиве realTimeInterval.

В блоке обработки сообщения WM_PAINT определяются минимальное, максимальное и среднее значения для реальной задержки. Эти величины выводятся в окно программы вместе с результатами первых двадцати измерений.

Величина заказанной задержки задается с помощью макроса SLEEP_TIME. В табл. 10.1 приведены результаты тестирования для разных значений SLEEP_TIME.

Таблица 10.1. Реальная задержка, обеспечиваемая функцией Sleep

SLEEP_TIME, Реальная задержка, мс	Минимальная	Максимальная	Средняя	Ближайшее большее значение, кратное разрешению системного таймера, мс
1	9,6	21,64	15,63	15,625
5	11,74	21,81	15,62	15,625
10	9,7	21,54	15,68	15,625
16	25,28	37,16	31,25	31,25
20	25,26	37,24	31,25	31,25
40	40,94	52,71	46,87	46,875
100	103,43	115,30	109,36	109,375
1000	1000,63	1002,32	1000,76	1000

Из приведенных данных видно, что реальная задержка, обеспечиваемая функцией Sleep, находится в некоторой области ближайшего большего значения, кратного разрешению системного таймера (15,625 мс).

Теперь обратите внимание на инструкцию Sleep(20), помеченную в программе номером #2. Если ее убрать, то самый первый вызов функции Sleep(SLEEP_TIME) даст результат realTimeInterval[0], значительно отличающийся от последующих вызовов. При малых величинах SLEEP_TIME отклонения от среднего значения достигают 80 %, а для задержки в 1000 мс — порядка 10 %. Трудно дать внятное объяснение этому факту, пользуясь информацией из справочной системы MSDN. Но, конечно, следует принять его во внимание, когда вы используете функцию Sleep для калибровки измерительного инструмента (см. исходный текст класса KTimer).

Наши дальнейшие эксперименты показали, что разрешение функции Sleep можно существенно повысить, если предварительно вызвать функцию timeBeginPeriod(1), которая относится к мультимедийной библиотеке Windows. Эта функция устанавливает разрешение системного мультимедийного таймера, равное 1 мс.

Когда повышенное разрешение системного таймера становится ненужным в дальнейшей работе приложения, рекомендуется вызвать функцию timeEndPeriod(1) для освобождения задействованных системных ресурсов.

Чтобы убедиться в этом, раскомментируйте в листинге 10.3 инструкции, помеченные номерами #1 и #3, и проведите испытания повторно. Скорее всего, вы получите результаты, близкие к тем, которые показаны в табл. 10.2.

Таблица 10.2. Реальная задержка, обеспечиваемая функцией Sleep после вызова timeBeginPeriod(1)

SLEEP_TIME, мс	Реальная задержка, мс	Минимальная	Максимальная	Средняя
1	1,66	4,33		1,98
2	2,78	3,07		2,93
3	3,75	4,06		3,91
5	5,38	7,11		5,87
10	10,26	12,93		10,78
20	20,21	20,80		20,51
100	100,20	100,95		100,57
1000	999,85	1001,81		1000,34

Хотя в области малых значений (менее 10 мс) реальная задержка получается с существенной относительной погрешностью, все же эти результаты намного предпочтительнее тех, которые были показаны в табл. 10.1.

Использование метода uDelay класса KTimer

Если вам нужна задержка в микросекундном диапазоне, то функция Sleep становится бесполезной. Возможны разные варианты решения этой проблемы, один из них — вызов метода uDelay класса KTimer. Чтобы посмотреть, как он работает, можно модифицировать программу SleepTest (см. листинг 10.3) следующим образом.

Добавьте макрос

```
#define U_SEC 1 // задержка в микросекундах
```

и определение еще одного объекта класса KTimer:

```
KTimer timer1;
```

Затем замените код профилировки, содержащийся в блоке обработки сообщения WM_CREATE, на следующий фрагмент:

```
timer1.SetUnit(Usec);
timer1.SetUnit(Usec);
for (i = 0; i < N; ++i) {
    timer1.Start();
    timer1.uDelay(U_SEC);
    realTimeInterval[i] = timer1.GetTime();
}
```

После этого проведите испытания программы для разных значений U_SEC. На нашем компьютере были получены результаты, показанные в табл. 10.3.

COBET

Не забывайте, что реализация класса KTimer основана на использовании ассемблерной команды rdtscll, которая поддерживается процессорами модельных рядов не ниже Pentium III. Если вам нужно обеспечить работоспособность приложения на более старых аппаратных платформах, рекомендуем для реализации малых задержек использовать приведенный ниже класс QTimer.

Таблица 10.3. Реальная задержка, обеспечиваемая методом uDelay класса KTimer

U_SEC, мкс	Реальная задержка, мкс	Минимальная	Максимальная	Средняя
1	1,20	1,25	1,21	
2	2,19	2,22	2,20	
5	5,14	5,18	5,15	
10	10,14	10,16	10,15	
50	50,20	50,29	50,27	
100	100,15	100,24	100,23	
1000	1000,15	1679,37	1007,03	

Класс QTimer

Класс QTimer имеет интерфейс, совпадающий с интерфейсом класса KTimer. Но реализация его методов основана на использовании счетчика монитора производительности. В листинге 10.4 приведено определение данного класса.

Листинг 10.4. Класс QTimer

```
////////////////////////////////////////////////////////////////
// QTimer.h
#ifndef QTIMER_H
#define QTIMER_H

#pragma warning(disable : 4035)
typedef enum { Msec, Usec, Ns_100, Ns_10, Nsec } TimeUnit;
class QTimer {
    LARGE_INTEGER freq;
    LARGE_INTEGER count_beg;
    LARGE_INTEGER count_end;

    LONGLONG ll_c_beg;
    LONGLONG ll_c_end;
    long d_tic;

    double dt;
    DWORD lowPart;
    LONG highPart;

    // коэффициенты-делители
    double nCyclePer1nanoSec;
    double nCyclePer10nanoSec;
    double nCyclePer100nanoSec;
    double nCyclePer1microSec;
    double nCyclePer1milliSec;
    double divisor; // текущий делитель

public:
    QTimer(void) {
        // Калибровка
        timeBeginPeriod(1);
        Sleep(20);

        BOOL success = QueryPerformanceFrequency(&freq);
        if (!success)
```

продолжение ↗

Листинг 10.4. (продолжение)

```
MessageBox(NULL, "Счетчик QueryPerformance не поддерживается",
          0, MB_OK | MB_ICONSTOP);
lowPart = freq.LowPart;
highPart = freq.HighPart;

// Калибровка делителей
nCyclePer1nanoSec    = freq.LowPart / 1000000000.;
nCyclePer10nanoSec   = freq.LowPart / 100000000.;
nCyclePer100nanoSec  = freq.LowPart / 10000000.;
nCyclePer1microSec   = freq.LowPart / 1000000.;
nCyclePer1milliSec   = freq.LowPart / 1000.;
timeEndPeriod(1);
divizor = nCyclePer1milliSec; // делитель по умолчанию
}

void SetUnit(TimeUnit unit) {
    switch (unit) {
        case Msec: divizor = nCyclePer1milliSec; break;
        case Usec: divizor = nCyclePer1microSec; break;
        case Nsec: divizor = nCyclePer1nanoSec; break;
        case Ns_10: divizor = nCyclePer10nanoSec; break;
        case Ns_100: divizor = nCyclePer100nanoSec; break;
    }
}

inline void Start(void) { QueryPerformanceCounter(&count_beg); }

inline long GetTick(void) {
    QueryPerformanceCounter(&count_end);
    ll_c_beg = count_beg.QuadPart;
    ll_c_end = count_end.QuadPart;
    d_tic = ll_c_end - ll_c_beg;
    return d_tic;
}

inline double GetTime(void) {
    QueryPerformanceCounter(&count_end);
    ll_c_beg = count_beg.QuadPart;
    ll_c_end = count_end.QuadPart;
    d_tic = ll_c_end - ll_c_beg;

    return d_tic / divizor;
}

inline int GetTickPerUsec(void) { return (int)nCyclePer1microSec; }

inline void uDelay(int uSec) {
    int tElapsed = 0;
    SetUnit(Usec);
    Start();
    while ( tElapsed < uSec)
        tElapsed = (int)GetTime();
}
```

```
};

#endif /* QTIMER_H */
////////////////////////////////////////////////////////////////
```

Поскольку класс `QTimer` имеет такой же интерфейс, что и класс `KTimer`, использование его объектов не отличается от использования объектов класса `KTimer`.

Стандартный таймер

Стандартный таймер в Windows – это устройство, периодически уведомляющее приложение о завершении заданного интервала времени. Вы можете присоединить стандартный таймер к своей программе с помощью функции `SetTimer`:

```
UINT_PTR SetTimer(
    HWND hWnd,           // дескриптор окна
    UINT_PTR nIDEvent,   // идентификатор таймера
    UINT uElapse,        // интервал в миллисекундах
    TIMERPROC lpTimeProc // адрес функции - обработчика сообщения WM_TIMER
);
```

Первый параметр, `hWnd`, содержит дескриптор окна, ассоциированного с данным таймером. Если этот параметр равен `NULL`, то с таймером не связывается никакое окно и сообщения от таймера будут приходить в специально созданную для этого функцию.

Второй параметр, `nIDEvent`, позволяет указывать идентификатор таймера, которым может быть произвольное целое число (но не нуль). Если программа использует более одного таймера, то рекомендуется определить идентификаторы таймеров в виде именованных констант, например, с помощью типа `enum` или директивы `#define`. Это улучшает читаемость кода программы. Если параметр `hWnd` равен `NULL`, то параметр `nIDEvent` игнорируется.

Третий параметр, `uElapse`, задает интервал, который может находиться в пределах (теоретически) от 1 до 4 294 967 295 мс, что составляет около 50 дней. Это значение определяет темп, с которым Windows будет посыпать вашей программе сообщения `WM_TIMER`. Сообщения `WM_TIMER` направляются либо оконной процедуре для окна `hWnd`, если параметр `lpTimeProc` равен `NULL`, либо функции обратного вызова с адресом `lpTimeProc` — в противном случае.

Если на месте `hWnd` указано значение `NULL`, то возвращаемое функцией значение является идентификатором созданного таймера. В любом случае функция `SetTimer` возвращает нулевое значение, если она не смогла создать таймер.

Если таймер по истечении некоторого времени больше не нужен, то рекомендуется его уничтожить, вызвав функцию `KillTimer`:

```
BOOL KillTimer(HWND hWnd, UINT_PTR uIDEvent);
```

В параметре `hWnd` указывается дескриптор окна, с которым был связан таймер. Это значение должно совпадать со значением `hWnd`, указанным при вызове функции `SetTimer`.

Второй параметр содержит идентификатор уничтожаемого таймера. На его месте используется либо идентификатор, указанный при создании таймера, если он создавался для окна, либо значение, возвращенное функцией `SetTimer` (для таймера, имеющего собственную функцию обработки сообщений).

Первый способ использования стандартных таймеров

В первом способе таймер подключается к окну, заданному параметром hWnd при вызове функции SetTimer, а параметр lpTimeProc получает значение NULL. В этом случае функция окна, к которому подключен таймер, будет получать сообщения от таймера с кодом WM_TIMER.

Допустим, что вашей программе требуется два таймера. Один таймер имеет период срабатывания в 1 с, а другой обладает периодом срабатывания в 1 мин. Вы могли бы сначала определить идентификаторы таймеров с помощью инструкций #define:

```
#define TIMER_SEC 1
#define TIMER_MIN 2
```

Затем либо в теле функции WinMain, либо в блоке обработки сообщения WM_CREATE оконной процедуры нужно вызвать функции для создания двух таймеров:

```
SetTimer(hWnd, TIMER_SEC, 1000, NULL);
SetTimer(hWnd, TIMER_MIN, 60000, NULL);
```

Поскольку параметр wParam сообщения WM_TIMER содержит идентификатор таймера, то логика обработки сообщения WM_TIMER в оконной процедуре обычно выглядит примерно так:

```
case WM_TIMER:
switch (wParam) {
    case TIMER_SEC:
        ... // обработка одного сообщения в секунду
        break;
    case TIMER_MIN:
        ... // обработка одного сообщения в минуту
        break;
}
break;
```

После того как эти таймеры перестанут быть нужными для работы приложения, рекомендуется их уничтожить, вызвав следующие инструкции:

```
KillTimer(hWnd, TIMER_SEC);
KillTimer(hWnd, TIMER_MIN);
```

Отметим, что на практике реальный интервал генерации сообщений WM_TIMER может значительно отличаться от заданного вами значения для параметра uElapse, особенно для малых значений. Здесь необходимо учитывать следующие недокументированные особенности функционирования стандартного таймера. Во-первых, реальный интервал не может быть меньше разрешения системного таймера. Во-вторых, реальный интервал всегда является некоторым приближением к ближайшему большему значению, кратному разрешению системного таймера. Например, если системный таймер имеет разрешение 15,625 мс, то ряд допустимых значений, в окрестностях которых реализуются значения реального интервала, имеет вид 15,625 мс, 31,25 мс, 46,875 мс, 62,5 мс и т. д.

Приведем результаты экспериментов по измерению реального интервала генерации сообщений WM_TIMER, проведенных на компьютере с процессором Intel

Celeron CPU 2,0 ГГц и операционной системой Microsoft Windows 2000 (разрешение системного таймера 15,625 мс). Измерения осуществлялись путем фиксации времени поступления в оконную процедуру сообщения WM_TIMER для тысячи последовательных тиков стандартного таймера. Тестовая программа приведена в листинге 10.5.

Листинг 10.5. Проект SetTimer1Test

```
////////////////////////////////////////////////////////////////////////
// SetTimer1Test.cpp
#include <windows.h>
#include <stdio.h>
#include "KWnd.h"
#include "KTimer.h"

KTimer timer;
#define ID_TIMER 1
#define TIME_PERIOD 1
#define N 1000
double realTimeInterval[N];

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
//=====================================================
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWnd mainWnd("SetTimer1 - test", hInstance, nCmdShow, WndProc);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

//=====================================================
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    static int x, y;
    char text[200];
    int i;
    double minT, maxT, amount;
    static int count = -1;

    switch (uMsg)
    {
    case WM_CREATE:
        SetTimer(hWnd, ID_TIMER, TIME_PERIOD, NULL);
        break;
    case WM_TIMER:
        if (count >= 0 && count < N)
            realTimeInterval[count] = timer.GetTime();
        timer.Start();
        count++;
    
```

продолжение ↗

Листинг 10.5. (продолжение)

```

if (count == N) InvalidateRect(hWnd, NULL, TRUE);
break;

case WM_PAINT:
hDC = BeginPaint(hWnd, &ps);
sprintf(text, "Фактическая величина задержки для TIME_PERIOD = %d", TIME_PERIOD);
TextOut(hDC, 0, 20, text, strlen(text));
/* код, аналогичный приведенному в листинге 10.3 */
EndPaint(hWnd, &ps);
break;

case WM_DESTROY:
KillTimer(hWnd, ID_TIMER);
PostQuitMessage(0);
break;

default:
return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}
/////////////////////////////////////////////////////////////////

```

Результаты тестирования представлены в табл. 10.4.

Таблица 10.4. Замеры реального интервала срабатывания стандартного таймера

TIME_PERIOD, мс	Реальный интервал, мс	Ближайшее большее значение, кратное разрешению системного таймера, мс		
	Минимальный	Максимальный	Средний	
1	14,79	16,16	15,62	15,625
10	15,09	16,17	15,62	15,625
20	30,47	32,07	31,26	31,25
40	46,18	47,56	46,88	46,875
100	108,90	110,10	109,38	109,375
1000	1000,07	1015,70	1000,86	1000

Отметим, что указанные результаты получены для работы приложения как бы в «стерильных условиях». Иными словами, во время эксперимента на компьютере не было других работающих *ресурсоемких* приложений. Это раз. Кроме того, пользователь не прикасался к окну тестируемого приложения. Это два.

Если же во время эксперимента многократно перемещать с помощью мыши окно работающего приложения, то разброс значений реального интервала срабатывания таймера¹ будет совсем другим. Например, для заказанного интервала 100 мс был получен разброс 0,52–503,38 мс. Этот факт требует некоторых объяснений, но сначала нужно разобраться, каков механизм продвижения сообщения WM_TIMER от виртуального таймера к оконной процедуре.

¹ Разница между минимальным и максимальным значениями.

Таймерные прерывания по своей природе являются асинхронными, так как по отношению к выполняемой программе их появление предсказать нельзя. Они ставятся в обычную очередь сообщений приложения и обрабатываются, как все остальные сообщения. Но это еще не все. Сообщение WM_TIMER обладает самым низким приоритетом по отношению к другим сообщениям. Единственным исключением является сообщение WM_PAINT, обладающее еще более низким приоритетом. Поэтому функция GetMessage отправляет сообщение WM_TIMER на обработку только тогда, когда в очереди сообщений не осталось более приоритетных сообщений.

Таким образом, когда пользователь перемещает окно или изменяет его размеры, соответствующие сообщения устремляются в очередь сообщений и, обладая более высоким приоритетом, оттесняют сообщение WM_TIMER в конец очереди.

Сообщения WM_TIMER похожи на сообщения WM_PAINT еще в одном аспекте. Когда приложение чем-то занято и не успевает обработать предыдущее сообщение WM_TIMER, то Windows не накапливает в очереди сообщений несколько сообщений WM_TIMER. Вместо этого Windows объединяет несколько сообщений WM_TIMER из очереди в одно сообщение. В такой ситуации приложение не способно определить количество потерянных сообщений WM_TIMER.

Теперь мы можем объяснить наблюдаемый разброс 0,52–503,38 мс для реального интервала срабатывания таймера при ожидаемом интервале 109,375 мс. Значение 503,38 мс отражает ситуацию, когда четыре последовательных сообщения WM_TIMER были потеряны и только пятое из них было обработано. Причем оно попало в очередь сообщений не сразу, а с задержкой примерно 43,5 мс. Значение 0,52 мс могло получиться в результате аналогичной задержки примерно на 108,855 мс при постановке в очередь предыдущего сообщения WM_TIMER (из-за обработки сообщения WM_MOVING с более высоким приоритетом), поэтому от текущего интервала осталось примерно 0,52 мс.

Пример первого способа использования стандартного таймера вы можете найти в листинге 12.1 (глава 12).

Второй способ использования стандартных таймеров

При первом способе установки таймера сообщения WM_TIMER посылаются в обычную оконную процедуру. Второй способ предписывает Windows пересыпать сообщения таймера другой функции из вашей программы. Это функция обратного вызова (call-back), и она должна быть определена в глобальной области видимости следующим образом¹:

```
VOID CALLBACK TimerProc(HWND hwnd, UINT uMsg, UINT_PTR idEvent,  
DWORD dwTime) {  
    ... // обработка сообщения WM_TIMER  
}
```

Перед установкой таймера вы должны определить переменную для сохранения идентификатора таймера, например:

```
static int idTimer;
```

¹ Вместо имени TimeProc можно использовать любое другое имя.

Сама установка таймера выглядит следующим образом:

```
idTimer = SetTimer(NULL, NULL, TIME_PERIOD, (TIMERPROC)TimerProc);
```

где **TIME_PERIOD** — именованная константа, задающая интервал в миллисекундах.

После окончания работы с таймером он уничтожается вызовом функции `KillTimer(NULL, idTimer)`:

В заключение следует отметить, что принципиальных различий в функционировании стандартных таймеров, созданных первым или вторым способом, не наблюдается. И в том, и в другом случаях таймеры генерируют сообщения **WM_TIMER**, поступающие в очередь сообщений приложения, а проблемы, связанные с обработкой этого сообщения, уже рассматривались выше.

Хотя стандартный таймер Windows не отличается высокой точностью и надежностью, тем не менее, приложения могут использовать его сервис, если отклонения в периоде генерации сообщений таймера не очень важны для решаемой задачи. В противном случае рекомендуется прибегать к услугам мультимедийного таймера.

Мультимедийный таймер

Для использования в вашем приложении мультимедийного таймера необходимо подключить к проекту мультимедийную библиотеку `winmm.lib`¹ и добавить в начало файла, в котором вызываются функции библиотеки, следующую директиву:

```
#include <Mmsystem.h>
```

Содержащаяся в библиотеке функция `timeGetDevCaps` позволяет узнать поддерживаемое системой разрешение мультимедийного таймера. Для этого определите переменную структурного типа

```
TIMECAPS tc;
```

и вызовите функцию

```
timeGetDevCaps(&tc, sizeof(TIMECAPS));
```

В результате этого вызова поля `tc.wPeriodMin` и `tc.wPeriodMax` будут содержать минимальное и максимальное разрешения в миллисекундах, поддерживаемые для мультимедийного таймера. Для нашего компьютера, например, были получены значения 1 мс и 1 000 000 мс соответственно.

Мультимедийная библиотека содержит функции `timeBeginPeriod` и `timeEndPeriod`, предназначенные для установки и отмены конкретного разрешения мультимедийного таймера. Величина разрешения в миллисекундах передается в виде параметра в обеих функциях.

MSDN рекомендует вызывать функцию `timeBeginPeriod(tc.wPeriodMin)` непосредственно перед тем, как обратиться к сервису мультимедийного таймера. Повышенное разрешение таймера, по-видимому, реализуется в системе при помощи создания отдельного потока с высоким приоритетом выполнения. Поэтому рекомендуется отменять режим повышенного разрешения таймера (`timeEndPeriod`), как только он перестает быть нужным.

¹ При работе с Visual Studio 6. нужно выполнить команду меню Project ▶ Settings..., перейти на вкладку Link и в текстовом поле Object/library modules указать имя библиотеки `winmm.lib`.

Ранее мы уже использовали функции `timeBeginPeriod` и `timeEndPeriod` для повышения точности работы функции `Sleep`. Степень влияния этих функций на работу собственно мультимедийного таймера мы выясним, когда проведем эксперименты с тестовой программой.

Функции `timeSetEvent` и `timeKillEvent`

Мультимедийный таймер выполняется в своем собственном потоке. Он отслеживает заданный временной интервал и активизирует по истечении этого интервала таймерное событие. После этого Windows либо вызывает заданную функцию обратного вызова, либо устанавливает заданное событие.

Сервис мультимедийного таймера вызывается с помощью функции `timeSetEvent`:

```
MMRESULT timeSetEvent(UINT uDelay, UINT uResolution, LPTIMECALLBACK lpTimeProc,
DWORD dwUser, UINT fuEvent);
```

В параметре `uDelay` указывается задержка активизации таймерного события. В качестве единиц измерения используются миллисекунды. Если значение параметра выходит за пределы задержек, поддерживаемых таймером, то функция вернет код ошибки.

Параметр `uResolution` содержит разрешение таймерного события в миллисекундах. Чем меньше значение этого параметра, тем выше разрешение. Нулевое значение параметра означает, что периодические события таймера должны появляться с максимально возможной точностью. Однако чем выше разрешение, тем больше будут накладные расходы операционной системы, поэтому желательно выбирать максимальное значение параметра `uResolution`, при котором удовлетворяются потребности приложения.

Параметр `lpTimeProc` содержит адрес функции обратного вызова, которая вызывается после установки таймерного события. Если параметр `fuEvent` содержит флаг `TIME_CALLBACK_EVENT_SET` или `TIME_CALLBACK_EVENT_PULSE`, то параметр `lpTimeProc` интерпретируется как дескриптор объекта «событие».

Параметр `dwUser` может содержать данные, передаваемые в функцию `lpTimeProc`, интерпретация которых определяется программистом. В частности, эти данные иногда используются для организации обратной связи с вызывающей функцией.

И наконец, в параметре `fuEvent` задается тип таймерного события. Этот параметр может содержать один из следующих флагов:

Значение	Интерпретация
<code>TIME_ONESHOT</code>	Событие происходит один раз после истечения <code>uDelay</code> миллисекунд
<code>TIME_PERIODIC</code>	Событие происходит каждые <code>uDelay</code> миллисекунд

Также параметр `fuEvent` может содержать один из флагов, определяющих действия Windows по истечении заданного интервала времени:

Значение	Интерпретация
<code>TIME_CALLBACK_FUNCTION</code>	Вызывается функция с адресом <code>lpTimeProc</code> (значение по умолчанию)
<code>TIME_CALLBACK_EVENT_SET</code>	Вызывается функция <code>SetEvent</code> для установки события с дескриптором <code>lpTimeProc</code>
<code>TIME_CALLBACK_EVENT_PULSE</code>	Вызывается функция <code>PulseEvent</code> для установки (с последующим сбросом) события с дескриптором <code>lpTimeProc</code>

В случае успешного завершения функция `timeSetEvent` возвращает идентификатор таймерного события. Если при выполнении функции произошла ошибка, то функция возвращает значение `NULL`.

Как только работа с таймером будет завершена, следует вызвать функцию `timeKillEvent` для освобождения задействованных системных ресурсов. В качестве параметра функции `timeKillEvent` передается тот идентификатор таймерного события, который был получен от функции `timeSetEvent`.

Тестирование мультимедийного таймера

Чтобы сравнить поведение мультимедийного таймера с поведением стандартного таймера, проведем эксперименты по измерению реального интервала вызова функции `lpTimeProc`. Тестовая программа приведена в листинге 10.6.

Листинг 10.6. Проект MmTimerTest

```
//////////  
// MmTimerTest.cpp  
#include <windows.h>  
#include <stdio.h>  
#include <Mmsystem.h>  
#include <fstream>  
using namespace std;  
  
#include "KWnd.h"  
#include "KTimer.h"  
  
ofstream flog;  
KTimer timer;  
  
enum UserMsg { UM_READY = WM_USER+1 };  
#define TIME_PERIOD 1  
#define N 1000  
double realTimeInterval[N];  
  
void CALLBACK TimeProc(UINT, UINT, DWORD, DWORD);  
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);  
=====  
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)  
{  
    flog.open("flog.txt");  
    MSG msg;  
    KWnd mainWnd("MM timer - test", hInstance, nCmdShow, WndProc);  
  
    while (GetMessage(&msg, NULL, 0, 0)) {  
        TranslateMessage(&msg);  
        DispatchMessage(&msg);  
    }  
  
    flog.precision(3);  
    for (int i = 0; i < N; ++i)  
        flog << realTimeInterval[i] << endl;  
    flog.close();  
    return msg.wParam;  
}  
=====
```

```

LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hDC;
    PAINTSTRUCT ps;
    char text[200];
    int i;
    const int i0 = 30; // номер первого обрабатываемого измерения
    double minT, maxT, amount;
    static MMRESULT mmResult;

    switch (uMsg)
    {
        case WM_CREATE:
//timeBeginPeriod(1);      // #1

            mmResult = timeSetEvent(TIME_PERIOD, 0, TimeProc, reinterpret_cast<DWORD>(hWnd),
TIME_PERIODIC);
            if (!mmResult)
                MessageBox(hWnd, "Мультимедийный таймер не создан", "Error", MB_OK);
            break;

        case WM_PAINT:
            hDC = BeginPaint(hWnd, &ps);
            // Определяем min и max значения
            minT = maxT = amount = realTimeInterval[i0];
            for (i = i0 + 1; i < N; ++i) {
                if (realTimeInterval[i] < minT)
                    minT = realTimeInterval[i];
                if (realTimeInterval[i] > maxT)
                    maxT = realTimeInterval[i];
                amount += realTimeInterval[i];
            }
            sprintf(text, "minT = %.3f, maxT = %.3f, average = %.3f\n",
minT, maxT, amount / (N - i0));
            TextOut(hDC, 10, 10, text, strlen(text));
            EndPaint(hWnd, &ps);
            break;

        case UM_READY:
            MessageBox(hWnd, "Готово", "Завершение задания", MB_OK);
            SendMessage(hWnd, WM_DESTROY, 0, 0);
            break;

        case WM_DESTROY:
            timeKillEvent(mmResult);
//timeEndPeriod(1);      // #2
            PostQuitMessage(0);
            break;

        default:
            return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }
    return 0;
}

//=====
void CALLBACK TimeProc(UINT wTimerID, UINT msg, DWORD dwUser, DWORD dw1, DWORD dw2)

```

продолжение ⤵

Листинг 10.6 (продолжение)

```

{
    static int count = -1;
    HWND hwnd = reinterpret_cast<HWND>(dwUser);

    if (count >= 0 && count < N)
        realTimeInterval[count] = timer.GetTime();
    timer.Start();
    count++;
    if (count == N) {
        InvalidateRect(hwnd, NULL, TRUE);
        SendMessage(hwnd, UM_READY, 0, 0);
    }
}
/////////////////////////////////////////////////////////////////

```

Обратите внимание на то, что при вызове функции `timeSetEvent` в качестве четвертого параметра передается дескриптор окна приложения с преобразованием его к типу `DWORD`. Внутри функции `TimeProc` этот дескриптор используется для обратной связи с оконной процедурой при вызове функций `InvalidateRect` и `SendMessage`.

Период срабатывания мультимедийного таймера задается макросом `TIME_PERIOD`. Вычисляя минимальное, максимальное и среднее значения периода срабатывания таймера, мы отбрасываем первые i_0 измерений. В программе константа i_0 определена со значением 30. Это сделано, чтобы устранить погрешность измерений, обусловленную особенностями функционирования конвейерной системы процессора.

Выполнение программы для периода, равного 1 мс, дает следующие итоговые результаты:

```
minT = 0.649, maxT = 1.973, average = 1.00.
```

Ну что ж, значение для средней величины периода срабатывания можно оценить на «отлично». Но почему такой большой разброс между минимальным и максимальным значениями? Чтобы выяснить это, заглянем в файл протокола `flog.txt`, в котором сохранены данные всех N измерений.

Изучение его содержимого показывает, что значения в диапазоне 1,94–1,966, превышающие заданный период почти на 1 мс, встречаются с завидным постоянством — примерно один раз в 40 мс. Этот факт нетрудно объяснить. Система Windows каждые 40 мс отдает ресурсы процессора примерно на 1 мс системному диспетчеру потоков. Другое крайнее значение, 0,649, встретилось только один раз на 1000 измерений, причем оно идет сразу вслед за значением 1,3. Такое впечатление, что интервал 1,949, принадлежащий указанному выше диапазону, разбрался на две неравные части: 1,3 и 0,649. В то же время подавляющее число измерений находится в диапазоне 0,971–0,999.

Итоговые результаты для этого и других значений `TIME_PERIOD` приведены в табл. 10.5.

Таблица 10.5. Замеры реального интервала срабатывания мультимедийного таймера

TIME_PERIOD, мс	Реальный интервал, мс		
	Минимальный	Максимальный	Средний
1	0,649	1,973	1,000
2	1,177	3,424	2,000

TIME_PERIOD, мс	Реальный интервал, мс		
	Минимальный	Максимальный	Средний
5	4,061	6,289	5,000
10	9,220	11,278	10,000
50	49,165	51,288	50,001
100	99,065	100,589	99,999
1000	999,747	1000,256	1000,001

Теперь обратим внимание на закомментированные инструкции с номерами #1 и #2. Их можно раскомментировать, чтобы выяснить, как влияет вызов функции `timeBeginPeriod(1)` на работу мультимедийного таймера. Проведение повторных испытаний дает удивительный результат. Вызов функции `timeBeginPeriod(1)` *никак не повлиял* на работу мультимедийного таймера! Следовательно, с учетом сказанного выше о влиянии `timeBeginPeriod` на загрузку ресурсов системы, лучше обойтись без вызова этой функции.

Проверим теперь работу мультимедийного таймера в жестких условиях «реального мира». Под жесткими условиями здесь понимаются интенсивные действия пользователя, многократно перемещающего с помощью мыши окно тестового приложения. Напомним, что стандартный таймер повел себя очень плохо в такой ситуации. Например, для заданного интервала 100 мс был получен разброс значений реального интервала срабатывания от 0,52 до 503,38 мс.

Аналогичный эксперимент с приложением `MmTimerTest` дал следующий результат:

```
minT = 99.57, maxT = 100.62, average = 100.00.
```

Таким образом, мультимедийный таймер позволяет создавать программы, функционирующие почти в режиме реального времени. «Почти» — потому что Windows не является операционной системой реального времени, о чем уже говорилось в начале этой главы.

Осталось сделать два заключительных замечания. Когда вы будете применять мультимедийный таймер для решения реальных задач, очевидно, что вы замените код профилировки в теле функции `TimeProc` на некий реальный код. Не забудьте при этом учесть, что если время выполнения этого кода будет превышать период таймера `TIME_PERIOD`, то периодичность вызова функции `TimeProc` будет определяться уже не таймером, а тормозными характеристиками вашего кода.

Прежде чем использовать мультимедийный таймер, хорошо подумайте, действительно ли это вам нужно? В очень многих случаях характеристики стандартного таймера, несмотря на все его недостатки, позволяют решать те проблемы, которые должны быть решены в вашем приложении. Во всех таких случаях предпочтение следует отдавать стандартному таймеру.

11

Библиотеки динамической компоновки DLL

Библиотеки динамической компоновки (dynamic link libraries (DLL)) являются исполняемыми файлами особого формата, которые содержат функции, данные или ресурсы, доступные для других приложений. Как следует из названия, библиотеки DLL загружаются при необходимости либо на стадии загрузки, либо на стадии выполнения приложения.

Очевидно, что в рамках модели «клиент-сервер» DLL исполняет роль сервера, предлагающего дополнительную функциональность вашему приложению. Приложения, использующие данную возможность, являются клиентами DLL.

Особый формат модулей DLL предполагает наличие в них так называемых *разделов импорта и экспорта*. Раздел экспорта указывает те идентификаторы объектов (функций, классов, переменных), доступ к которым разрешен для клиентов.

Подавляющее большинство DLL (за исключением DLL, содержащих только ресурсы) импортирует функции из системных DLL — kernel32.dll, user32.dll, gdi32.dll и других библиотек. Если ваш проект создается в среде Visual Studio, то его опции автоматически включают стандартный набор таких библиотек. Иногда в этот список требуется добавить дополнительные DLL, содержащие функции, необходимые вашему приложению. Например, в проектах, приведенных в главе 8, мы всегда подключали дополнительно библиотеку comctl32.dll, чтобы использовать элементы управления общего пользования.

В этой главе мы рассмотрим как можно создавать и использовать DLL-модули в ваших приложениях.

Применение DLL может дать ряд преимуществ:

- **Расширение функциональности приложения.** DLL можно загружать в адресное пространство процесса на этапе выполнения, что позволит программе, определив, какие действия от нее требуются, подгружать нужный код. Поэтому, разрабатывая приложение, можно предусмотреть расширение его функциональности за счет DLL от других производителей программного обеспечения.
- **Более простое управление проектом.** Обычно большие проекты разбиваются на модули, разрабатываемые разными группами. Использование DLL упрощает отладку, тестирование и сопровождение проекта.
- **Экономия памяти.** Если одну и ту же DLL используют несколько приложений, то в оперативной памяти хранится только один ее экземпляр, доступный этим приложениям. Например, использование DLL-версии библиотеки C/C++

позволяет избежать многократного дублирования в памяти кода таких функций, как `sprintf`, `strlen`, `fopen` и других.

- **Разделение ресурсов.** DLL могут содержать такие ресурсы, как строки, растровые изображения, шаблоны диалоговых окон. Этими ресурсами может воспользоваться любое приложение.
- **Упрощение локализации.** DLL идеально подходит для локализации приложений. Например, программа, содержащая только код без компонентов пользовательского интерфейса, может загружать DLL, реализующую компоненты локализованного интерфейса.
- **Возможность использования разных языков программирования.** Например, пользовательский интерфейс приложения вы можете реализовать на Microsoft Visual Basic, а прикладную логику — на C++. Программа на Visual Basic может загружать DLL, написанные на C++, Фортране и других языках. Однако если из DLL экспортируются классы, это может привести к дополнительным проблемам (например, из-за декорирования имен компилятором), которые вы должны будете решать.

Добавим, что без применения DLL было бы невозможным построение приложений по технологии COM¹.

В то же время, решение о том, нужно ли создавать собственные DLL-библиотеки, должно приниматься с учетом всех плюсов и минусов технологии DLL. К минусам относятся:

- Увеличение количества поставляемых компонент: кроме основного EXE-файла потребуется отслеживать и все необходимые DLL-модули.
- Большее время загрузки приложения (особенно при неявном способе загрузки и большом количестве DLL-модулей).
- Необходимость синхронизации версий библиотек и версий использующих их клиентов. Пожалуй, это самая неприятная проблема в этой технологии.

Очевидно, что при разработке небольших проектов (утилиты, тестовые приложения) вряд ли имеет смысл создавать DLL для реализации отдельных функций.

DLL и адресное пространство процесса

Исполняемый код в DLL не предполагает автономного использования. Содержимое каждого DLL-файла загружается приложением и проецируется на адресное пространство вызывающего процесса. Это достигается либо за счет *неявного связывания* при загрузке, либо за счет *явного связывания* в период выполнения.

Только после этого для вызывающего потока становятся доступными функции или другие ресурсы библиотеки. В свою очередь, DLL получает доступ ко всем ресурсам потока. Когда поток вызывает из DLL какую-либо функцию, та считывает свои параметры из стека потока и размещает в нем же собственные локальные переменные. Кроме того, любые созданные кодом DLL объекты принадлежат вызывающему потоку или процессу — DLL ничем не владеет.

¹ Component Object Model (COM) — спецификация для создания приложений с компонентной архитектурой.

Другое приложение также может воспользоваться уже загруженной DLL, связав ее с адресным пространством своего процесса.

Важно понимать, что процесс, загрузивший DLL, получает собственную копию глобальных данных, используемых этой библиотекой. Это защищает DLL от ошибок приложений, а процессы, использующие DLL, от взаимного влияния друг на друга.

В тех случаях, когда реализация DLL требует работы с динамической памятью, будьте особенно внимательны к корректному освобождению ресурсов. Например, если DLL-функция вызывает `VirtualAlloc`, система резервирует область в адресном пространстве того процесса, которому принадлежит поток, обратившийся к DLL-функции. Предположим, что DLL выгружается из адресного пространства процесса. В этом случае автоматического освобождения зарезервированной области не произойдет, так как система не фиксирует того, что она зарезервирована DLL-функцией. Считается, что эта область принадлежит процессу и освободится только если поток процесса вызовет `VirtualFree` или завершится сам процесс.

Чтобы отслеживать подобные ситуации, DLL должна содержать особую функцию — `DllMain`, которая рассматривается ниже.

Создание собственной DLL

Во многих случаях создать библиотеку DLL проще, чем приложение, потому что она является лишь набором автономных функций, пригодных для использования любой программой или другой DLL. Зачастую в DLL нет кода, предназначенного для создания окон или для обработки сообщений.

Рассмотрим создание DLL на примере проектирования библиотеки `MyLib.dll`, содержащей единственную функцию

```
void Print(LPCTSTR text);
```

которая является оболочкой для вызова функции `MessageBox`. Сомнительно, что эта библиотека найдет какое-либо практическое применение, но для объяснения рассматриваемой технологии она вполне подходит.

Однако если функция `Print` будет объявлена с прототипом записанным выше, то поток, загрузивший библиотеку `MyLib.dll`, не сможет вызвать данную функцию, так как она окажется для него невидимой. Чтобы «открыть» какие-то из своих объектов для внешнего мира, DLL должна объявить их *экспортируемыми* с помощью модификатора `__declspec(dllexport)`. Кроме этого, в объявление функции следует добавить модификатор `extern "C"`, запрещающий компилятору C++ осуществлять так называемое декорирование имен.

ПРИМЕЧАНИЕ

Декорирование (*mangling*) имен — специфическое явление, присущее компиляторам языка C++. Дело в том, что компилятор C++ всегда добавляет к имени функции сокращенный список формальных параметров и тип возвращаемого значения. Компилятор делает это, чтобы решить проблему идентификации перегруженных функций и правильно связать вызовы функций в программе.

Таким образом, если вы забудете добавить модификатор `extern "C"`, то в откомпилированном модуле `MyLib.dll` имя нашей функции `Print` превратится в нечто похожее на `?Print@@YAXPBD@Z`. В этом случае при явном связывании DLL с клиентским приложением возникнут проблемы.

В листинге 11.1 приведен текст проекта для создания библиотеки MyLib.dll.

Листинг 11.1. Проект MyLib

```
//////////  
// MyLib.h - заголовочный файл DLL-модуля  
#ifndef _MYLIB_H  
#define _MYLIB_H  
  
#include <windows.h>  
  
#define EXPORT extern "C" __declspec(dllexport)  
  
// Здесь определяются прототипы экспортируемых функций  
EXPORT void Print(LPCTSTR text);  
  
#endif // _MYLIB_H  
//////////  
// MyLib.cpp - исходный файл DLL-модуля  
#include <stdio.h>  
#include "MyLib.h"  
int count = 0;  
  
EXPORT void Print(LPCTSTR text)  
{  
    count++;  
    char title[40];  
    sprintf(title, "MyLib: Print: вызов %d", count);  
    MessageBox(NULL, text, title, MB_OK);  
}  
//////////
```

В заголовочном файле использованы «стражи включения» — директивы условной компиляции `#ifndef` и `#endif`. Благодаря им исключается повторное включение этого файла в одной единице трансляции, что важно при построении многофайловых проектов.

Директивой

```
#define EXPORT extern "C" __declspec(dllexport)
```

определяется макрос `EXPORT`, используемый в дальнейшем коде в качестве дополнительного спецификатора заголовков экспортируемых функций.

А теперь — внимание! Чтобы скомпилировать DLL-модуль в среде Visual C++ 6.0, вы должны создать проект типа «Win32 Dynamic-Link Library»¹.

Для этого выберите команду меню **File** ▶ **New** и в появившемся диалоговом окне **New** выберите вкладку **Project**. Последующие действия таковы:

- введите нужное имя проекта, например **MyLib**;
- укажите папку для размещения проекта;
- выберите опцию **Win32 Dynamic-Link Library**;
- нажмите кнопку **OK**.

Далее в появившемся окне **Win32 Dynamic-Link Library – Step 1 of 1** выберите опцию **An empty DLL project** и нажмите кнопку **Finish**.

¹ До сих пор мы работали с проектами типа «Win32 Application».

Когда проект создан, добавьте в его состав файлы MyLib.h и MyLib.cpp и откомпилируйте (F7).

Что обеспечивает среда при построении проекта этого типа? Чтобы получить ответ на этот вопрос, загляните в папку Debug. В ней вы найдете два новых файла:

- ❑ MyLib.dll – DLL-файл, содержащий собственно библиотеку;
- ❑ MyLib.lib – LIB-файл, содержащий список идентификаторов, импортируемых из DLL. Этот LIB-файл, называемый также *библиотекой импорта*, нужен при компоновке любого EXE-модуля, ссылающегося на такие идентификаторы.

Компоновщик также вставляет в конечный DLL-файл *раздел экспорта*, содержащий список экспортимых идентификаторов. В раздел экспорта помещаются также относительные виртуальные адреса этих идентификаторов.

Вызов функций из DLL

Существует три способа загрузки DLL: а) *неявная*, б) *явная*, в) *отложенная*.

Неявная загрузка DLL

Для построения приложения, рассчитанного на неявную загрузку DLL, необходимо иметь:

- ❑ Библиотечный H-файл с описаниями используемых объектов из DLL (прототипы функций, объявления классов и типов). Этот файл используется компилятором.
- ❑ LIB-файл со списком импортируемых идентификаторов. Этот файл нужно добавить в настройки проекта (в список библиотек, используемых компоновщиком).

Компиляция проекта осуществляется обычным образом. Используя объектные модули и LIB-файл, а также учитывая ссылки на импортируемые идентификаторы, компоновщик собирает загрузочный EXE-модуль. В этом модуле компоновщик помещает также *раздел импорта*, где перечисляются имена всех необходимых DLL-модулей. Для каждой DLL в разделе импорта указывается, на какие символьные имена функций и переменных встречаются ссылки в коде исполняемого файла. Эти сведения будет использовать загрузчик операционной системы.

Что же происходит на этапе выполнения клиентского приложения? После запуска EXE-модуля *загрузчик операционной системы* выполняет следующие операции:

- ❑ Создает виртуальное адресное пространство для нового процесса и проецирует на него исполняемый модуль.
- ❑ Анализирует раздел импорта, определяя все необходимые DLL-модули и тоже проецируя их на адресное пространство процесса. Заметим, что DLL может импортировать функции и переменные из другой DLL. А значит, у нее может быть собственный раздел импорта, для которого необходимо повторить те же действия. В результате на инициализацию процесса можетйти довольно длительное время.

После отображения EXE-модуля и всех DLL-модулей на адресное пространство процесса его первичный поток готов к выполнению, и приложение начинает работу.

Еще следует отметить, что загрузчик ищет DLL на дисковых устройствах в следующей последовательности:

- ❑ Каталог, содержащий EXE-файл.
- ❑ Текущий каталог процесса.
- ❑ Системный каталог Windows.
- ❑ Основной каталог Windows.
- ❑ Каталоги, указанные в переменной окружения PATH.

Чтобы проверить, как работает неявная загрузка DLL, создадим клиентское приложение `ImplClient`, использующее функцию `Print` из библиотеки `MyLib.dll`. Текст тестовой программы приведен в листинге 11.2.

Листинг 11.2. Проект `ImplClient`

```
//////////  
// ImplClient.cpp  
#include <windows.h>  
#include "MyLib.h"  
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)  
{  
    Print("Hello!");  
    Print("The experiment was a success!");  
    Print("By!");  
    return 0;  
}  
//////////
```

Создайте проект типа `Win32 Application` и добавьте в его состав файлы `MyLib.h` и `ImplClient.cpp`.

Скопируйте в папку проекта библиотеку импорта `MyLib.lib`, которая была построена в предыдущем проекте.

Добавьте в настройках проекта ссылку на нужную нам библиотеку импорта. Для этого выберите команду меню `Project ▶ Settings` и в появившемся диалоговом окне `Project Settings` выберите вкладку `Link`. Последующие действия таковы:

- ❑ в списке `Category` выберите опцию `General`;
- ❑ в конце списка `Object/library modules` добавьте имя нашей библиотеки — `MyLib.lib`;
- ❑ нажмите кнопку `OK`.

Откомпилируйте проект (`F7`).

Для того чтобы EXE-файл смог выполняться, в одной папке с ним должен находиться используемый DLL-файл. Так как мы будем запускать приложение из среды Visual Studio, скопируйте файл `MyLib.dll` в папку `Debug`.

Запустите клиентское приложение. На экране должно появиться диалоговое окно, показанное на рис. 11.1 слева. Если вы будете продолжать нажимать кнопку `OK`, то должны увидеть еще два окна, (рис. 11.1 по центру и справа). Это означает, что клиентское приложение успешно выполняет свою работу, пользуясь сервисом (функцией `Print`), предоставляемым сервером `MyLib.dll`.

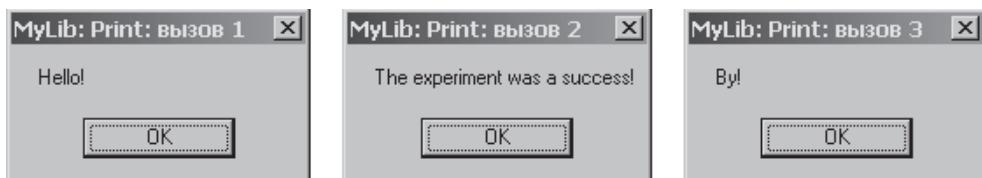


Рис. 11.1. Результаты трех вызовов функции Print из библиотеки MyLib.dll

Завершая эксперимент, давайте посмотрим с помощью отладчика Visual Studio, в какой момент происходит загрузка нашей библиотеки? Информацию, выводимую отладчиком, следует смотреть в окне Output. Нажмите клавишу F10 для запуска отладчика в пошаговом режиме. Появившаяся на левом поле окна редактирования желтая стрелка показывает строчку кода, которая будет выполнена после очередного нажатия клавиши F10. После первого нажатия (когда еще не выполнилось ни одной строчки кода) в окно Output выводится следующая информация:

```
Loaded 'C:\WINNT\system32\ntdll.dll', no matching symbolic information found.
Loaded symbols for 'F:\ProgWinApi\ImplClient\Debug\MyLib.dll'
Loaded 'C:\WINNT\system32\user32.dll', no matching symbolic information found.
Loaded 'C:\WINNT\system32\kernel32.dll', no matching symbolic information found.
Loaded 'C:\WINNT\system32\GDI32.DLL', no matching symbolic information found.
```

Таким образом, мы видим, что библиотека MyLib.dll загружается действительно на этапе загрузки приложения — после загрузки системной библиотеки ntdll.dll.

В заключение отметим, что способ неявной загрузки DLL используется довольно широко из-за своей простоты. Однако этому способу присущи определенные недостатки и ограничения:

- ❑ Все подключенные DLL загружаются всегда, даже если в течение всего сеанса работы программы ни разу не обратится ни к одной из них.
- ❑ Если хотя бы одна из требуемых DLL отсутствует, то загрузка исполняемого файла прекращается, а система выдает сообщение наподобие следующего: «A required DLL file *MyLib.dll* was not found» — даже если отсутствие этой DLL некритично для исполнения программы в нужном для вас режиме.

Явная загрузка DLL

Явная загрузка устраняет отмеченные выше недостатки ценой некоторого усложнения кода. Программисту придется самому заботиться о загрузке DLL и подключении экспортных функций. Зато явная загрузка позволяет подгружать DLL по мере необходимости и дает возможность программе обрабатывать ситуации, возникающие при отсутствии DLL.

В случае явной загрузки процесс работы с DLL происходит в три этапа:

1. Загрузка DLL с помощью функции LoadLibrary (или ее расширенного аналога LoadLibraryEx). В случае успешной загрузки функция возвращает дескриптор hLib типа HMODULE, что позволяет в дальнейшем обращаться к этой DLL.
2. Вызовы функции GetProcAddress для получения указателей на требуемые функции или другие объекты. В качестве первого параметра функция GetProcAddress получает дескриптор hLib, в качестве второго параметра — С-строку с иденти-

фикатором импортируемого объекта. Далее полученный указатель используется клиентом. Например, если это указатель на функцию, то осуществляется вызов нужной функции.

3. Когда загруженная динамическая библиотека больше не нужна, рекомендуется ее освободить, вызвав функцию `FreeLibrary`. Освобождение библиотеки не означает, что операционная система немедленно удалит ее из памяти. Задержка выгрузки предусмотрена на тот случай, когда эта же DLL через некоторое время вновь понадобится какому-то процессу. Но если возникнут проблемы с оперативной памятью, Windows в первую очередь удаляет из памяти освобожденные библиотеки.

Чтобы продемонстрировать, как работает явная загрузка DLL, создадим клиентское приложение `ExplClient`, использующее уже знакомую нам функцию `Print` из библиотеки `MyLib.dll`. Текст тестовой программы приведен в листинге 11.3.

Листинг 11.3. Проект ExplClient

```
///////////  
// ExplClient.cpp  
#include <windows.h>  
  
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)  
{  
    // Дескриптор загружаемой DLL  
    HMODULE hLib;  
    // Загружаем библиотеку  
    hLib = LoadLibrary("MyLib.dll");  
    if (!hLib) {  
        MessageBox(NULL, "Библиотека MyLib.dll не найдена", "Error", MB_OK);  
        return 1;  
    }  
  
    // MYPROC - тип указателя на функцию, совместимый с прототипом функции,  
    // вызываемой из DLL  
    typedef void (*MYPROC) (LPCTSTR str);  
  
    // Объявляем указатель на функцию типа MYPROC  
    MYPROC pMyProc;  
    // Получаем адрес функции Print  
    pMyProc = (MYPROC)(GetProcAddress(hLib, "Print"));  
    if (!pMyProc) {  
        MessageBox(NULL, "В DLL отсутствует функция Print", "Error", MB_OK);  
        return 1;  
    }  
    // Используем функцию Print из DLL  
    pMyProc("Hello!");  
    pMyProc("The experiment was a success!");  
    pMyProc("By!");  
  
    // Выгружаем библиотеку из памяти  
    FreeLibrary(hLib);  
    return 0;  
}
```

Создайте проект типа Win32 Application и добавьте в его состав файл ExplClient.cpp.

Обратите внимание на то, что заголовочный файл MyLib.h в клиентском приложении, использующем явную загрузку, подключать не надо. Также нет нужды указывать библиотеку импорта в настройках проекта.

После компиляции приступим к тестированию программы. Библиотечный модуль MyLib.dll должен находиться в одном каталоге с EXE-модулем. Если программа запускается из среды Visual Studio, то следует скопировать MyLib.dll в папку Debug. Проверка выполнения приложения показывает, что его поведение не отличается от поведения программы ImplClient.exe.

Отложенная загрузка DLL

Этот вариант загрузки появился значительно позже первых двух видов, описанных выше. Например, в среде Visual Studio поддержка данной возможности реализована, начиная с шестой версии.

DLL *отложенной загрузки* (*delay-load DLL*) – это неявно связываемая DLL, которая не загружается до тех пор, пока ваш код не обратится к какому-нибудь экспортируемому из нее идентификатору. Такие DLL могут быть полезны в следующих ситуациях:

- ❑ Если ваше приложение использует несколько DLL, его инициализация может занимать длительное время, требуемое загрузчику для проецирования всех DLL на адресное пространство процесса. DLL отложенной загрузки позволяют решить эту проблему, распределяя загрузку DLL в ходе выполнения приложения.
- ❑ Если приложение предназначено для работы в различных версиях ОС, то часть функций может появиться лишь в поздних версиях ОС и не использоваться в текущей версии. Но если программа не вызывает конкретной функции, то DLL ей не нужна, и она может спокойно продолжать работу. При обращении же к несуществующей функции можно предусмотреть выдачу пользователю соответствующего предупреждения.

Использовать отложенную загрузку DLL достаточно просто, так как большую часть работы берут на себя компилятор и компоновщик, вам необходимо только задать требуемые настройки в проекте клиентского приложения.

Напомним, что для реализации метода неявной загрузки DLL мы добавляли в список библиотек, используемых компоновщиком, требуемую библиотеку импорта (в нашем примере – MyLib.lib). Для реализации метода отложенной загрузки требуется также повторить это действие, но дополнительно в указанный список нужно добавить еще и системную библиотеку импорта delayimp.lib. Кроме этого, требуется добавить в опциях компоновщика флаг /delayload:MyLib.dll.

Перечисленные настройки заставляют компоновщик выполнить следующие операции:

- ❑ внедрить в EXE-модуль специальную функцию _delayLoadHelper;
- ❑ удалить MyLib.dll из раздела импорта исполняемого модуля, чтобы загрузчик операционной системы не пытался выполнить неявную загрузку этой библиотеки на этапе загрузки приложения;
- ❑ добавить в EXE-файл новый раздел *отложенного импорта* со списком функций, импортируемых из MyLib.dll;
- ❑ преобразовать вызовы функций из DLL к вызовам _delayLoadHelper.

На этапе выполнения приложения вызов функции из DLL реализуется обращением к `_delayLoadHelper`. Эта функция, используя информацию из раздела отложенного импорта, вызывает сначала `LoadLibrary`, а затем `GetProcAddress`. Получив адрес DLL-функции, `_delayLoadHelper` делает так, чтобы в дальнейшем эта DLL-функция вызывалась напрямую. Отметим, что каждая функция в DLL настраивается индивидуально при первом ее вызове.

Проверим описанную технологию на примере клиентского приложения `DelayLoadClient`, исходный текст которого, приведенный в листинге 11.4, по существу не отличается от текста программы `ImplClient`:

Листинг 11.4. Проект DelayLoadClient

```
//////////  
// DelayLoadClient.cpp  
#include <windows.h>  
#include "MyLib.h"  
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)  
{  
    Print("Hello!");  
    Print("The experiment was a success!");  
    Print("By!");  
    return 0;  
}  
//////////
```

Создайте проект типа «Win32 Application» и добавьте в его состав файлы `MyLib.h` и `DelayLoadClient.cpp`. Скопируйте в папку проекта библиотеку импорта `MyLib.lib`.

Добавьте в настройки компоновщика ссылки на две библиотеки импорта: `MyLib.lib` и `delayimp.lib`. Как обычно, действуйте через `Project` ▶ `Settings` ▶ `Link` ▶ `Object/library modules`. Затем, оставаясь на вкладке `Link`, добавьте в окне `Project Options` флаг для компоновщика `/delayload:MyLib.dll`.

После компиляции проекта скопируйте в папку `Debug` файл `MyLib.dll` и запустите EXE-файл на выполнение. Результаты будут такими же, как и при запуске приложения `ImplClient` (см. рис. 11.1).

Различия в выполнении этих приложений можно оценить только при помощи отладчика. Нажмите клавишу `F10` для запуска отладчика в пошаговом режиме. После первого нажатия, когда еще не выполнилось ни одной строчки кода, в окне `Output` появляется следующая информация:

```
Loaded 'C:\WINNT\system32\ntdll.dll', no matching symbolic information found.  
Loaded 'C:\WINNT\system32\kernel32.dll', no matching symbolic information found.
```

Мы видим, что библиотека `MyLib.dll` на этапе загрузки приложения не загружается. И только после вызова DLL-функции `Print("Hello!")` — еще два нажатия клавиши `F10` — в окне `Output` появляется:

```
Loaded symbols for 'F:\ProgWinApi\DelayLoadClient\Debug\MyLib.dll'  
Loaded 'C:\WINNT\system32\user32.dll', no matching symbolic information found.  
Loaded 'C:\WINNT\system32\GDI32.DLL', no matching symbolic information found.
```

При последующих вызовах функции `Print` повторной загрузки библиотеки `MyLib.dll` не происходит. Таким образом, все работает так, как и было обещано для метода отложенной загрузки!

Загрузка ресурсов из DLL

Помимо функций, динамические библиотеки могут содержать и ресурсы — строки, пиктограммы, рисунки, и т. д. Хранение ресурсов в DLL особенно удобно при создании приложений с многоязычным интерфейсом. В этом случае, заменив одну DLL на другую, мы заменяем все надписи в программе, скажем, с русского языка на английский, не меняя при этом кода приложения! Аналогично можно менять пиктограммы, внешний вид диалогов и т. д.

Создание DLL, содержащей только ресурсы, осуществляется аналогично созданию DLL с исполняемым кодом, но вместо файла *.cpp в проект добавляется файл *.rc. Технология добавления в проект ресурсов различных типов изложена в главе 5.

Однако следует учесть некоторые тонкости. Предположим, что нам нужно создать библиотеку MyDll.dll, содержащую только ресурсы, при помощи проекта MyDll. Напомним, что после вызова редактора ресурсов соответствующего типа и подготовки самого ресурса (например, таблицы строк) необходимо сохранить созданный ресурс в файле MyDll.rc. При этом редактор ресурсов автоматически создает заголовочный файл resource.h, содержащий все идентификаторы, встречающиеся в файле MyDll.rc. Впоследствии файл resource.h должен быть подключен директивой #include к тем файлам клиентского приложения, в которых используются ресурсы из библиотеки MyDll.dll.

До сих пор нас устраивало имя заголовочного файла resource.h, так как он использовался только в одном проекте. Теперь же, создавая DLL, предназначенную для экспорта в другие проекты, мы должны побеспокоиться об уникальности имени ее заголовочного файла. Поэтому в технологической цепочке подготовки проекта DLL к компиляции должна появиться операция «изменить имя файла resource.h на имя MyDll.h».

Кроме этого, в настройках компоновщика в проекте DLL необходимо добавить флаг /noentry. Это флаг означает, что DLL содержит только ресурсы, и поэтому ей не нужна входная функция DllMain.

Рассмотрим в качестве примера, как можно расширить функциональность программы ImplClient (листинг 11.2), если все строки, которые она использует, вынести в отдельную DLL. Но сначала создадим две библиотеки: MyResEng.dll — с англоязычным текстом, и MyResRus.dll — с русскоязычным текстом.

Библиотека MyResEng.dll. Для ее создания выполните следующие шаги:

1. Создайте проект типа Win32 Dynamic-Link Library с именем MyResEng.
2. Вызовите окно редактора таблицы строк, действуя через Insert ▶ Resource ▶ String Table ▶ New.
3. Введите в таблицу три строки:

Идентификатор	Строка
IDS_1	Hello!
IDS_2	The experiment was a success!
IDS_3	By!

4. Сохраните таблицу строк (File ▶ Save As) в файле MyResEng.rc.
5. Откройте файл MyResEng.rc в текстовом режиме и замените ссылку на заголовочный файл resource.h ссылкой на MyResEng.h.

6. Откройте папку проекта и переименуйте файл `resource.h` на `MyResEng.h`.
7. Добавьте в состав проекта (вкладка `FileView`) файл ресурсов `MyResEng.rc` и заголовочный файл `MyResEng.h`.
8. В настройках компоновщика `Project > Settings > Link > Project Options` добавьте флаг `/noentry`.
9. Откомпилируйте (`F7`).

В папке `Debug` вы найдете библиотечный модуль `MyResEng.dll`. Библиотека импорта `*.lib` в данном случае не нужна, поэтому она и не создается компоновщиком.

Библиотека `MyResRus.dll` создается аналогично при помощи проекта `MyResRus`. Единственное отличие — содержание строк, которые вы введете в таблицу строк:

Идентификатор	Строка
<code>IDS_1</code>	Привет!
<code>IDS_2</code>	Опыт удался!
<code>IDS_3</code>	Пока!

Ну и конечно, соответствующие файлы должны иметь имена `MyResRus.rc` и `MyResRus.h`. После компиляции будет создан модуль `MyResRus.dll`.

Клиентское приложение DllResClient.

1. Создайте проект типа `Win32 Application` с именем `DllResClient`.
2. Добавьте в его состав файл `DllResClient.cpp` с текстом, приведенным в листинге 11.5, а также файлы `MyLib.h`, `MyResEng.h` и `MyResRus.h`.

Листинг 11.5. Проект DllResClient

```
//-----  
// DllResClient.cpp  
//-----  
// Запуск программы из командной строки с одним параметром:  
//     DllResClient.exe Eng    - для англоязычного пользователя  
//     DllResClient.exe Rus    - для русскоязычного пользователя  
//-----  
#include <windows.h>  
#include "MyLib.h"  
#include "MyResEng.h"  
  
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)  
{  
    char cmdLine[100];  
    strcpy(cmdLine, lpCmdLine);  
  
    HMODULE hLib;  
    if (!strcmp(cmdLine, "Eng"))  
        hLib = LoadLibraryEx("MyResEng.dll", 0, LOAD_LIBRARY_AS_DATAFILE);  
    else if (!strcmp(cmdLine, "Rus"))  
        hLib = LoadLibraryEx("MyResRus.dll", 0, LOAD_LIBRARY_AS_DATAFILE);  
    else {  
        MessageBox(NULL, "Отсутствует или неверно задан параметр в командной строке",  
                  "Error", MB_OK);  
        return 0;  
    }  
  
    char buf[100];  
    LoadString(hLib, IDS_1, buf, 99);  
    Print(buf);
```

продолжение ↴

Листинг 11.5. (продолжение)

```
LoadString(hLib, IDS_2, buf, 99);
Print(buf);

LoadString(hLib, IDS_3, buf, 99);
Print(buf);
return 0;
}
////////////////////////////////////////////////////////////////
```

Как следует из комментариев в тексте программы, она должна запускаться в режиме командной строки с параметром, задающим тип интерфейса. Если значением параметра является Eng, программа выводит сообщения на английском языке, если задано значение Rus — на русском.

Обратите внимание на то, что загрузка библиотеки DLL осуществляется вызовом функции `LoadLibraryEx`, хотя можно было бы использовать и `LoadLibrary`. Вызов функции `LoadLibraryEx` здесь предпочтительней, так как благодаря значению ее третьего параметра `LOAD_LIBRARY_AS_DATAFILE` система узнает, что DLL содержит только данные. Это позволяет ускорить загрузку, так как системе достаточно лишь спроцировать DLL на адресное пространство процесса.

После компиляции проекта не забудьте скопировать в папку Debug библиотечные модули `MyResEng.dll`, `MyResRus.dll` и `MyLib.dll`. Запустите программу `DllResClient.exe` на выполнение из командной строки сначала с параметром Eng, а затем — с параметром Rus.

Очевидно, что на рынке программных продуктов приложение `DllResClient.exe`, обладая многоязычным интерфейсом, будет иметь явные преимущества перед приложением `ImplClient!`

Функция входа/выхода

Предположим, что вашей библиотеке динамической компоновки требуется некоторая *инициализация* и *деинициализация*. Например, если в DLL при ее загрузке выделяются какие-то ресурсы, то при ее освобождении эти ресурсы также должны освобождаться.

Особое значение имеет деинициализация: поскольку при отключении DLL от адресного пространства процесса выделенная ею память сама собой не освобождается, а открытые файлы — не закрываются, DLL должна самостоятельно обеспечивать «уборку мусора».

Для решения указанных проблем вы можете включить в состав DLL специальную *функцию точки входа DllMain*. Эта функция вызывается операционной системой в следующих случаях:

- ❑ когда DLL проецируется на адресное пространство процесса (подключение DLL);
- ❑ когда процессом, загрузившим DLL, вызывается новый поток;
- ❑ когда завершается поток, принадлежащий процессу, который связан с DLL;
- ❑ когда процесс освобождает DLL (отключение DLL).

Функция точки входа `DllMain` имеет следующий прототип:

```
BOOL WINAPI DllMain(
    HINSTANCE hinstDLL, // дескриптор DLL-модуля
```

```

    DWORD fdwReason,           // флаг причины вызова функции
    LPVOID lpvReserved        // дополнительная информация
);

```

В момент вызова функция получает информацию от операционной системы через свои параметры.

Первый параметр, `hinstDLL`, принимает значение дескриптора модуля DLL, являющееся, по сути, виртуальным адресом загрузки DLL. Если в библиотеке имеются вызовы функций, которым нужен данный дескриптор, необходимо сохранить значение `hinstDLL` в глобальной переменной.

Второй параметр, `fdwReason`, может принимать одно из следующих значений:

Значение	Интерпретация
<code>DLL_PROCESS_ATTACH</code>	Уведомление о том, что DLL загружена в адресное пространство процесса либо в результате его старта, либо в результате вызова функции <code>LoadLibrary</code> .
<code>DLL_THREAD_ATTACH</code>	Уведомление о том, что текущий процесс создал новый поток. Это уведомление посыпается всем DLL, подключенным к процессу. Вызов <code>DllMain</code> происходит в контексте нового потока.
<code>DLL_THREAD_DETACH</code>	Уведомление о том, что поток корректно завершается. Вызов <code>DllMain</code> происходит в контексте завершающегося потока.
<code>DLL_PROCESS_DETACH</code>	Уведомление о том, что DLL отключается от адресного пространства процесса в результате одного из трех событий: а) неудачное завершение загрузки DLL; б) вызов функции <code>FreeLibrary</code> ; в) завершение процесса.

Если при вызове функции используется первый параметр со значением `DLL_PROCESS_ATTACH`, то по значению третьего параметра можно выяснить, каким способом загружается DLL. При явной загрузке параметр `lpvReserved` равен нулю, а при неявной загрузке принимает ненулевое значение.

Следует отметить, что:

- ❑ Поток, вызвавший `DllMain` со значением `DLL_PROCESS_ATTACH`, не вызывает повторно `DllMain` со значением `DLL_THREAD_ATTACH`.
- ❑ Когда DLL загружается вызовом функции `LoadLibrary`, существующие потоки не вызывают `DllMain` для вновь загруженной библиотеки.
- ❑ Функция `DllMain` не вызывается, если поток или процесс завершаются по причине вызова функции `TerminateThread` или `TerminateProcess`.

Поскольку функция `DllMain` должна обрабатывать все возможные причины своего вызова, ее код обычно выглядит примерно так:

```

BOOL APIENTRY DllMain( HANDLE hModule, DWORD dwReason, LPVOID lpReserved)
{
    switch (dwReason) {
        case DLL_PROCESS_ATTACH:
            // Выполняем действия по инициализации (например, выделение памяти)
            // Если произошла ошибка, то возвращаем FALSE;
            return TRUE;

        case DLL_THREAD_ATTACH:
            // Выполняем действия по инициализации, связанные с новым потоком
            break;

        case DLL_THREAD_DETACH:
    }
}

```

```
// Освобождаем переменные, связанные с потоком
break;

case DLL_PROCESS_DETACH:
    // Выполняем все действия по деинициализации
    break;
}
return TRUE; // этот код возврата игнорируется
}
```

Если DLL проектируется с учетом ее использования в многопоточном приложении, то ей может потребоваться так называемая *локальная память потока* (TLS). Действия по инициализации и деинициализации такой памяти также осуществляются в функции `DllMain`.

Пример использования функции `DllMain` приведен далее в листинге 11.7.

Локальная память потока (TLS)

Мы уже говорили, что процесс, загрузивший DLL, получает собственную копию глобальных данных, используемых этой библиотекой. Это защищает процессы, использующие DLL, от взаимного влияния друг на друга. Но если DLL используется несколькими потоками одного процесса, то глобальные переменные библиотеки разделяются всеми потоками. Это может создавать неприятные проблемы.

Рассмотрим, например, многопоточное приложение `MtClient` (листинг 11.6), использующее библиотеку динамической загрузки `MyLib.dll`.

Листинг 11.6. Проект `MtClient`

```
=====
// MtClient.cpp
#include <windows.h>
#include "MyLib.h"

DWORD WINAPI ThreadFuncA(LPVOID);
DWORD WINAPI ThreadFuncB(LPVOID);

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
=====

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    HANDLE hThreadA = CreateThread(NULL, 0, ThreadFuncA, 0, 0, NULL);
    HANDLE hThreadB = CreateThread(NULL, 0, ThreadFuncB, 0, 0, NULL);

    MessageBox(NULL, "Завершить основной поток?", "WinMain", MB_OK);
    return 0;
}
=====

DWORD WINAPI ThreadFuncA(LPVOID lpv)
{
    Print("Поток А: Привет!");
    Print("Поток А: Эксперимент удался!");
    Print("Поток А: Пока!");
    return 0;
```

```

}

//=====
DWORD WINAPI ThreadFuncB(LPVOID lpv)
{
    Print("Поток Б: Hello!");
    Print("Поток Б: The experiment was a success!");
    Print("Поток Б: By!");
    return 0;
}
///////////

```

В этой программе основной поток (функция `WinMain`) создает два дочерних потока с входными функциями `ThreadFuncA` и `ThreadFuncB`. В каждом дочернем потоке имеются три вызова DLL-функции `Print`.

Создайте проект типа `Win32 Application` с именем `MtClient` и добавьте в его состав файлы `MtClient.cpp` и `MyLib.h`. Скопируйте в папку проекта библиотеку импорта `MyLib.lib`. Добавьте в настройки компоновщика ссылку на библиотеку импорта `MyLib.lib`.

После компиляции проекта скопируйте в папку `Debug` файл `MyLib.dll` и запустите EXE-файл на выполнение. На экране появятся три диалоговых окна, сформированных вызовами функции `MessageBox`:

N	Заголовок окна	Текст в окне
1	MyLib: Print: вызов 1	Поток А: Привет!
2	MyLib: Print: вызов 2	Поток Б: Hello!
3	WinMain	Завершить основной поток?

Однако, скорее всего, вы увидите только третье окно, поскольку первые два окна будут лежать под ним. Так уж устроена функция `MessageBox` — она размещает диалоговое окно всегда в центре экрана. Но, переместив с помощью мыши эти окна в разные позиции, вы получите возможность увидеть одновременно все три окна.

Обратите внимание на заголовок окна, сформированного функцией `Print` из потока Б. В заголовке указано, что это уже второй вызов данной функции, хотя мы точно знаем, что для потока Б он является первым. Если мы продолжим работать с программой, щелкнув на кнопке `OK` в появляющихся диалоговых окнах, то увидим, что нумерация вызовов функции `Print` является сквозной (общей для всех потоков). Причина очевидна — эта нумерация формируется при помощи глобальной переменной `count` в файле `MyLib.cpp`.

А теперь представьте, что мы создаем серверное приложение, которое обрабатывает каждый клиентский запрос, создавая отдельный поток, и в этом потоке вызывается DLL-функция, выполняющая некоторую работу. Но кроме основной работы она подсчитывает количество обращений клиента, чтобы потом выставить банковский счет за обслуживание. Понятно, что глобальная переменная `count` в исходном коде DLL плохо справится с такой задачей.

Возможны и другие ситуации, когда некоторый объект должен быть глобальным для отдельного потока, но при этом не разделяться другими потоками процесса.

Чтобы обеспечить такую возможность, в Win32 API был создан механизм локальной памяти потока (*Thread Local Storage* — TLS). Существуют две разновидности такой памяти: а) *динамическая TLS*, б) *статическая TLS*.

Динамическая TLS

Организация и функционирование динамической TLS поясняются на рис 11.2. Когда стартует любой поток, операционная система выделяет ему в памяти так называемый *массив TLS-слотов*. Элементы этого массива представляют собой четырехбайтные ячейки, которые и называются *слотами*.

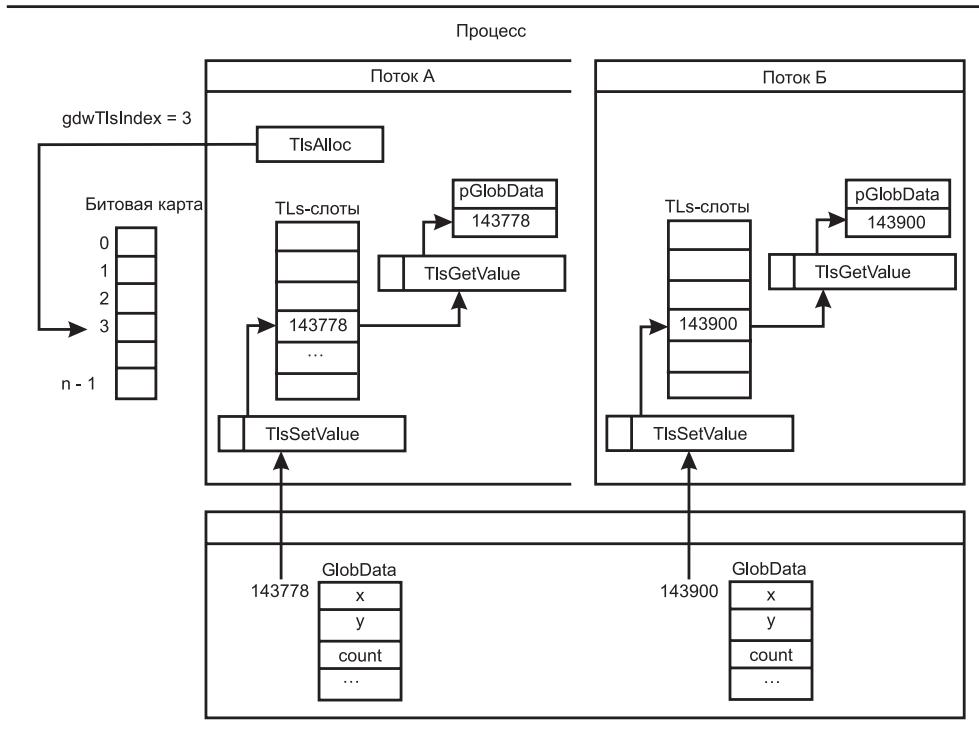


Рис. 11.2. Организация динамической TLS

Стартовавший поток не имеет доступа к массиву TLS-слотов, пока не получит от системы индекс *первого свободного слота*. Дело в том, что индексация всех слотов для всех потоков одного процесса является общей. Поэтому, если стартовавшие ранее потоки уже зарезервировали для себя слоты, например, с индексами 0, 1, 2, то новый поток имеет возможность «приватизировать» слот с индексом 3. Для получения свободного слота вызывается функция `TlsAlloc`:

```
gdwTlsIndex = TlsAlloc();
```

Функция `TlsAlloc` резервирует первый свободный слот и возвращает его индекс, который обычно сохраняется в глобальной переменной `gdwTlsIndex`. В случае неудачи (нет свободных слотов) функция возвращает значение `-1`.

Индекс `gdwTlsIndex` может использоваться в дальнейшем всеми потоками процесса для доступа к зарезервированному слоту. Обратите внимание на важный момент: информация, содержащаяся в «одноименных» слотах разных потоков, является уникальной для каждого потока.

Количество ячеек *n* в массиве TLS-слотов определяется константой `TLS_MINIMUM_AVAILABLE`. Для Windows 95/NT эта константа равна 64, но, начиная с Windows 2000, она уже имеет значение 1088.

Заметим, что функция `TlsAlloc` в своей реализации сканирует глобальную битовую карту, содержащую *n* разрядов. Каждый разряд в этой карте находится в состоянии *free* или *in_use*, указывая, свободен или занят соответствующий TLS-слот. Если найден бит со значением *free*, функция `TlsAlloc` изменяет его значение на *in_use* и возвращает найденный индекс. Перед возвратом управления функция проходит по всем потокам и обнуляет TLS-слоты с выделенным индексом. Данная операция облегчает потокам корректную инициализацию своих слотов.

Непосредственного доступа к слоту у потока нет. Чтобы записать значение в слот, поток вызывает функцию `TlsSetValue`, передавая ей значение индекса `gdwTlsIndex`:

```
TlsSetValue(gdwTlsIndex, pGlobData);
```

Здесь `pGlobData` — адрес в памяти, по которому размещены данные, глобальные в контексте потока, но недоступные для других потоков. Как правило, такие данные содержат более чем одну переменную. В принципе, можно было бы для каждой переменной резервировать отдельный TLS-индекс. Но, с учетом имеющихся ограничений на количество слотов, выделенных процессу, это было бы крайне нерационально.

Поэтому обычно работа с локальной памятью потока организуется следующим образом:

- ❑ Все необходимые переменные собираются в общую структуру, например:

```
struct GlobData {  
    int x;  
    int y;  
    int count;  
    // ...  
    char buffer[256];  
};
```

Затем объявляется указатель на эту структуру:

```
GlobData* pGlobData;
```

- ❑ Поток выделяет память в куче (heap) для сохранения структуры типа `GlobData`. Для этого вызывается одна из функций резервирования памяти, например, `LocalAlloc`:

```
pGlobData = (GlobData*)LocalAlloc(LPTR, sizeof(GlobData));
```

- ❑ Полученный адрес `pGlobData` сохраняется в соответствующем TLS-слоте при помощи функции `TlsSetValue`:

```
TlsSetValue(gdwTlsIndex, pGlobData);
```

- ❑ Когда той или иной функции требуется доступ к данным в TLS, вызывается функция `TlsGetValue`:

```
GlobData* pGlobData = (GlobData*)TlsGetValue(gdwTlsIndex);
```

Полученный указатель `pGlobData` позволяет использовать любое поле структуры `GlobData` как для чтения, так и для записи.

- ❑ Перед завершением работы приложения использующийся слот должен быть освобожден. Для этой операции вызывается функция `TlsFree`, принимающая

в качестве параметра индекс `gdwTlsIndex`. Результатом выполнения `TlsFree` является установка соответствующего разряда битовой карты в состояние *free*.

TLS может применяться в любом многопоточном приложении. Особенно актуально использование механизма локальной памяти потока при разработке библиотек динамической компоновки. Ведь DLL могут загружаться любыми приложениями, в том числе и многопоточными.

Покажем использование описанной технологии на примере разработки библиотеки **MyMtLib**, которая имеет такую же функциональность, как и **MyLib**, но адаптирована для работы с многопоточными приложениями.

Напомним, что библиотека **MyLib** предоставляет своим клиентам единственную функцию

```
void Print(LPCTSTR text);
```

которая выводит на экран окно сообщений с текстом `text` и заголовком вида:

```
MyLib: Print: вызов N
```

где `N` — целое число, обозначающее номер вызова функции `Print`.

Мы уже выяснили, что для того, чтобы нумерация вызовов функции `Print` относилась бы именно к тому потоку, из которого она вызывается, необходимо поместить глобальную переменную `count` в локальную память потока.

Другим недостатком интерфейса библиотеки **MyLib** является отсутствие возможности управлять позицией вывода окна сообщений на экране. Функция `MessageBox`, использованная в реализации функции `Print`, всегда выводит это окно в центре экрана. А это неудобно, когда несколько потоков одновременно вызывают `Print`: диалоговые окна «складываются» в стопку — каждое следующее поверх предыдущего.

Очевидный путь решения этой проблемы — отказаться от использования функции `MessageBox` и реализовать окно сообщений «врукопашную», при помощи модального диалогового окна. Однако такой путь порождает новые проблемы. Необходимо где-то сохранять:

- ❑ строку `text`, передаваемую через параметр функции `Print`, чтобы потом использовать эту строку в диалоговой процедуре;
- ❑ строку `title` с заголовком диалогового окна, также используемую в диалоговой процедуре;
- ❑ значения координат `x`, `y`, определяющих позицию вывода диалогового окна.

И опять единственным возможным корректным решением является хранение всех этих данных в локальной памяти потока.

Все эти соображения учтены в проекте **MyMtLib** (листинг 11.7).

Кроме функции `Print` библиотека **MyMtLib** содержит: а) функцию `InitMyMtLib`, позволяющую настроить позицию вывода окна сообщений; б) функцию `PrintBoxDlgProc`, исполняющую роль диалоговой процедуры; в) функцию точки входа `DllMain`, в которую помещены действия по инициализации/денициализации локальной памяти потока.

Для того чтобы функционирование библиотеки **MyMtLib** стало более наглядным, в ее коде сделаны вставки функции `TRACE`, обеспечивающей вывод отладочной информации в окно `Output`. Эта функция определена в файле `KWndEx.cpp` (см. листинг 8.4 в главе 8).

Листинг 11.7. Проект MyMtLib

```

///////////////////////////////////////////////////////////////////
// MyMtLib.h
#include <windows.h>
#define EXPORT extern "C" __declspec(dllexport)
EXPORT void Print(LPCTSTR text);
EXPORT void InitMyMtLib(int x, int y);
///////////////////////////////////////////////////////////////////
// MyMtLib.cpp
#include <stdio.h>
#include "MyMtLib.h"
#include "KWindEx.h"
#include "resource.h"

struct GlobData {
    int x;
    int y;
    int count;
    char title[100];
    char buffer[256];
};

HINSTANCE hInstDll;
DWORD gdwTlsIndex;

BOOL CALLBACK PrintBoxDlgProc(HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM lParam);
//=====
EXPORT void Print(LPCTSTR text)
{
    // Отладочный вывод
    DWORD threadID = GetCurrentThreadId();
    TRACE("Print: вызов из потока %X, параметр text = '%s'\n",
        threadID, text);

    // Получаем адрес памяти, связанный с индексом gdwTlsIndex
    GlobData* pGlobData = (GlobData*)TlsGetValue(gdwTlsIndex);
    pGlobData->count++;

    sprintf(pGlobData->title, "MyMtLib: Print: вызов %d", pGlobData->count);
    strncpy(pGlobData->buffer, text, 99);

    DialogBox(hInstDll, MAKEINTRESOURCE(IDD_PRINT_BOX), NULL, PrintBoxDlgProc);
}
//=====
EXPORT void InitMyMtLib(int x, int y)
{
    // Получаем адрес памяти, связанный с индексом gdwTlsIndex
    GlobData* pGlobData = (GlobData*)TlsGetValue(gdwTlsIndex);

    pGlobData->x = x;
    pGlobData->y = y;
    pGlobData->count = 0;
}
//=====
BOOL CALLBACK PrintBoxDlgProc(HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    RECT r;

```

продолжение ↗

Листинг 11.7. (продолжение)

```
GlobData* pGlobData = (GlobData*)TlsGetValue(gdwTlsIndex);

switch (uMsg) {

    case WM_INITDIALOG:
        SetWindowText(hDlg, pGlobData->title);
        SetWindowText(GetDlgItem(hDlg, IDC_EDIT1), pGlobData->buffer);

        GetWindowRect(hDlg, &r);
        MoveWindow(hDlg, pGlobData->x, pGlobData->y,
                   r.right - r.left, r.bottom - r.top, TRUE);
        return TRUE;

    case WM_COMMAND:
        switch (LOWORD(wParam)) {
            case IDOK:
                EndDialog(hDlg, 0);
                return TRUE;
            }
            break;
        }
        return FALSE;
    }

//=====
BOOL APIENTRY DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
    GlobData* pGlobData;
    hInstDll = hinstDLL;
    DWORD threadID;

    switch (fdwReason) {

        case DLL_PROCESS_ATTACH:
            // Отладочный вывод
            TRACE("DllMain: подключение DLL, hinstDLL = %X\n", hinstDLL);

            // Размещаем TLS-индекс
            if ((gdwTlsIndex = TlsAlloc()) == 0xFFFFFFFF)
                return FALSE;

            TRACE("DllMain: получен индекс gdwTlsIndex = %d первого свободного слота\n",
                  gdwTlsIndex);
            break;

        case DLL_THREAD_ATTACH: // появился новый поток
            // Отладочный вывод
            threadID = GetCurrentThreadId();
            TRACE("DllMain: создание процессом нового потока, threadID = %X\n", threadID);

            if (TlsGetValue(gdwTlsIndex) == NULL) // если память в TLS еще не выделена
            {
                // Выделяем память в куче
                pGlobData = (GlobData*)LocalAlloc(LPTR, sizeof(GlobData));
            }
    }
}
```

```

// Связываем TLS-индекс потока с адресом памяти pGlobData
TlsSetValue(gdwTlsIndex, pGlobData);
// Отладочный вывод
TRACE("DllMain: выделена память в куче с адресом %X\n", pGlobData);
TRACE("DllMain: адрес %X записан в слот с индексом %d\n", pGlobData,
gdwTlsIndex);
}

break;

case DLL_THREAD_DETACH:
// Отладочный вывод
threadID = GetCurrentThreadId();
TRACE("DllMain: завершение потока, threadID = %X\n", threadID);

// Освобождаем размещеннную память для этого потока
pGlobData = (GlobData*)TlsGetValue(gdwTlsIndex);
if (pGlobData != NULL)
    LocalFree((HLOCAL)pGlobData);
break;

case DLL_PROCESS_DETACH:
// Отладочный вывод
TRACE("DllMain: отключение DLL\n");

// Освобождаем TLS-index.
TlsFree(gdwTlsIndex);
break;
}
return TRUE;
}
////////////////////////////////////////////////////////////////

```

Заметим, что при вызове функции `LocalAlloc`, выделяющей блок памяти в куче, ее первый параметр имеет значение `LPTR`. Константа `LPTR` — это флаг, включающий в себя комбинацию двух других флагов: `LMEM_FIXED` и `LMEM_ZEROINIT`. Значение `LMEM_FIXED` побуждает функцию выделить фиксированный (неперемещаемый) блок памяти в куче и вернуть указатель на начало этого блока. Значение `LMEM_ZEROINIT` обеспечивает инициализацию содержимого выделенного блока памяти нулями.

Создайте проект типа `Win32 Dynamic-Link Library` с именем `MyMtLib` и добавьте в его состав файлы `MyMtLib.h`, `MyMtLib.cpp`, `KWndEx.h` и `KWndEx.cpp`.

С помощью редактора ресурсов добавьте в проект ресурс диалогового окна. В свойствах диалогового окна установите `ID` равным `IDD_PRINT_BOX` и снимите флашок `System menu`. Уменьшите высоту окна примерно на 1/3.

Удалите кнопку `Cancel`, а кнопку `OK` разместите по центру у нижней границы окна.

В центре окна поместите элемент управления `Edit box`, установив для него следующие свойства: выравнивание текста — по центру, флажок `Border` — снят, флажки `Multiline`, `AutoHScroll`, `Read only` — установлены.

После компиляции проекта в папке `Debug` должны появиться файлы `MyMtLib.dll` и `MyMtLib.lib`.

Теперь подготовим клиентское приложение `MtTlsClient` (листинг 11.8) для испытания новой библиотеки.

Листинг 11.8. Проект MtTlsClient

```
//////////  
// MtTlsClient.cpp  
#include <windows.h>  
#include "MyMtLib.h"  
DWORD WINAPI ThreadFuncA(LPVOID);  
DWORD WINAPI ThreadFuncB(LPVOID);  
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);  
=====  
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)  
{  
    HANDLE hThreadA = CreateThread(NULL, 0, ThreadFuncA, 0, 0, NULL);  
    HANDLE hThreadB = CreateThread(NULL, 0, ThreadFuncB, 0, 0, NULL);  
    MessageBox(NULL, "Завершить основной поток?", "WinMain", MB_OK);  
    return 0;  
}  
=====  
DWORD WINAPI ThreadFuncA(LPVOID lpv)  
{  
    InitMyMtLib(400, 350);  
    Print("Поток А: Привет!");  
    Print("Поток А: Эксперимент удался!");  
    Print("Поток А: Пока!");  
    return 0;  
}  
=====  
DWORD WINAPI ThreadFuncB(LPVOID lpv)  
{  
    InitMyMtLib(660, 350);  
    Print("Поток Б: Hello!");  
    Print("Поток Б: The experiment was a success!");  
    Print("Поток Б: By!");  
    return 0;  
}  
//////////
```

Создайте проект типа **Win32 Application** с именем **MtTlsClient** и добавьте в его состав файлы **MtTlsClient.cpp** и **MyMtLib.h**. Скопируйте в папку проекта библиотеку импорта **MyMtLib.lib**. Добавьте в настройки компоновщика ссылку на библиотеку импорта **MyMtLib.lib**. После компиляции проекта скопируйте в папку **Debug** файл **MyMtLib.dll** и запустите программу на выполнение в отладочном режиме (F5). На экране должны появиться три окна, как показано на рис. 11.3.

Поочередно щелкая на кнопке **OK** двух верхних окон, вы можете убедиться, что нумерация вызовов функции **Print** осуществляется для каждого потока отдельно.

А теперь посмотрите на отладочный вывод в окне **Output**. С некоторыми сокращениями он должен выглядеть примерно так:

```
Loaded symbols for 'F:\ProgWinApi\MtTlsClient\Debug\MyMtLib.dll'  
...  
DllMain: подключение DLL, hinstDLL = 10000000  
DllMain: получен индекс gdwTlsIndex = 3 первого свободного слота  
DllMain: создание процессом нового потока, threadID = 8E0  
DllMain: выделена память в куче с адресом 143778  
DllMain: адрес 143778 записан в слот с индексом 3  
Print: вызов из потока 8E0, параметр text = 'Поток А: Привет!'
```

```
DllMain: создание процессом нового потока, threadID = AEC
DllMain: выделена память в куче с адресом 143900
DllMain: адрес 143900 записан в слот с индексом 3
Print: вызов из потока AEC, параметр text = 'Поток Б: Hello!'
...
Print: вызов из потока 8E0, параметр text = 'Поток А: Эксперимент удался!'
Print: вызов из потока AEC, параметр text = 'Поток Б: The experiment was a success!'
Print: вызов из потока 8E0, параметр text = 'Поток А: Пока!'
Print: вызов из потока AEC, параметр text = 'Поток Б: By!'
DllMain: завершение потока, threadID = 8E0
DllMain: завершение потока, threadID = AEC
DllMain: отключение DLL
```

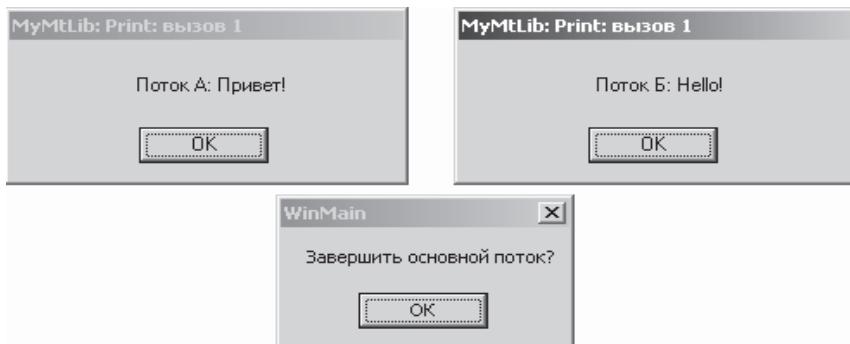


Рис. 11.3. Результат первого вызова функции Print из потоков А и Б

Статическая TLS

Использование этого вида локальной памяти потока на порядок проще. Достаточно при объявлении переменной добавить модификатор `__declspec(thread)`, например:

```
__declspec(thread) int count = 0;
```

чтобы каждый поток работал с собственной копией этой переменной.

Модификатор `__declspec(thread)` поддерживается компилятором Visual C++. Он сообщает компилятору, что соответствующую переменную следует поместить в специальный раздел исполняемого файла, имеющий имя `.tls`.

Функционирование статической TLS происходит в тесном взаимодействии с операционной системой. Загружая приложение в память, система отыскивает в исполняемом файле раздел `.tls` и выделяет блок памяти для хранения всех статических TLS-переменных. При этом учитываются переменные, объявленные как в самом EXE-файле, так и во всех DLL с неявной компоновкой. Каждая ссылка на TLS-переменную переадресуется к участку, расположенному в выделенном блоке памяти. Поэтому компилятору приходится генерировать дополнительный код для ссылок на статические TLS-переменные, что увеличивает размер приложения и замедляет скорость его работы.

Но главным недостатком статической TLS является то, что если вы объявили переменные с модификатором `__declspec(thread)` в коде DLL, то эту библиотеку нельзя будет загружать на этапе выполнения приложения при помощи функции `LoadLibrary`. Если же вы попытаетесь это сделать, то получите ошибку «Access violation». Поэтому при разработке DLL следует использовать только динамическую TLS.

12 Специальные приложения

В этой главе предлагаются примеры решений для двух практических областей. Сначала излагаются основы техники анимации в Windows-программах. Затем рассматривается рисование в реальном времени в приложениях, имитирующих бортовую аппаратуру.

Анимация

Создание анимационных приложений — это весьма обширная тема, и ей посвящено несколько статей в MSDN. В связи с ограничениями на объем данной книги мы рассмотрим только самую простую технику анимации, суть которой заключается в чередовании нескольких шагов. Нужно нарисовать объект, через некоторое время стереть его и изменить позицию объекта. Промежуток времени, в течение которого объект остается в текущей позиции, обычно задается таймером.

Приложение со стандартным таймером

В качестве примера рассмотрим программу «Прыгающий мячик», в которой имитируется полет мяча в прямоугольном боксе от одной стенки к другой. Ударяясь о стенку бокса, мяч отскакивает под соответствующим углом и продолжает движение в новом направлении. Роль «бокса» в этой программе исполняет клиентская область окна приложения. Чтобы сделать пример более интересным, дополним его двумя требованиями: а) мяч должен вращаться во время движения; б) фон, на котором происходит полет мяча, должен изображать некоторую текстуру.

Первая попытка анимации содержится в листинге 12.1. В этой программе демонстрируется применение следующих средств Win32 API:

- Использование узорной кисти (`CreatePatternBrush`) для фона окна (`SetClassLong`) и для операции стирания (`FillRect`) предыдущего изображения мяча.
- Контекст устройства в памяти¹ (`CreateCompatibleDC`) для размещения в нем DDB-растра с изображением мяча (`SelectObject`) и последующего его вывода в контекст дисплея (`BitBlt`).

¹ Или совместимый контекст устройства.

- ❑ Использование региона отсечения (`CreateEllipticRgn`, `SelectClipRgn`) для выделения в прямоугольном растре области с изображением мяча, которая затем копируется при помощи функции `BitBlt`.
- ❑ Мировые преобразования (`SetWorldTransform`) для перемещения и вращения изображения мяча.
- ❑ Функции `SaveDC` и `RestoreDC`, применяемые для сохранения и восстановления текущего состояния контекста устройства.

Листинг 12.1. Проект BounceBall¹

```

///////////////////////////////////////////////////////////////////
// BounceBall.cpp
#include <windows.h>
#include <stdio.h>
#include <math.h>
#include "KWnd.h"
#include "resource.h"

#define Pi 3.14159265

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
void DrawBall(HWND, HDC, HBITMAP, BITMAP, FLOAT, FLOAT, int);
//=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG lpMsg;
    KWnd mainWnd("Bounce ball", hInstance, nCmdShow, WndProc);

    while (GetMessage(&lpMsg, NULL, 0, 0)) {
        TranslateMessage(&lpMsg);
        DispatchMessage(&lpMsg);
    }
    return (lpMsg.wParam);
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    static FLOAT x = 0, y = 0;      // текущая позиция мяча
    static FLOAT dX, dY;          // приращения координат для сдвига на новую позицию
    static int alpha = 0;          // угол, определяющий вращение мяча

    static HDC hDC;               // контекст дисплея
    static HBRUSH hBkBrush;        // узорная кисть
    HBITMAP hBmpBkgr;            // растр для узорной кисти
    static HBITMAP hBmpBall;       // растр с изображением мяча
    static BITMAP bm;              // параметры растра
    RECT rect;                  // прямоугольник клиентской области
    RECT rBall;                 // прямоугольник, ограничивающий изображение мяча

    switch(uMsg)
    {
        case WM_CREATE:
            hDC = GetDC(hWnd);

```

продолжение ↗

¹ Не забудьте добавить к проекту файлы `KWnd.h` и `KWnd.cpp`, текст которых приведен в листинге 1.2.

Листинг 12.1. (продолжение)

```
GetClientRect(hWnd, &rect);
dX = rect.right / 100.;
dY = rect.bottom / 50.;

// Создать таймер (0.1 с)
SetTimer(hWnd, 1, 100, NULL);

hBmpBkgr = LoadBitmap((HINSTANCE)GetWindowLong(hWnd, GWL_HINSTANCE),
    MAKEINTRESOURCE(IDB_STONE));
hBkBrush = CreatePatternBrush(hBmpBkgr);
SetClassLong(hWnd, GCL_HBRBACKGROUND, (LONG)hBkBrush);

hBmpBall = LoadBitmap((HINSTANCE)GetWindowLong(hWnd, GWL_HINSTANCE),
    MAKEINTRESOURCE(IDB_BALL));
GetObject(hBmpBall, sizeof(bm), (LPSTR)&bm);

SetGraphicsMode(hDC, GM_ADVANCED);
break;

case WM_TIMER:
GetClientRect(hWnd, &rect);
// Стираем прежнюю картинку мяча
SetRect(&rBall, (int)x, (int)y, (int)x + bm.bmWidth,
    (int)y + bm.bmHeight);
FillRect(hDC, &rBall, hBkBrush);

// Новая позиция мяча
x += dX;
y += dY;
alpha += 10;
if (alpha > 360) alpha = 0;

// Если мяч достиг края окна, направление его движения изменяется
if(x + bm.bmWidth > rect.right || x < 0)
    dX = -dX;
if(y + bm.bmHeight > rect.bottom || y < 0)
    dY = -dY;

DrawBall(hWnd, hDC, hBmpBall, bm, x, y, alpha);
break;

case WM_DESTROY:
KillTimer(hWnd, 1);
ReleaseDC(hWnd, hDC);
PostQuitMessage(0);
break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}

//=====
void DrawBall(HWND hWnd, HDC hdc, HBITMAP hBmp, BITMAP bm, FLOAT x, FLOAT y,
    int alpha) {
    XFORM xform;
```

```

HRGN hRgn;
// Подготовка к выводу мяча
HDC hBallMemDC = CreateCompatibleDC(hdc);
SelectObject(hBallMemDC, hBmp);
// Создаем регион отсечения
hRgn = CreateEllipticRgn(x, y, x + bm.bmWidth, y + bm.bmHeight);
SelectClipRgn(hdc, hRgn);
// Мировые преобразования для перемещения и вращения мяча
xform.eM11 = (FLOAT) cos(alpha * 2 * Pi / 360); //вращение
xform.eM12 = (FLOAT) sin(alpha * 2 * Pi / 360); //вращение
xform.eM21 = (FLOAT) -sin(alpha * 2 * Pi / 360); //вращение
xform.eM22 = (FLOAT) cos(alpha * 2 * Pi / 360); //вращение
xform.eDx = x + bm.bmWidth / 2.; //смещение по оси x
xform.eDy = y + bm.bmHeight / 2.; //смещение по оси y

// Вывод мяча
SaveDC(hdc);
BOOL ret = SetWorldTransform(hdc, &xform);
BitBlt(hdc, -bm.bmWidth/2, -bm.bmHeight/2,
       bm.bmWidth, bm.bmHeight, hBallMemDC, 0, 0, SRCCOPY);
RestoreDC(hdc, -1);

SelectClipRgn(hdc, NULL);
DeleteObject(hRgn);
DeleteDC(hBallMemDC);
}
/////////////////////////////////////////////////////////////////

```

Обратите внимание на вызов функции `SetGraphicsMode` для переключения контекста устройства в графический режим (`GM_ADVANCED`). Этот режим нужен для использования мировой системы координат и мировых преобразований. В теле функции `DrawBall` мировые преобразования реализуются вызовом функции `SetWorldTransform`, а изменяющиеся значения полей структуры `xform` обеспечивают эффект перемещения и вращения мяча. Эти преобразования должны использоваться *только при выводе изображения мяча*. Поэтому перед вызовом `SetWorldTransform` мы запоминаем текущее состояние контекста устройства с помощью `SaveDC`, а затем восстанавливаем его вызовом `RestoreDC`. Основные события разворачиваются в блоке обработки сообщения `WM_TIMER`. Приведенный код вместе с комментариями, на наш взгляд, не требует дополнительных пояснений.

Чтобы скомпилировать и запустить эту программу на вашем компьютере, необходимо добавить к проекту файл ресурсов, содержащих изображения рисунка фона и мяча. Сначала подготовьте и разместите в папке проекта BMP-файлы с этими изображениями. Для рисунка фона мы воспользовались файлом `Stone.bmp`, который можно скопировать из пакета Microsoft Office¹. Второй файл, `Ball.bmp`, несложно подготовить в любом графическом редакторе, например в MS Paint, сохранив его в 256-цветном формате². Процедура добавления к проекту растровых ресурсов в режиме «Импорт»³ описана в главе 5. Определяя эти ресурсы, свяжите файл

¹ Файл `stone.bmp` обычно находится в папке Styles, путь к которой может выглядеть как `C:\Program Files\Microsoft Office\Office 10\Bitmaps\Styles`.

² Вы можете также взять готовый файл `Ball.bmp` в составе проекта `BounceBall`, если скопируете файлы к данной книге, доступные на сайте издательства www.piter.com.

³ Когда DIB-растр подготовлен с помощью внешнего графического редактора

Stone.bmp с идентификатором IDB_STONE, а файл Ball.bmp — с идентификатором IDB_BALL. В результате в составе вашего проекта должны появиться файлы BounceBall.rc и resource.h.

Когда вы успешно решите все эти проблемы и откомпилируете проект, в окне программы BounceBall появится летающий мячик, как показано на рис. 12.1.

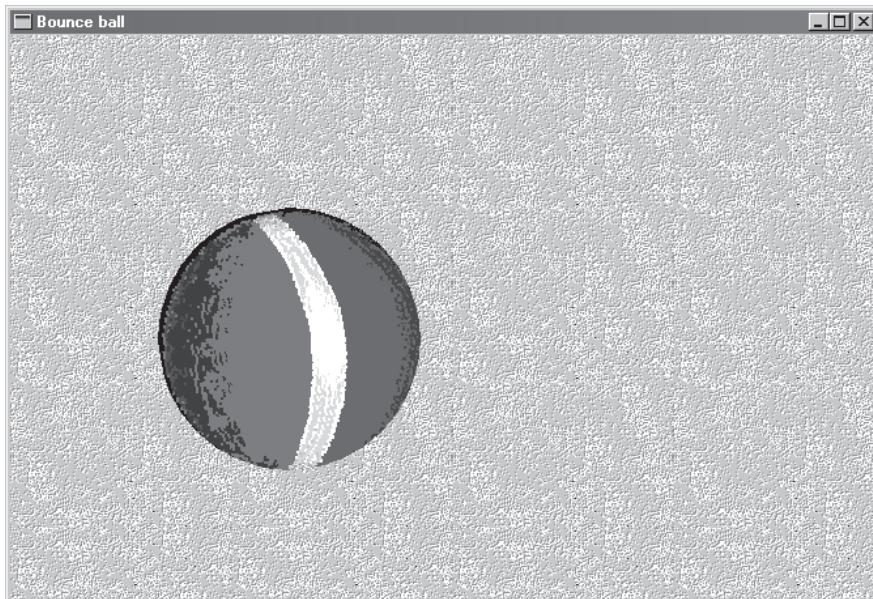


Рис. 12.1. Прыгающий мяч в приложении BounceBall

Все работает замечательно, за исключением одной детали. Дело в том, что вращающийся мячик отвратительно мерцает. Причиной этого является быстрое последовательное выполнение двух операций с контекстом дисплея: стирание прежнего изображения мяча (FillRect) и вывод нового изображения (BitBlt). Вы можете убедиться в этом, закомментировав вызов функции FillRect. После этого изображение перестанет мерцать, но вместо летающего мячика получится что-то вроде червя, прогрызающего тоннель в камне.

Для решения обнаруженной нами проблемы обычно используется прием, известный как *двойная буферизация*.

Двойная буферизация

Неприятное мерцание изображения в анимационном приложении можно устранить, если сформировать очередную фазу картинки в *виртуальном контексте устройства*. Для этого используется контекст в памяти, совместимый с контекстом дисплея. В нашем случае очередная фаза содержит две операции: а) стереть предшествующее изображение мяча; б) нарисовать новое изображение мяча. После этого содержимое совместимого контекста копируется в контекст дисплея.

Продемонстрируем эту технику в листинге 12.2, содержащем модификацию предыдущей программы.

Листинг 12.2. Проект BounceBallAdv

```

////////// //////////////////////////////////////////////////////////////////
// BounceBallAdv.cpp
#include <windows.h>
#include <stdio.h>
#include <math.h>
#include "KWnd.h"
#include "resource.h"

#define Pi 3.14159265

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
void DrawBall(HWND, HDC, HDC, HBITMAP, BITMAP, FLOAT, FLOAT, int);
//=====
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG lpMsg;
    KWnd mainWnd("Bounce ball", hInstance, nCmdShow, WndProc);

    while (GetMessage(&lpMsg, NULL, 0, 0)) {
        TranslateMessage(&lpMsg);
        DispatchMessage(&lpMsg);
    }
    return (lpMsg.wParam);
}

//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam,
    LPARAM lParam)
{
    static FLOAT x = 0, y = 0; // текущая позиция мяча
    static FLOAT dX, dY; // приращения координат для сдвига на новую позицию
    static int alpha = 0; // угол, определяющий вращение мяча

    static HDC hDC; // контекст дисплея
    static HBRUSH hBkBrush; // узорная кисть
    HBITMAP hBmpBkgr; // растр для узорной кисти
    static HBITMAP hBmpBall; // растр с изображением мяча
    static BITMAP bm; // параметры раstra
    RECT rect; // прямоугольник клиентской области
    RECT rBall; // прямоугольник, ограничивающий изображение мяча

    static HDC hMemDcFrame; // совместимый контекст (контекст в памяти)
    static HBITMAP hBmpFrame; // растр для совместимого контекста
    static int count = 0;

    switch(uMsg)
    {
    case WM_CREATE:
        hDC = GetDC(hWnd);
        GetClientRect(hWnd, &rect);
        dX = rect.right / 100.;
        dY = rect.bottom / 50.;
        // Создать таймер (0.1 сек)
}

```

продолжение ➔

Листинг 11.2 (продолжение)

```
SetTimer(hWnd, 1, 100, NULL);

hBmpBkgr = LoadBitmap((HINSTANCE)GetWindowLong(hWnd, GWL_HINSTANCE),
                      MAKEINTRESOURCE(IDB_STONE));
hBkBrush = CreatePatternBrush(hBmpBkgr);
SetClassLong(hWnd, GCL_HBRBACKGROUND, (LONG)hBkBrush);

hBmpBall = LoadBitmap((HINSTANCE)GetWindowLong(hWnd, GWL_HINSTANCE),
                      MAKEINTRESOURCE(IDB_BALL));
GetObject(hBmpBall, sizeof(bm), (LPSTR)&bm);

hMemDcFrame = CreateCompatibleDC(hDC);
hBmpFrame = CreateCompatibleBitmap(hDC, rect.right, rect.bottom);
SelectObject(hMemDcFrame, hBmpFrame);

SetGraphicsMode(hMemDcFrame, GM_ADVANCED);
break;

case WM_SIZE:
GetClientRect(hWnd, &rect);
hBmpFrame = CreateCompatibleBitmap(hDC, rect.right, rect.bottom);
DeleteObject(SelectObject(hMemDcFrame, hBmpFrame));
// Копирование фона в hMemDcFrame
BitBlt(hMemDcFrame, 0, 0, rect.right, rect.bottom, hDC, 0, 0,
        SRCCOPY);
break;

case WM_TIMER:
GetClientRect(hWnd, &rect);
if (!count) { // Копирование фона в hMemDcFrame
    BitBlt(hMemDcFrame, 0, 0, rect.right, rect.bottom, hDC, 0, 0,
            SRCCOPY);
    count++;
}
// Стираем прежнюю картинку мяча
SetRect(&rBall, x, y, x + bm.bmWidth, y + bm.bmHeight);
FillRect(hMemDcFrame, &rBall, hBkBrush);

// Новая позиция мяча
x += dX;
y += dY;
alpha += 10;
if (alpha > 360) alpha = 0;

// Если мяч достиг края окна, направление его движения изменяется
if(x + bm.bmWidth > rect.right || x < 0)
    dX = -dX;
if(y + bm.bmHeight > rect.bottom || y < 0)
    dY = -dY;

DrawBall(hWnd, hDC, hMemDcFrame, hBmpBall, bm, (int)x, (int)y, alpha);
break;

case WM_DESTROY:
KillTimer(hWnd, 1);
ReleaseDC(hWnd, hDC);
```

```

DeleteDC(hMemDcFrame);
PostQuitMessage(0);
break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}

return 0;
}

//=====
void DrawBall(HWND hWnd, HDC hdc, HDC hMemFrameDC, HBITMAP hBmp,
    BITMAP bm, FLOAT x, FLOAT y, int alpha) {
    XFORM xform;
    HRGN hRgn;
    // Подготовка к выводу мяча
    HDC hMemDcBall = CreateCompatibleDC(hdc);
    SelectObject(hMemDcBall, hBmp);

    // Создаем регион отсечения
    hRgn = CreateEllipticRgn(x, y, x + bm.bmWidth, y + bm.bmHeight);
    SelectClipRgn(hMemFrameDC, hRgn);

    // Мировые преобразования для перемещения и вращения мяча
    xform.eM11 = (FLOAT) cos(alpha * 2 * Pi / 360); //вращение
    xform.eM12 = (FLOAT) sin(alpha * 2 * Pi / 360); //вращение
    xform.eM21 = (FLOAT) -sin(alpha * 2 * Pi / 360); //вращение
    xform.eM22 = (FLOAT) cos(alpha * 2 * Pi / 360); //вращение
    xform.eDx = x + bm.bmWidth / 2; //смещение по оси x
    xform.eDy = y + bm.bmHeight / 2; //смещение по оси y

    // Вывод мяча в контекст hMemFrameDC
    SaveDC(hMemFrameDC);
    BOOL ret = SetWorldTransform(hMemFrameDC, &xform);
    BitBlt(hMemFrameDC, -bm.bmWidth/2, -bm.bmHeight/2,
        bm.bmWidth, bm.bmHeight, hMemDcBall, 0, 0, SRCCOPY);
    RestoreDC(hMemFrameDC, -1);
    // Копирование изображения из hMemFrameDC в hdc
    RECT rect;
    GetClientRect(hWnd, &rect);
    BitBlt(hdc, 0, 0, rect.right, rect.bottom, hMemFrameDC, 0, 0, SRCCOPY);

    SelectClipRgn(hMemFrameDC, NULL);
    DeleteObject(hRgn);
    DeleteDC(hMemDcBall);
}
///////////

```

Двойная буферизация в программе реализована на основе контекста в памяти `hMemDcFrame`, совместимого с контекстом дисплея. Виртуальный контекст создается вызовом функции `CreateCompatibleDC`, после чего в него выбирается при помощи функции `SelectObject` растр `hBmpFrame`, также совместимый с контекстом дисплея и имеющий размеры клиентской области окна приложения.

Инициализация контекста `hMemDcFrame` для копирования в него изображения фона осуществляется в блоке обработки сообщения `WM_TIMER`. Чтобы инициализация была однократной, мы используем счетчик `count` и вызываем функцию `BitBlt` только при нулевом значении счетчика. Также нужно позаботиться о новой

инициализации контекста hMemDcFrame в случае изменения размеров окна. Для этого добавлен код обработки сообщения WM_SIZE.

Также обратите внимание на то, что стирание прежней картинки мяча в виртуальном контексте происходит в блоке обработки сообщения WM_TIMER. После вычисления новых координат мяча вызывается функция DrawBall, и ей передается виртуальный контекст в качестве параметра hMemFrameDC. В теле функции DrawBall завершается формирование очередной фазы картинки, когда вызывается функция BitBlt для вывода изображения мяча в виртуальный контекст. Только после этого вся картинка копируется при помощи второго вызова BitBlt из hMemFrameDC в контекст дисплея.

Рекомендуем вам откомпилировать этот проект и убедиться в эффективности метода двойной буферизации.

Рисование в реальном времени

Разработка авиационной бортовой радиотехнической аппаратуры содержит этап тестирования и взаимной стыковки отдельных подсистем. При этом часто вместо отсутствующей подсистемы применяются программные имитаторы. К приложениям, имитирующими реальную аппаратуру, предъявляются довольно жесткие требования. Например, программный имитатор приемника радиолокационной информации, используемый вместо реального индикатора, должен успевать выводить на экран радиолокационную картинку в том темпе, в котором ее выдает передатчик локатора.

Рассмотрим проблемы, возникающие при создании таких программ, на примере разработки имитатора приемника информации от метеорадиолокатора.

Требования к приемнику информации от метеорадиолокатора

Передатчик метеорадиолокатора формирует и выдает по цифровому каналу связи информацию от метеорадиолокатора в соответствии со стандартом «ARINC characteristic 708A»¹. Этот стандарт в числе прочих требований определяет также параметры протокола передачи:

- Информация от метеорадиолокатора передается пакетами размером 1600 бит. Каждый пакет относится к одному лучу сканирования.
- В состав пакета входит:
 - служебная информация;
 - информация об угле сканирования — 12 бит;
 - цветовая информация для 512 точек дальности локационного луча — 1536 бит (по три бита на каждую точку).
- Период выдачи пакетов составляет от 5,00 до 7,82 мс.
- Максимальная угловая скорость вращения антенны радиолокатора — 90 °/с .

¹ ARINC — аббревиатура фирмы «Aeronautical Radio, INC.» (USA).

Приемник информации от метеорадиолокатора должен получать указанные пакеты из цифрового канала связи, распаковывать полученную информацию и отображать радиолокационную картинку на своем индикаторе.

Следует, конечно, более подробно рассмотреть взаимосвязь некоторых параметров протокола передачи.

Код угла сканирования `scanAngle` представлен целым 12-разрядным двоичным числом. Значение `scanAngle` может рассматриваться и как беззнаковое число, находящееся в диапазоне от 0 до 4095 в десятичном исчислении, и как знаковое число в диапазоне от -2048 до +2047. В любом случае получается 4096 значений, которые должны покрывать максимальный сектор обзора радиолокатора, равный 360°. Отсюда вес `digitWeight` одной единицы кода `scanAngle` вычисляется как $360 / 4096 = 0,087890625^\circ$.

Если `PACK_PERIOD` — период выдачи пакетов в миллисекундах, то максимальная возможная скорость сканирования (при условии передачи каждого пакета) ограничена следующей величиной:

```
speedWithPackPeriod = digitWeight / (PACK_PERIOD / 1000.)
```

Подставив указанное выше значение для `digitWeight` и значение 5 для `PACK_PERIOD`, получим значение `speedWithPackPeriod = 17,578°/с`. Но ведь протоколом определена максимальная скорость сканирования $90^\circ/\text{с}$!

Чтобы успевать передавать информацию при такой скорости сканирования, передатчик не отправляет в канал связи каждый очередной пакет, а пропускает несколько пакетов. Введем понятие *коэффициента просеивания*, который определяется следующим выражением:

```
sieveCoeff = ceil(sweepSpeedDue / speedWithPackPeriod);
```

В этом выражении `sweepSpeedDue` — требуемая скорость сканирования.

Например, если `sweepSpeedDue = 90`, то для `PACK_PERIOD = 5` получаем значение коэффициента просеивания `sieveCoeff = 6`. Это значит, что передатчик должен отправлять в канал связи каждый шестой пакет.

Приемник должен знать скорость сканирования `sweepSpeedDue` и период выдачи пакетов. Тогда он может вычислить коэффициент `sieveCoeff`, который используется на приемной стороне уже как *коэффициент размножения* луча развертки. Только в этом случае будет восстановлена исходная радиолокационная картинка (конечно, с теми потерями, которые вызваны пропуском `sieveCoeff-1` лучей).

Разработка модели программного имитатора

Настоящий программный имитатор приемника информации от метеорадиолокатора является довольно сложным приложением. К тому же для его функционирования требуется соответствующее аппаратное обеспечение. Поскольку сейчас нас интересуют только вопросы реализации быстрого рисования, в этой главе рассматривается построение упрощенной модели программного имитатора.

В упрощенной модели имитатора приемника информации от метеорадиолокатора мы абстрагируемся от таких проблем, как прием пакетов из цифрового канала связи и их распаковка для извлечения необходимых параметров.

Модель имитатора будет воспроизводить условную радиолокационную картинку в секторе от -60 до $+60^\circ$, состоящую из четырех «колец» разного цвета.

Период передачи пакетов PACK_PERIOD будет 5 мс. Меню приложения должно позволять выбрать требуемую скорость сканирования из множества значений { 15, 30, 60, 90 } градусов в секунду.

В строке состояния приложения должны отображаться следующие значения: а) скорость сканирования; б) коэффициент размножения лучей; в) расчетное время сканирования всего сектора (в прямом и обратном направлениях); г) фактическое время сканирования всего сектора.

Первая версия модели программного имитатора

Первая версия будет называться ArincReceiverBad. Смысл этого названия станет ясным на этапе тестирования программы.

Создайте новый проект с именем ArincReceiverBad. Скопируйте из папки проекта ToolTip (см. листинг 8.4) в папку проекта ArincReceiverBad файлы KwndEx.h и KwndEx.cpp. Скопируйте из папки проекта KTimer (см. листинг 10.1) файл KTimer.h. Добавьте скопированные файлы в состав проекта. Также добавьте к настройкам проекта на вкладке Link имена библиотек comctl32.lib и winmm.lib.

Добавьте к приложению ресурс меню с идентификатором IDR_MENU1. Главное меню должно содержать пункты с атрибутами, указанными в табл. 12.1.

Таблица 12.1. Пункты главного меню

Имя пункта	Тип пункта	Идентификатор
Скорость сканирования	Подменю	—
Старт	Команда	IDM_START
Стоп	Команда	IDM_STOP

Подменю Скорость сканирования должно содержать пункты с атрибутами, приведенными в табл. 12.2.

Таблица 12.2. Пункты подменю «Скорость сканирования»

Имя пункта	Тип пункта	Идентификатор
15 °/с	Команда	IDM_SPEED_15
30 °/с	Команда	IDM_SPEED_30
60 °/с	Команда	IDM_SPEED_60
90 °/с	Команда	IDM_SPEED_90

Добавьте в состав проекта файл ArincReceiverBad.cpp с текстом, приведенным в листинге 12.3.

Листинг 12.3. Проект ArincReceiverBad

```
///////////
// ArincReceiverBad.cpp
#include <windows.h>
#include <commctrl.h>
#include <Mmsystem.h>
#include <stdio.h>
#include <math.h>

#include "resource.h"
#include "KwndEx.h"
```

```

#include    "KTimer.h"

#define ID_STATUSBAR 201

#define PACK_PERIOD 5      // в миллисекундах

#define LEFT_MARGIN -683 // angle = -60°/c
#define RIGHT_MARGIN 683 // angle = +60°/c
#define Pi 3.141592653589793

HDC g_hDC;
HWND hWndStatusBar;

double sweepSpeedDue; // требуемая скорость развертки (градусов в секунду)
double speedWithPackPeriod; // скорость развертки для периода PACK_PERIOD
int sieveCoeff; // коэффициент просеивания/размножения лучей развертки

double sweepTimeDue; // расчетное время просмотра сектора (с)
double sweepTimeFact; // фактическое время просмотра сектора (с)

int scanAngle; // код угла сканирования в принятом пакете
double digitWeight; // вес единицы кода scanAngle

COLORREF cell[512]; // массив цветов точек дальности
int X0, Y0; // начало системы координат для развертки
double x, y; // текущие координаты вывода точки

KTimer timer;

typedef enum { FORTH, BACK } DIR;
DIR direction = FORTH; // направление сканирования

void UpdateStatusBar();
void CALLBACK TimerFunc(UINT, DWORD, DWORD, DWORD);
void GetPacket(); // Получение пакета из канала связи
void GetNextAngle(); // Вычисление следующего значения кода угла
void SetColorBeamForth(); // Имитация формирования массива cell
// для прямого сканирования
void SetColorBeamBack(); // Имитация формирования массива cell
// для обратного сканирования
void DrawBeam(); // Вывод луча на экран

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

//=====================================================================
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWndEx mainWnd("ArincReceiverBad", hInstance, nCmdShow, WndProc,
        MAKEINTRESOURCE(IDR_MENU1), 100, 100, 900, 600);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

```

продолжение ↗

Листинг 12.3 (продолжение)

```
//=====
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    static HMENU hMenu; // дескриптор главного меню
    RECT rect, rcSB;
    int aWidths[4];
    static MMRESULT mmTimer;

    switch (uMsg)
    {
        case WM_CREATE:
            hMenu = GetMenu(hWnd);
            SetClassLong(hWnd, GCL_HBRBACKGROUND,
                         (LONG)CreateSolidBrush(RGB(100, 100, 100)));

            sweepSpeedDue = 90;
            CheckMenuItem(GetSubMenu(hMenu, 0), IDM_SPEED_15,
                          IDM_SPEED_90, IDM_SPEED_90, MF_BYCOMMAND);

            // Создание строки состояния
            hwndStatusBar = CreateStatusWindow(WS_CHILD | WS_VISIBLE, "", hWnd,
                                               ID_STATUSBAR);
            aWidths [0] = 205;
            aWidths [1] = 355;
            aWidths [2] = 610;
            aWidths [3] = -1;
            SendMessage(hwndStatusBar, SB_SETPARTS, 4, (LPARAM)aWidths);
            SendMessage(hwndStatusBar, SB_SIMPLE, FALSE, 0);

            digitWeight = 360.0 / 4096;
            speedWithPackPeriod = digitWeight / (PACK_PERIOD / 1000.);
            sieveCoeff = ceil(sweepSpeedDue / speedWithPackPeriod);

            UpdateStatusBar();

            GetClientRect(hWnd, &rect);
            GetWindowRect(hwndStatusBar, &rcSB);
            X0 = rect.right / 2;
            Y0 = rect.bottom - (rcSB.bottom - rcSB.top);

            // Контекст устройства для рисования развертки
            g_hDC = GetDC(hWnd);
            SetColorBeamForth();
            break;

        case WM_SIZE:
            SendMessage (hwndStatusBar, WM_SIZE, wParam, lParam);
            break;

        case WM_COMMAND:
            switch (LOWORD(wParam))
            {
                case IDM_SPEED_15:
                    sweepSpeedDue = 15;
                    CheckMenuItem(GetSubMenu(hMenu, 0), IDM_SPEED_15,
                                  IDM_SPEED_90, LOWORD(wParam), MF_BYCOMMAND);
                    break;
            }
    }
}
```

```

case IDM_SPEED_30:
    sweepSpeedDue = 30;
    CheckMenuItem(GetSubMenu(hMenu, 0), IDM_SPEED_15,
                  IDM_SPEED_90, LOWORD(wParam), MF_BYCOMMAND);
    break;
case IDM_SPEED_60:
    sweepSpeedDue = 60;
    CheckMenuItem(GetSubMenu(hMenu, 0), IDM_SPEED_15,
                  IDM_SPEED_90, LOWORD(wParam), MF_BYCOMMAND);
    break;
case IDM_SPEED_90:
    sweepSpeedDue = 90;
    CheckMenuItem(GetSubMenu(hMenu, 0), IDM_SPEED_15,
                  IDM_SPEED_90, LOWORD(wParam), MF_BYCOMMAND);
    break;

case IDM_START:
    timer.Start();
    mmTimer = timeSetEvent(PACK_PERIOD, 0, TimerFunc,
                           reinterpret_cast<DWORD>(hWnd), TIME_PERIODIC);
    if (!mmTimer)
        MessageBox(hWnd, "Error of timeSetEvent!", NULL,
                  MB_OK | MB_ICONSTOP);
    break;

case IDM_STOP:
    timeKillEvent(mmTimer);
    break;
}

sieveCoeff = ceil(sweepSpeedDue / speedWithPackPeriod);
UpdateStatusBar();
break;

case WM_DESTROY:
    ReleaseDC(hWnd, g_hDC);
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}

//=====
void UpdateStatusBar()
{
    char text[100];
    sprintf(text, "Скорость сканирования: %.0f °/с", sweepSpeedDue);
    SendMessage(hwndStatusBar, SB_SETTEXT, 0, (LPARAM)text);

    sprintf(text, "Коэф. размнож. лучей: %d", sieveCoeff);
    SendMessage(hwndStatusBar, SB_SETTEXT, 1, (LPARAM)text);

    sweepTimeDue = 2.*((RIGHT_MARGIN-LEFT_MARGIN) *
                      (PACK_PERIOD/1000.) / sieveCoeff;
    sprintf(text, "Расчетное время просмотра сектора: %.2f с",
           sweepTimeDue);
}

```

продолжение ↗

Листинг 12.3 (продолжение)

```
SendMessage(hwndStatusBar, SB_SETTEXT, 2, (LPARAM)text);

sprintf(text, "Фактическое время просмотра сектора: %.2f с",
       sweepTimeFact);
SendMessage(hwndStatusBar, SB_SETTEXT, 3, (LPARAM)text);
}

//=====
void CALLBACK TimerFunc(UINT wTimerID, UINT msg, DWORD dwUser, DWORD dw1,
                       DWORD dw2)
{
    // Дескриптор окна-владельца
    HWND hwndOwner = reinterpret_cast<HWND>(dwUser);
    // (в модели имитатора не используется, но может понадобиться
    // в реальном имитаторе для отправки сообщений окну-владельцу)

    GetPacket(); // получение пакета из канала связи
    DrawBeam(); // вывод луча из полученного пакета

    // Размножение лучей
    for (int i = 1; i < sieveCoeff; ++i) {
        GetNextAngle();
        DrawBeam();
    }
}

//=====
void GetPacket() {
    GetNextAngle();
}

//=====
void GetNextAngle() {
    char text[100];

    if (direction == FORTH)
        if (++scanAngle >= RIGHT_MARGIN) {
            direction = BACK;
            SetColorBeamBack();

            sweepTimeFact = timer.GetTime() / 1000;
            timer.Start();
            sprintf(text, "Фактическое время просмотра сектора: %.2f с",
                   sweepTimeFact);
            SendMessage(hwndStatusBar, SB_SETTEXT, 3, (LPARAM)text);
        }
    if (direction == BACK)
        if (--scanAngle <= LEFT_MARGIN) {
            direction = FORTH;
            SetColorBeamForth();
        }
}

//=====
void SetColorBeamForth() {
    for (int i = 0; i < 512; ++i) {
```

```

        if (i < 128)      cell[i] = RGB(255, 0, 0);
        else if (i < 256) cell[i] = RGB(0, 255, 0);
        else if (i < 384) cell[i] = RGB(255, 255, 0);
        else               cell[i] = RGB(0, 0, 255);
    }
}

//=====
void SetColorBeamBack() {
    for (int i = 0; i < 512; ++i) {
        if (i < 128)      cell[i] = RGB(0, 0, 255);
        else if (i < 256) cell[i] = RGB(255, 255, 0);
        else if (i < 384) cell[i] = RGB(0, 255, 0);
        else               cell[i] = RGB(255, 0, 0);
    }
}
//=====

void DrawBeam() {
    x = X0;
    y = Y0;

    double angle = scanAngle * digitWeight;
    double C = cos(Pi * (90 - angle) / 180);
    double S = sin(Pi * (90 - angle) / 180);

    for (int k = 0; k < 512; ++k) {
        x += C;   y -= S;
        SetPixel(g_hDC, (int)x, (int)y, cell[k]);
    }
}
///////////

```

Следует обратить внимание на некоторые детали реализации модели.

Имитатор приемника информации от метеорадиолокатора должен опрашивать цифровой канал связи с периодом `PACK_PERIOD`, равным 5 мс. Эта подзадача реализована с помощью мультимедийного таймера, который, как было показано в главе 10, можно успешно использовать в миллисекундном диапазоне. Таймер запускается по команде меню **Старт** и может быть остановлен в любой момент командой **Стоп**.

После запуска таймера операционная система вызывает каждые 5 мс функцию обратного вызова `TimerFunc`. Эта функция и выполняет основную работу имитатора. Через третий параметр функция `TimerFunc` получает дескриптор главного окна приложения. Хотя в данном случае этот дескриптор в теле функции `TimerFunc` не используется, в реальном имитаторе он может пригодиться для отправки сообщений главному окну.

Алгоритм функции `TimerFunc` не требует особых пояснений благодаря продуманной функциональной декомпозиции и выразительным именам вызываемых функций.

А вот функция `GetPacket` нуждается в таких пояснениях. Если бы предметом нашего рассмотрения был реальный имитатор приемника информации от метеорадиолокатора, то «реальная» функция `GetPacket` должна была бы получать очередной пакет из канала связи и обеспечивать его распаковку. В результате такой распаковки эта функция извлекала бы значение кода угла сканирования и формировалась бы массив `cell`. Но так как рассматриваемая программа является всего лишь моделью

реального имитатора, то «модельная» реализация функции `GetPacket` просто вызывает функцию `GetNextAngle`.

Функция `GetNextAngle` обеспечивает формирование следующего значения кода угла сканирования. При каждом вызове этой функции знак приращения кода угла `scanAngle` зависит от направления сканирования. При движении по часовой стрелке (`FORTH`), код угла увеличивается на единицу, при движении против часовой стрелки (`BACK`) — уменьшается на единицу. На границах сектора сканирования переменная `direction` изменяет свое значение. Помимо этого вызывается функция `SetColorBeamForth` или `SetColorBeamBack`, чтобы заполнить массив `cell` цветовыми кодами для точек дальности локационного луча. В результате при движении вперед рисуются кольца красного, зеленого, желтого и синего цветов (перечисление идет от центра к периферии). При движении назад рисуются кольца синего, желтого, зеленого и красного цветов.

В реальном имитаторе вызовы функций `SetColorBeamForth` и `SetColorBeamBack` отсутствуют, поскольку массив `cell` формируется функцией `GetPacket`.

В теле функции `GetNextAngle` также осуществляется хронометраж фактического времени просмотра сектора в обоих направлениях. Измерение отрезка времени между предыдущим и текущим прохождениями правой границы сектора осуществляется с помощью объекта `timer` класса `KTimer`, который рассматривался в главе 10. Измеренный интервал выводится в четвертое поле строки состояния приложения.

Рисование луча развертки реализовано в теле функции `DrawBeam`. Точки дальности локационного луча из массива `cell` выводятся на экран с помощью функции `SetPixel`. Для рисования используется контекст устройства с глобальным дескриптором `g_hDC`. Приложение получает этот контекст один раз при своем старте в блоке обработки сообщения `WM_CREATE` и освобождает его при завершении работы в блоке обработки сообщения `WM_DESTROY`.

После компиляции проекта перейдем к его тестированию.

На этапе тестирования ресурсоемких приложений очень важно наблюдать за степенью загрузки центрального процессора (ЦП). Операционная система содержит удобный инструмент — системное приложение Диспетчер задач `Windows (Task Manager)`, позволяющее вести такое наблюдение.

Для вызова диспетчера задач используйте комбинацию клавиш `Ctrl+Shift+Esc`. В строке состояния окна диспетчера задач отображается текущая загрузка центрального процессора в процентах. Это суммарная загрузка ЦП от всех запущенных процессов. Если переключиться на вкладку `Процессы`, то можно наблюдать за процентом загрузки ЦП от каждого отдельного процесса.

Теперь запустите приложение `ArincReceiverBad`. По умолчанию в приложении установлена скорость сканирования 90 °/с. Выполните команду меню `Старт`. Имитатор начнет воспроизводить «радиолокационную картинку» примерно так, как показано на рис. 12.2.

Здесь приводятся результаты тестирования программы на компьютере с процессором Intel Celeron CPU 2,0 ГГц и операционной системой `Microsoft Windows 2000`.

Диспетчер задач через несколько секунд после начала рисования показывает 100 % загрузки центрального процессора! При этом программа реагирует на команды пользователя с задержкой в несколько десятков секунд. Точно так же парализованы и другие работающие на компьютере приложения, включая и диспетчер задач.

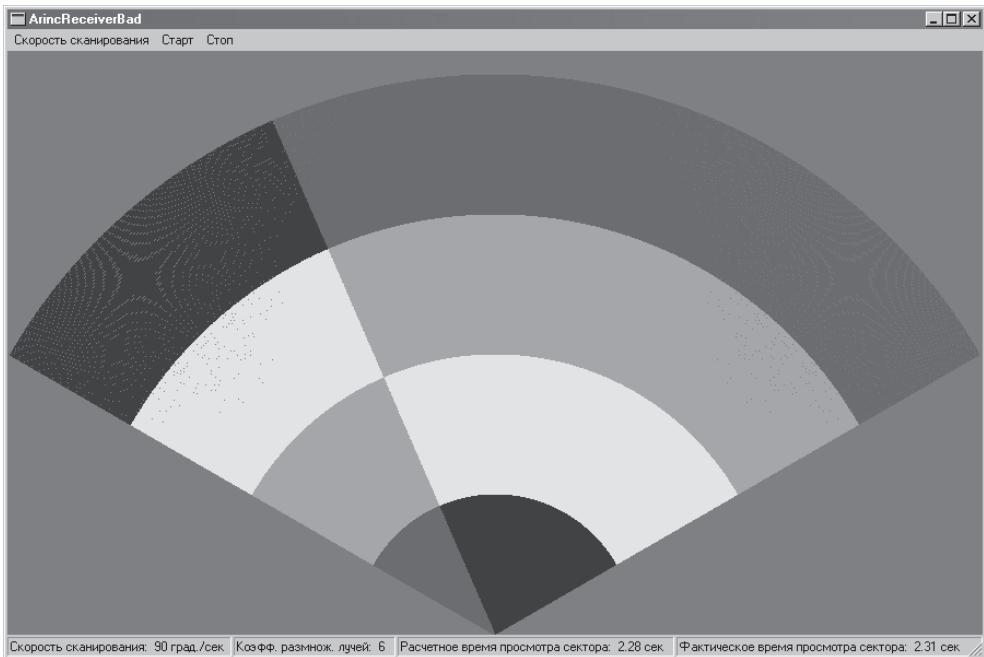


Рис. 12.2. Окно приложения ArincReceiverBad

Очевидно, что для скорости сканирования $90^{\circ}/\text{с}$ наш имитатор не справляется с рисованием в реальном времени. Об этом же свидетельствует и превышение фактического времени просмотра сектора ($2,52 \text{ с}$) над расчетным временем просмотра сектора ($2,28 \text{ с}$).

При тестировании программы со скоростью сканирования $60^{\circ}/\text{с}$ загрузка ЦП падает до 60% , и приложением уже можно пользоваться. Но не будем забывать, что в рассматриваемой модели имитатора многие подзадачи не решаются, а в настоящем имитаторе нагрузка на процессор возрастет.

В чем же причина низких эксплуатационных качеств, которые продемонстрировала первая версия нашей программы? – В использовании функции `SetPixel!`

В главе 2 мы уже говорили, что реализация функции `SetPixel` такова, что процесс вывода пикселя занимает более 1000 тактов работы процессора. Поэтому для быстрого рисования нужны другие технологии.

Одним из возможных решений данной проблемы является применение DIB-секций для рисования. Использование этой технологии мы покажем во второй версии программы.

Вторая версия модели программного имитатора

Основы применения DIB-секций были изложены в главе 3, но без демонстрации на программных примерах. В данном разделе этот пробел будет восполнен.

Напомним, что DIB-секцией называется DIB-растр, который обеспечивает непосредственные чтение и запись как со стороны приложения, так и со стороны GDI. Массив пикселов DIB-секции хранится обычно в виртуальной памяти приложения.

Для создания DIB-секции используется функция `CreateDIBSection`, например:

```
HBITMAP hBmp = CreateDIBSection(hdc, &bmi, DIB_RGB_COLORS, &pBits, NULL, 0);
```

Функции передается адрес информационного блока `bmi`, содержащего характеристики создаваемого раstra. После выполнения функции в аргумент `pBits` записывается указатель на массив пикселов DIB-секции. Быстрое рисование на поверхности DIB-секции основано на прямом доступе к пикселам в массиве `pBits`. Чтобы вычислить адрес для доступа к байтам, представляющим пикセル с координатами (x, y) , нужно знать количество байт, занимаемых каждой строкой изображения. Эта величина вычисляется с помощью следующего выражения:

```
bytePerLine = ((width * bmi.biBitCount + 31) / 32) * 4;
```

Ниже будет рассмотрен класс `KDibSection`, в котором инкапсулированы операции рисования на DIB-секции. Определение и использование класса `KDibSection` демонстрируется во второй версии программного имитатора.

Создайте новый проект с именем `ArincReceiver`. Скопируйте из папки проекта `ArincReceiverBad` (см. листинг 12.3) в папку проекта `ArincReceiver` файлы с расширениями `.h`, `.cpp` и `.rc`, скорректировав их имена заменой подстроки `ArincReceiverBad` на `ArincReceiver`. Добавьте скопированные файлы в состав проекта. Добавьте к настройкам проекта на вкладке `Link` имена библиотек `comctl32.lib` и `winmm.lib`.

Добавьте к проекту файлы `KDibSection.h` и `KDibSection.cpp` с текстом, содержащимся в листинге 12.4. Отредактируйте файл `ArincReceiver.cpp`, приведя его текст в соответствие с листингом 12.4.

Листинг 12.4. Проект ArincReceiver

```
////////////////////////////////////////////////////////////////////////
// KDibSection.h
#include <windows.h>

class KDibSection {
public:
    KDibSection() : pBits(0) {}
    BOOL Create(HDC hdc, int _width, int _height);

    inline void SetPixel(int x, int y, COLORREF color) {
        int ind = 3 * x;
        ((BYTE*)pBits)[y*bytePerLine + ind] = GetBValue(color);
        ((BYTE*)pBits)[y*bytePerLine + ind + 1] = GetGValue(color);
        ((BYTE*)pBits)[y*bytePerLine + ind + 2] = GetRValue(color);
    }

    inline void Draw(HDC hdc) {
        StretchDIBits(hdc, 0, 0, width, height,
                      0, 0, width, height, pBits, &bmi,
                      DIB_RGB_COLORS, SRCCOPY);
    }

private:
    HBITMAP hBmp;
    BITMAPINFO bmi;
    int width;
    int height;
    int bytePerLine;
    PVOID pBits;
};
```

```
///////////////////////////////
// KdibSection.cpp
#include "KdibSection.h"

BOOL KDibSection::Create(HDC hdc, int _width, int _height) {
    width = _width;
    height = _height;

    bmi.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
    bmi.bmiHeader.biWidth = width;
    bmi.bmiHeader.biHeight = -height;
    bmi.bmiHeader.biPlanes = 1;
    bmi.bmiHeader.biBitCount = 24;
    bmi.bmiHeader.biCompression = BI_RGB;

    bytePerLine = ((width * bmi.bmiHeader.biBitCount + 31) / 32) * 4;

    hBmp = CreateDIBSection(hdc, &bmi, DIB_RGB_COLORS, &pBits, NULL, 0);
    return (hBmp != NULL);
}
/////////////////////////////
// ArincReceiver.cpp
#include <windows.h>
#include <commctrl.h>
#include <Mmsystem.h>
#include <stdio.h>
#include <math.h>

#include "resource.h"
#include "KWndEx.h"
#include "KTimer.h"
#include "KDibSection.h"

/* Определения констант и переменных - такие же, как в листинге 12.3 */

KDibSection dibSect;

/* Объявления прототипов функций - такие же, как в листинге 12.3 */

//=====================================================
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;
    KWndEx mainWnd("ArincReceiver", hInstance, nCmdShow, WndProc,
        MAKEINTRESOURCE(IDR_MENU1), 100, 100, 900, 600);

    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}
//=====================================================
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    static HMENU hMenu; // дескриптор главного меню
    RECT rect, rcSB;
```

продолжение ↗

Листинг 12.4 (продолжение)

```
int aWidths[4];
static MMRESULT mmTimer;

switch (uMsg)
{
case WM_CREATE:
    /* Здесь такой же текст, как в листинге 12.3 */
    // Создание DIB-секции
    dibSect.Create(g_hDC, rect.right, Y0);
    break;

case WM_SIZE:
    SendMessage (hwndStatusBar, WM_SIZE, wParam, lParam);
    break;

case WM_COMMAND:
    /* Здесь такой же текст, как в листинге 12.3 */
    break;

case WM_DESTROY:
    ReleaseDC(hWnd, g_hDC);
    PostQuitMessage(0);
    break;

default:
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return 0;
}

//=====
void UpdateStatusBar()
{
    /* Здесь такой же текст, как в листинге 12.3 */
}

//=====
void CALLBACK TimerFunc(UINT wTimerID, UINT msg, DWORD dwUser, DWORD dw1,
    DWORD dw2)
{
    // Дескриптор окна-владельца
    HWND hWndOwner = reinterpret_cast<HWND>(dwUser);
    // (в модели имитатора не используется, но может понадобиться
    // в реальном имитаторе для отправки сообщений окну-владельцу)

    POINT ppt[4]; // массив точек для создания региона отсечения
    // Определяем точки ppt[0] и ppt[3]
    ppt[0].y = ppt[3].y = Y0;
    if (direction == FORTH) { ppt[0].x = X0 - 1; ppt[3].x = X0 + 1; }
    if (direction == BACK) { ppt[0].x = X0 + 1; ppt[3].x = X0 - 1; }

    GetPacket(); // получение пакета из канала связи
    DrawBeam(); // вывод луча из полученного пакета
```

```
// Определяем точку ppt[1]
ppt[1].y = y;
if (direction == FORTH) ppt[1].x = x - 1;
if (direction == BACK)  ppt[1].x = x + 1;

// Размножение лучей
for (int i = 1; i < sieveCoeff; ++i) {
    GetNextAngle();
    DrawBeam();
}

// Определяем точку ppt[2]
ppt[2].y = y;
if (direction == FORTH) ppt[2].x = x + 1;;
if (direction == BACK)  ppt[2].x = x - 1;

// Создание региона отсечения
HRGN hRgn = CreatePolygonRgn(ppt, 4, WINDING);
SelectClipRgn(g_hDC, hRgn);

// Копирование изображения на экран (с учетом региона отсечения)
dibSect.Draw(g_hDC);

DeleteObject(hRgn);
}

//=====
void GetPacket() {
    GetNextAngle();
}

//=====
void GetNextAngle() {
    /* Здесь такой же текст, как в листинге 12.3 */
}

//=====
void SetColorBeamForth() {
    /* Здесь такой же текст, как в листинге 12.3 */
}

//=====
void SetColorBeamBack() {
    /* Здесь такой же текст, как в листинге 12.3 */
}

//=====
void DrawBeam() {
    x = X0;
    y = Y0;
    double angle, C, S;

    angle = scanAngle * digitWeight;
    C = cos(Pi * (90 - angle) / 180);

    S = sin(Pi * (90 - angle) / 180);
    for (int k = 0; k < 512; ++k) {
        x += C;      y -= S;
        // Вывод точки на поверхность DIB-секции
        dibSect.SetPixel((int)x, (int)y, cell[k]);
    }
}
/////////////////
```

Интерфейс класса KDibSection содержится в файле KDibSection.h, а реализация — в файле KDibSection.cpp.

Метод SetPixel предназначен для вывода пикселя с цветом color в точку (x, y) на поверхность DIB-секции. Рисование с помощью SetPixel создает изображение, размещенное в памяти в массиве pBits. Чтобы изображение появилось на экране дисплея, нужно вызвать метод Draw, копирующий изображение из DIB-секции в контекст устройства hdc.

Объект dibSect объявлен в глобальной области видимости:

```
KDibSection    dibSect;
```

DIB-секция создается в блоке обработки сообщения WM_CREATE:

```
dibSect.Create(g_hDC,    rect.right,    Y0);
```

с размерами, соответствующими видимой части клиентской области главного окна.

В теле функции DrawBeam вместо GDI-функции SetPixel вызывается одноименной метод объекта dibSect:

```
dibSect.SetPixel((int)x,    (int)y,    cell[k]);
```

После того как нарисованы основной луч и размножаемые лучи (в теле функции TimerFunc), должен быть вызван метод Draw для переноса изображения на экран.

Хотя метод SetPixel работает очень быстро, общее время рисования включает также и время выполнения метода Draw, которое зависит от площади копируемой поверхности, то есть от параметров раstra width и height. Измерение времени выполнения метода Draw в нашей программе (пока не были приняты специальные меры) показало результат около 8 мс. Очевидно, что такое время нас категорически не устраивает, поскольку период поступления пакетов равен 5 мс. Фактически, при такой реализации программа работала бы еще хуже, чем ArincReceiverBad.

К счастью, есть способ ускорить рисование при помощи регионов отсечения. Действительно, нет никакой необходимости каждые 5 мс копировать на экран весь растр размером width \times height. Достаточно скопировать только ту часть раstra, в которой нарисованы последние sieveCoeff лучей.

Для этого создается многоугольный регион:

```
HRGN hRgn = CreatePolygonRgn(ppt, 4, WINDING);
```

по четырем точкам из массива ppt. Правила определения граничных точек региона отсечения поясняются на рис. 12.3.

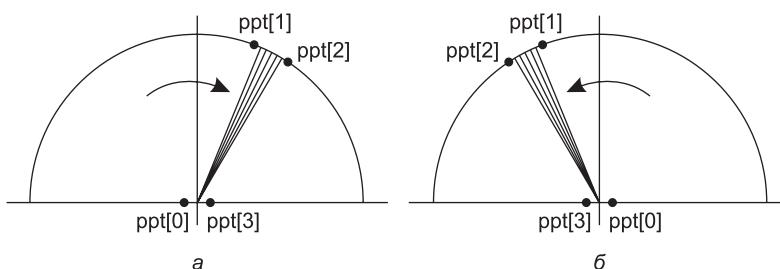


Рис. 12.3. Определение граничных точек региона отсечения: а) сканирование вперед; б) сканирование назад

Определение точек массива `ppt` и создание региона отсечения реализованы в теле функции `TimerFunc`. После выбора региона отсечения в контекст устройства `g_hDC` вызывается метод `Draw`:

```
dibSect.Draw(g_hDC);
```

Профилирование этой инструкции в отладочной версии программы (с помощью таймера – объекта класса `KTimer`) показало, что она выполняется примерно за 0,2 мс.

Тестирование приложения `ArincReceiver`, выполняемого со скоростью сканирования 90°/с, показало, что загрузка центрального процессора составляет не более 5 %!

Таким образом, для рисования в реальном времени использование DIB-секций в сочетании с регионом отсечения может быть неплохим решением.

Напомним, что термин «реальное время» в применении к Windows-приложениям является весьма условным, поскольку Windows не является операционной системой реального времени. Об этом мы уже говорили в главе 10. Поэтому возможны отдельные сбои в функционировании таких программ, как `ArincReceiver`, если Windows вдруг решит, что у него есть дела поважнее. Но так как имитатор `ArincReceiver` не управляет никакими стратегическими объектами, то подобные сбои вполне допустимы.

Приложение 1

Интегрированная среда Visual C++ 6.0

Integrated Development Environment (интегрированная среда разработки), или сокращенно *IDE* — это программный продукт, объединяющий текстовый редактор, компилятор, отладчик и справочную систему.

Мы предполагаем, что пакет Microsoft Visual Studio 6.0, в состав которого входит IDE Microsoft Visual C++ 6.0, уже установлен на вашем компьютере.

Любая программа, создаваемая в среде Visual C++, даже такая простая как «Hello, World!», всегда оформляется как отдельный *проект* (*project*).

Проект — это набор взаимосвязанных исходных файлов (с расширением .cpp), заголовочных файлов (с расширением .h), файла ресурсов (с расширением .rc) и некоторых других файлов, компиляция и компоновка которых позволяют создать исполняемую программу (файл с расширением .exe).

Разработчики Visual Studio предусмотрели сервис для коллективной разработки программных продуктов. Он реализован в виде «рабочей области». *Рабочая область* (*project workspace*) может содержать любое количество различных проектов, сгруппированных вместе для согласованной разработки.

В данном приложении приводятся минимально необходимые сведения для начала работы с интегрированной средой и для построения проекта типа *Win32 Application* — *empty project*. Все примеры программ в книге реализуются в проектах этого типа.

Рабочая область проекта у нас всегда будет содержать только один проект.

Запуск IDE. Типы приложений

Запуск Visual C++ осуществляется при помощи команды меню Пуск ▶ Программы ▶ Microsoft Visual Studio 6.0 ▶ Microsoft Visual C++ 6.0¹. Впрочем, можно просто щелкнуть мышью на соответствующей пиктограмме, если вы позаботились о ее размещении на рабочем столе компьютера.

После запуска Visual C++ появляется главное окно программы, показанное на рис. П1.1. В зависимости от настроек для вашего рабочего стола Visual C++ его вид может несколько отличаться от показанного на рисунке.

¹ На вашем компьютере путь к исполняемой команде меню может быть другим.

На самом деле главное окно Visual C++ принадлежит студии разработчика Microsoft Developer Studio, которая является интегрированной средой, поддерживающей Visual C++, Visual J, MS Fortran Power Station и некоторые другие продукты.

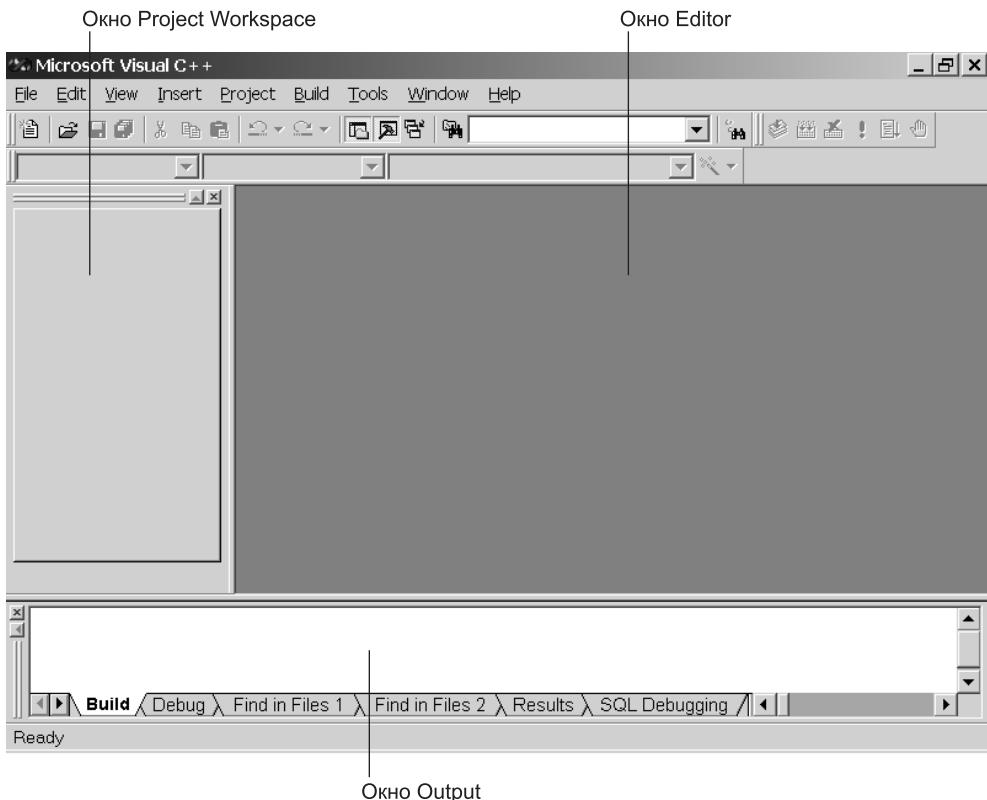


Рис. П1.1. Главное окно (рабочий стол) Visual C++

Рабочий стол Visual C++ обычно содержит три окна.

Окно Project Workspace (*окно рабочей области*) позволяет эффективно управлять проектом при написании и сопровождении больших многофайловых программ. Пока что (см. рис. П1.1) оно закрыто, но после создания нового проекта или загрузки сохраненного ранее проекта одна из вкладок этого окна будет содержать список файлов, входящих в проект.

Окно Editor (*окно редактора*) используется для ввода и проверки исходного кода.

Окно Output (*окно вывода*) служит для вывода сообщений о ходе компиляции, сборки и выполнения программы. В частности, сообщения о возникающих ошибках будут появляться именно в этом окне.

Под заголовком главного окна, как и во всех Windows-приложениях, располагается строка меню. Назначение команд меню и кнопок панелей инструментов мы будем рассматривать по мере необходимости, разбирая основные приемы работы в IDE. Здесь же только заметим, что для кнопок панелей инструментов предусмотрена удобная подсказка. Если пользователь наведет курсор мыши на кнопк-

ку и задержит его там на пару секунд, то всплывет окно подсказки, в котором будет описано предназначение кнопки.

Developer Studio позволяет создавать проекты разных типов, ориентированных на различные сферы применения. В этой книге все программные примеры должны строиться в виде проекта типа **Win32 Application — empty project**.

Создание нового проекта

Для создания нового проекта типа **Win32 Application** нужно выполнить простую последовательность действий:

- Выполните команду меню **File ▶ New...**
- В открывшемся диалоговом окне **New** выберите вкладку **Projects**. На этой вкладке надо выбрать тип **Win32 Application**, после чего указать имя проекта в текстовом поле **Project Name**, например, **HelloFromMsgBox**. Также потребуется ввести имя каталога размещения файлов проекта в текстовом поле **Location**¹. Если указанный каталог отсутствует, то он будет создан автоматически. После этого остается только нажать кнопку **OK**.
- После этого будет запущен так называемый *мастер приложений Application Wizard*, который открывает диалоговое окно **Win32 Application — Step1 of 1** с предложением определиться, какой подтип приложения вы хотите создать. В этом окне надо выбрать опцию **An empty project** и нажать кнопку **Finish**.
- После щелчка будет отображено окно **New Project Information** с параметрами проекта и информацией о каталоге, в котором будет размещен создаваемый проект. В этом окне нужно нажать кнопку **OK**.

После выполненных шагов на экране будет отображено окно, внешний вид которого показан на рис. П1.2.

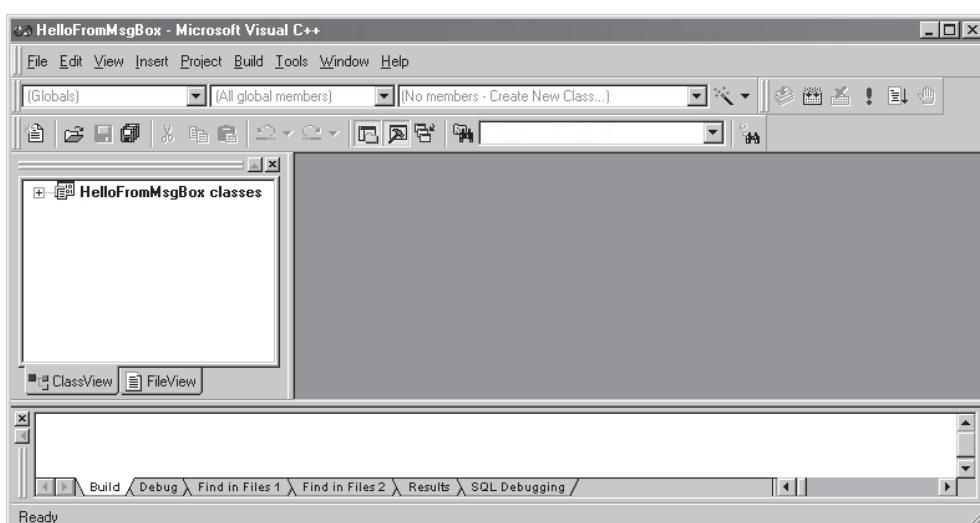


Рис. П1.2. Рабочий стол Visual C++ после создания проекта HelloFromMsgBox

¹ Рекомендуется для размещения проектов выделить специальную папку, например D:\MyProjects.

Прежде чем продолжать работу, свернем временно главное окно Visual C++ и заглянем в папку HelloFromMsgBox, созданную мастером приложений для нашего проекта, а точнее, для нашей рабочей области. Там будут расположены файлы HelloFromMsgBox.dsw, HelloFromMsgBox.dsp, HelloFromMsgBox.opt, HelloFromMsgBox.ncb. Помимо этого там же будет расположена папка Debug или Release, что зависит от конфигурации проекта. Поясним назначение перечисленных файлов:

- HelloFromMsgBox.dsw — файл рабочей области проекта, используемый внутри интегрированной среды разработки. Он содержит всю информацию о проектах, входящих в данную рабочую область.
- HelloFromMsgBox.dsp — проектный файл, используемый для построения отдельного проекта. В ранних версиях Visual C++ этот файл имел расширение .mak.
- HelloFromMsgBox.opt — файл, в котором хранятся опции рабочей области проекта. Благодаря этому файлу при каждом открытии рабочей области проекта все параметры Developer Studio, выбранные во время последнего сеанса работы с данной рабочей областью, будут восстановлены.
- HelloFromMsgBox.ncb — этот служебный файл создается компилятором и содержит информацию, которая используется в инструменте интегрированной среды под названием ClassView. Вкладка ClassView находится в окне Project Workspace и показывает все классы C++, определения которых входят в состав проекта, а также все члены этих классов.
- Debug — это папка, в которую будут помещаться файлы, формируемые компилятором и сборщиком. Из них нас будет интересовать только исполняемый файл, имеющий расширение .exe.

Развернем обратно главное окно Visual C++ с открытой рабочей областью (см. рис. П1.2), чтобы продолжить работу с первой программой. Следует отметить, что в окне Project Workspace появились вкладки Class View и File View.

Перейдите на вкладку File View. Эта вкладка предназначена для просмотра списка файлов проекта. Откроем список HelloFromMsgBox files, щелкнув мышью на значке с изображением плюса. В результате будет отображено дерево списка файлов, содержащее пиктограммы папок Source Files, Header Files, Resource Files. Все папки пусты, так как проект был создан с опцией An empty project.

Добавление к проекту файлов с исходным кодом

В состав проекта можно добавлять как новые файлы, так и уже существующие. Обе эти ситуации надо рассматривать отдельно.

Добавление существующего файла

В этом случае файл с исходным кодом (пусть это будет файл HelloFromMsgBox.cpp) вы уже подготовили ранее. Теперь, чтобы добавить его в состав проекта, выполните простую последовательность действий:

- Скопируйте исходный файл HelloFromMsgBox.cpp в папку рабочей области проекта. В данном случае это будет папка HelloFromMsgBox.
- Вернитесь к списку HelloFromMsgBox files в окне Project Workspace вашего проекта и щелкните на папке Source Files.
- В появившемся контекстном меню выберите команду Add Files to Folder...

- В открывшемся диалоговом окне **Insert Files into Project** выберите нужный файл (**HelloFromMsgBox.cpp**) и нажмите кнопку **OK**.

Добавление нового файла

Добавление в состав проекта нового файла производится несколько иначе:

- Выполните команду меню **File ▶ New...**. В результате этого будет открыто диалоговое окно **New**.
- На вкладке **Files** выберите тип файла (в данном случае это будет **C++ Source File**).
- В текстовом поле **File Name** нужно задать имя файла (в данном случае **HelloFromMsgBox.cpp**). При этом флагок **Add to project** должен быть установлен.
- Нажмите кнопку **OK**.

После этого в окне **Editor** появится окно текстового редактора с заголовком **HelloFromMsgBox.cpp**. Одновременно в окне **Project Workspace** слева от папки **Source Files** появится знак плюса. Если щелкнуть по нему мышью, то папка **Source Files** раскроется и будет отображен список файлов. В настоящий момент в нем будет только один файл **HelloFromMsgBox.cpp**.

ПРИМЕЧАНИЕ

Имеется альтернативный способ добавления к проекту нового файла. Необходимо щелкнуть правой кнопкой мыши на пиктограмме папки **Source Files** в окне **Project Workspace**, а затем во всплывшем контекстном меню выбрать команду **Add Files to Folder**. Появится диалоговое окно **Insert Files into Project** с текстовым полем «**Имя файла**». Необходимо ввести требуемое имя (например, **HelloFromMsgBox.cpp**) и нажать кнопку **OK**.

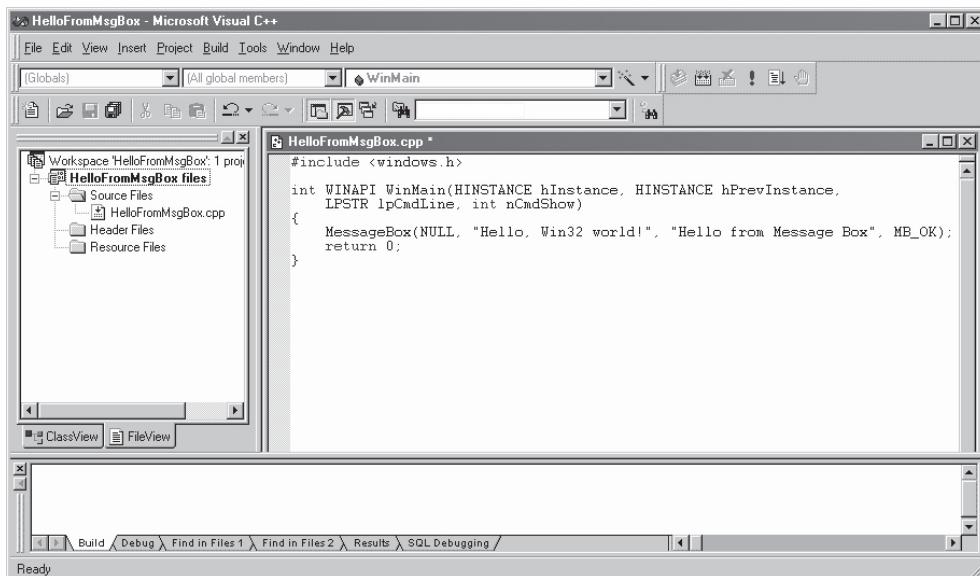


Рис. П1.3. Вид главного окна после ввода исходного текста в файл HelloFromMsgBox.cpp

Теперь займемся исходным текстом программы. Введите в окне текстового редактора, например, такой код:

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MessageBox(NULL, "Hello, Win32 world!", "Hello from Message Box", MB_OK);
    return 0;
}
```

На рис. П1.3 показан вид главного окна Visual C++ после проделанной работы.

Многофайловые проекты

Никаких особых усилий при создании многофайловых проектов вам прилагать не придется. Разработчики Developer Studio уже позаботились об удобстве вашей работы. Просто надо будет многоократно повторить процедуру создания и добавления исходных файлов, описанную выше.

В многофайловых проектах обычно присутствуют и заголовочные файлы. Они создаются/добавляются после щелчка правой кнопкой мыши на пиктограмме папки **Header Files** в окне **Project Workspace**. Затем во всплывшем контекстном меню необходимо выбрать команду **Add Files to Folder**. В результате будет отображено диалоговое окно **Insert Files into Project** с текстовым полем **Имя файла**. После ввода требуемого имени, например, **AnyName.h**, нужно нажать кнопку **OK**.

ПРИМЕЧАНИЕ —

Папки **Source Files** и **Header Files**, пиктограммы которых можно увидеть в окне **Project Workspace**, на самом деле физически не существуют. То есть все файлы помещаются в основную папку рабочей области проекта. Но, согласитесь, такое упорядочение дерева списка файлов в окне **Project Workspace** очень удобно.

Компиляция, компоновка и выполнение проекта

Эти операции могут быть выполнены или при помощи меню **Build** главного окна, или с помощью кнопок панели инструментов. Следует кратко описать основные команды меню **Build**:

- **Compile** — компиляция выбранного файла. Результаты компиляции отображаются в окне **Output**.
- **Build** — компоновка проекта. Компилируются все файлы, в которых произошли изменения с момента последней компоновки. После компиляции происходит сборка всех объектных модулей, включая библиотечные, в результирующий исполняемый файл. Сообщения об ошибках компоновки выводятся в окно **Output**. Если обе фазы компоновки завершились без ошибок, то созданный исполняемый файл с расширением **.exe** может быть запущен.
- **Rebuild All** — то же, что и **Build**, но при выполнении этой команды компилируются все файлы проекта независимо от того, были ли в них изменения.
- **Execute** — выполнение исполняемого файла, созданного в результате компоновки проекта.

Операции **Compile**, **Build** и **Execute** удобнее выполнять при помощи соответствующих кнопок панели инструментов **Build MiniBar** или через «горячие» клавиши. Панель инструментов **Build MiniBar** в увеличенном виде показана на рис. П1.4.

СОВЕТ

Иногда при компоновке многофайлового проекта с командой Build будут отображаться сообщения об ошибках компиляции или сборки, которые вы не можете объяснить. Рекомендуем вам в этом случае обязательно попробовать выполнить команду Rebuild All. Во многих случаях такой прием помогает «починить» ваш проект.

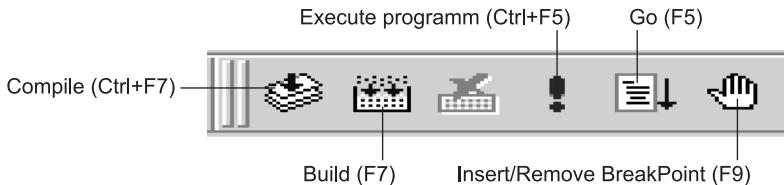


Рис. П1.4. Панель инструментов Build MiniBar

Откомпилируйте проект, щелкнув на кнопке Build (F7). Диагностические сообщения компилятора и сборщика будут отображаться в окне вывода Output. Мы надеемся, что у вас все будет в порядке и последняя строка в окне вывода будет выглядеть следующим образом:

HelloFromMsgBox.exe - 0 error(s), 0 warning(s).

Теперь вы можете запустить приложение на выполнение, нажав кнопку Execute Program (Ctrl+F5).

Кнопка Go (F5) используется для запуска программы в отладочном режиме. Приложение может выполняться под отладчиком, только если оно построено в отладочной конфигурации проекта.

Конфигурация проекта

Visual C++ позволяет строить проект либо в *отладочной конфигурации* (Win32 Debug), либо в *выпускной конфигурации* (Win32 Release). Пока программа не отлажена, лучше работать с проектами в отладочной конфигурации. Обычно она используется по умолчанию. Вы можете проверить, с какой конфигурацией на самом деле идет работа. Для этого надо выполнить команду меню Project ▶ Settings.... В результате будет открыто диалоговое окно Project Settings. Проверьте, какое значение установлено в окне комбинированного списка Settings For:. Если это не Win32 Debug, то нужно выполнить команду меню Build ▶ Set Active Configuration... и выбрать нужное значение.

Как закончить работу над проектом

Когда работа будет завершена, следует выполнить команду меню File ▶ Close Work-space. А можно просто закрыть приложение Visual C++.

Как открыть проект, над которым вы ранее работали

Есть три способа открыть проект:

1. Запустите среду Visual C++. Выберите в меню File пункт Open Workspace... В открывшемся диалоговом окне надо отыскать вашу папку с проектом, а в ней —

файл `ProjectName.dsw`. Теперь можно открыть этот файл, щелкнув по нему мышью.

2. Запустите среду Visual C++. Выберите меню `File` и наведите курсор мыши на пункт `Recent Workspaces`. Если во всплывшем меню со списком последних файлов, с которыми шла работа, вы найдете интересующий вас файл `ProjectName.dsw`, то щелкните на нем.
3. Найдите вашу папку с проектом, а в ней отыщите файл `ProjectName.dsw`, после чего просто щелкните на нем мышью.

Встроенная справочная система

Microsoft Developer Network (MSDN) — это набор онлайновых и оффлайновых служб, предназначенных для оказания помощи разработчику в написании приложений с использованием продуктов и технологий фирмы Microsoft.

Справочная система MSDN (MSDN Library) обычно устанавливается на компьютере вместе со средой Microsoft Visual Studio 6.0. Она содержит обширную информацию по программированию, в том числе и описание структур данных и функций Win32 API. Далее для краткости вместо термина «справочная система MSDN» будем употреблять короткий термин «MSDN».

Если MSDN установлена, то в IDE Visual C++ она доступна через меню `Help` главного окна. Кроме того, очень удобно пользоваться интерактивной справкой. Если вы находитесь в окне `Editor`, то достаточно навести текстовый курсор на интересующий вас оператор или библиотечную функцию C++ и нажать клавишу `F1`. При этом будет вызвана MSDN с отображением нужной вам информации. Если запрошенный термин встречается в разных разделах MSDN, то сначала появится диалоговое окно `Найденные разделы`. В списке разделов надо выбрать тот, в котором упоминается «Visual C++» или «Windows SDK».

Работа с отладчиком

Полное описание возможностей встроенного отладчика Visual C++ и приемов работы с ним может потребовать отдельной книги. Поэтому мы рассмотрим только начальные сведения о работе с отладчиком Visual C++. Проще всего это сделать, написав программу, заведомо содержащую несколько ошибок, а затем посмотреть, как с помощью отладчика можно найти и исправить эти ошибки.

Также мы научимся устанавливать в программе *точки прерывания* и выполнять программу до заданной точки. Когда во время выполнения встречается точка прерывания, программа останавливается, а на экране появляется отлаживаемый код. Это дает возможность детально выяснить, что именно происходит в программе в текущий момент.

Кроме того, программу можно выполнять последовательно, строку за строкой. Такой процесс называется *пошаговым выполнением*. Этот режим позволяет следить за тем, как изменяются значения различных переменных. Иногда он помогает понять, в чем заключается проблема. Если обнаруживается, что переменная принимает неожиданное значение, то это может послужить отправной точкой для выявления ошибки.

После обнаружения ошибки ее можно исправить и заново выполнить программу в отладочном режиме.

Чтобы не отвлекаться на детали реализации Windows-приложений, построим нашу демонстрационную программу как консольное приложение. Консольные приложения выглядят подобно программам, написанным в среде MS-DOS.

Для создания подобного проекта нужно сначала выполнить команду меню **File ▶ New...**. После этого нужно перейти на вкладку **Projects** диалогового окна **New**. На этой вкладке надо выбрать тип **Win32 Console Application**, ввести имя проекта в текстовом поле **Project Name**, указать имя каталога для размещения файлов в поле **Location** и нажать кнопку **OK**.

Назовем нашу программу именем **Buggy**. Программа должна вычислять среднее арифметическое первых пяти натуральных чисел. Нетрудно догадаться, что в итоге должно получаться число 3, однако из-за специально сделанных ошибок программа сначала будет выдавать неправильный ответ¹.

Для построения программы создайте проект типа «консольное приложение» с именем **Buggy** и добавьте к проекту файл **buggy.cpp** со следующим текстом:

```
#include <iostream.h>
int main()
{
    const N = 5;
    int a[N] = {1, 2, 3, 4, 5};
    float sum, average;
    int i;
    for (i = 1; i < N; i++)
        sum += a[i];
    average = sum / N;
    cout << "average = " << average << endl;
    return 0;
}
```

Откомпилируйте проект и запустите его. В консольном окне приложения будет выведено нечто вроде следующего результата:

```
average = -2.14748e+007
```

Программа вычислила, что среднее арифметическое первых пяти целых чисел равно $-21\ 474\ 800$ (на вашем компьютере может быть и другое число), а это мало похоже на число 3.0. Теперь можно начать отладку нашей злополучной программы.

Установка точки прерывания

Точка прерывания позволяет остановить выполнение программы перед любой выполняемой инструкцией, чтобы продолжать выполнение программы либо в пошаговом режиме, либо в непрерывном режиме до следующей точки прерывания.

Чтобы задать точку прерывания перед некоторым оператором, необходимо установить перед ним текстовый курсор и нажать клавишу **F9** или щелкнуть мышью на кнопке **Insert/Remove BreakPoint** на панели инструментов **Build MiniBar**. Точка прерывания обозначается в виде коричневого кружка на левом поле окна редактирования. Повторный щелчок на указанной кнопке снимает точку прерывания. В программе может быть несколько точек прерывания.

Выполнение программы до точки прерывания

Программа запускается в отладочном режиме с помощью команды меню **Build ▶ Start Debug ▶ Go**. В результате код программы выполняется до той строки, на которой

¹ Пожалуйста, закройте глаза на ошибки, если вы сразу увидите их в тексте программы. Приводимый пример несложен и предназначен только для изучения возможностей отладчика.

установлена точка прерывания. Затем программа останавливается и отображает в окне Editor ту часть кода, где находится точка прерывания. Причем желтая стрелка на левом поле указывает на строку, которая будет выполняться на следующем шаге отладки.

Продолжим демонстрацию описываемых средств на примере программы **Buggy**. Установите точку прерывания перед оператором **for** и запустите программу в отладочном режиме, нажав клавишу **F5**.

Обратите внимание на то, что меню **Build** заменилось на меню **Debug**. Вид главного окна Visual Studio в этот момент показан на рис. П1.5. Стрелка на левом поле окна редактора указывает на инструкцию **for**, и отладчик остановился перед ее выполнением.

Среди различных команд меню **Debug** особый интерес представляют команды **Step Into** (**F11**), **Step Over** (**F10**), **Step Out** (**Shift+F11**), **Run To Cursor** (**Ctrl+F10**) и **Stop Debugging** (**Shift+F5**). Выполнение последней команды вызывает завершение работы с отладчиком.

О дополнительных окнах отладчика, расположенных в нижней части главного окна Visual C++, мы поговорим попозже.

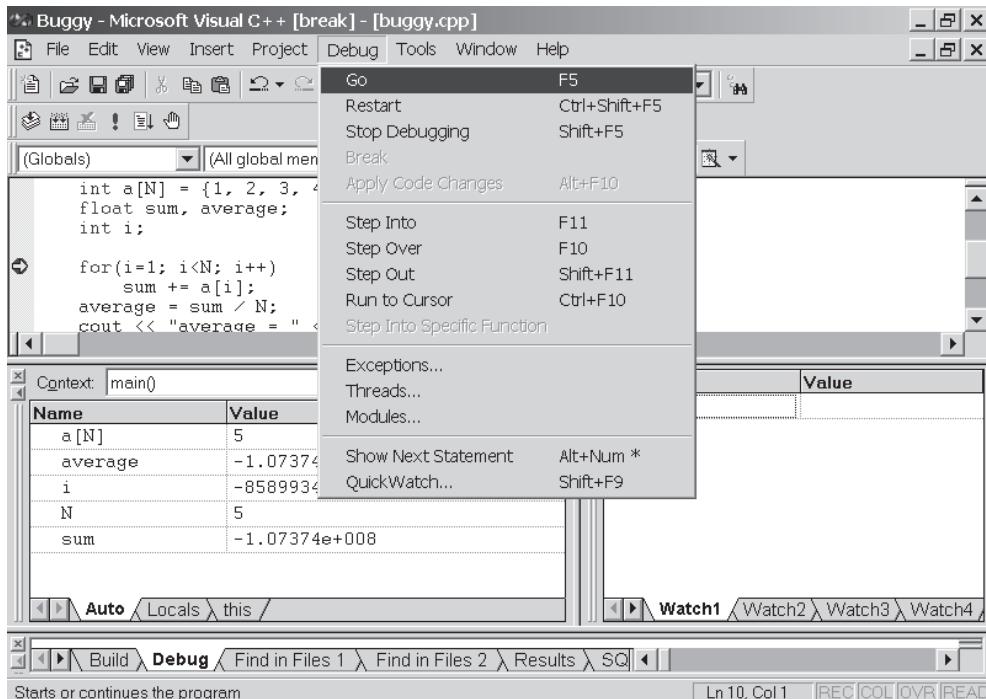


Рис. П1.5. Меню Debug и дополнительные окна отладчика

Пошаговое выполнение программы

Нажимая клавишу **F10**, можно выполнять один оператор программы за другим. Предположим, что при пошаговом выполнении программы вы дошли до строки, в которой вызывается некоторая функция **func1()**. Если вы хотите посмотреть, как работает код вызываемой функции, то надо нажать клавишу **F11**. Если же внут-

ренняя работа функции вас не интересует, а нужен только результат ее выполнения, то надо нажать клавишу F10.

Допустим, что вы вошли в тело функции func1(), нажав клавишу F11, но через некоторое время, пройдя несколько строк кода, решили выйти из функции. В этом случае надо нажать клавиши Shift+F11.

Существует и другая возможность пропустить пошаговое выполнение некоторого фрагмента программы. Надо установить текстовый курсор в нужное место программы и нажать клавиши Ctrl+F10.

Продолжим отладку программы Buggy. Нажмите клавишу F10. Указатель следующей выполняемой команды переместится на следующий оператор:

```
sum += a[i];
```

Проверка значений переменных во время выполнения программы

Чтобы узнать значение переменной sum, в которой будет накапливаться сумма элементов массива a, задержите над ней указатель мыши. Рядом с именем переменной на экране появляется подсказка со значением этой переменной:

```
sum = -1.07374e+008
```

Ага!!! Ведь еще не было никакого суммирования, и, следовательно, переменная sum, по идеи, должна быть равна нулю. Вот в чем дело! Мы забыли обнулить переменную sum до входа в цикл.

Нажмите комбинацию клавиш Shift+F5, чтобы выйти из отладчика и исправить найденную ошибку. Найдите строку с о следующим объявлением:

```
float sum, average;
```

Добавьте в нее инициализацию переменной sum:

```
float sum = 0, average;
```

Откомпилируйте заново проект при помощи кнопки Build (F7) и запустите его кнопкой Execute Program (Ctrl+F5). В результате будет отображен новый результат:

```
average = 2.8
```

Уже теплее, но все равно еще неправильно. Нажмите любую клавишу для завершения работы приложения. Необходимо продолжить отладку.

Установите точку прерывания перед оператором for и запустите программу в отладочном режиме, нажав клавишу F5. Теперь нажмите клавишу F10. Указатель следующей выполняемой команды переместится на следующий оператор:

```
sum += a[i];
```

Нажмите клавишу F10, и указатель переместится на следующий оператор:

```
for (i = 1; i < N; i++)
```

Задержите теперь над переменной sum указатель мыши. Рядом с именем переменной появится подсказка со значением этой переменной:

```
sum = 2.
```

Стоп!!! Позади первая итерация цикла, и в переменной sum должно находиться значение первого элемента массива a, то есть число 1. А мы получили число 2, то есть значение второго элемента массива a. Пришло время вспомнить, что в языке C++ нумерация элементов массива начинается с нуля. Поэтому ошибка находится в заголовке цикла:

```
for (i = 1; i < N; i++)
```

Эту ошибку нужно немедленно исправить. Повторите действия для исправления ошибки. Надо выйти из отладчика и исправить текст программы в операторе for:

```
for (i = 0; i < N; i++)
```

Теперь проект надо заново откомпилировать и запустить. Если в процессе исправления вы не внесли новых ошибок, то должен получиться результат

```
average = 3.0.
```

Итак, программа работает правильно.

В заключение отметим, что отладчик предоставляет и другие возможности для наблюдения за значениями переменных во время выполнения программы.

Окна Variables и Watch

Когда программа запускается в отладочном режиме, отладчик выводит два дополнительных окна для наблюдения за значениями переменных (см. рис. П1.5). Эти два окна размещаются ниже окна Editor, но выше окна Output. Окно Variables располагается слева, а окно Watch находится справа. Они могут быть спрятаны или вызваны на экран при помощи команды меню View ▶ Debug Windows.

На вкладке Auto окна Variables отображаются значения последних переменных, с которыми работал Visual C++. В верхней части окна Variables находится окно раскрывающегося списка Context, об использовании которого будет сказано ниже.

В окне Watch на любой из вкладок можно задать имя переменной, за текущими значениями которой вы хотите понаблюдать.

Более подробную информацию о работе с этими окнами вы можете найти в справочной системе через меню Help главного окна Visual C++.

Некоторые полезные инструменты

IDE Microsoft Visual C++ 6.0 содержит много удобных средств, облегчающих разработку программ. К ним относятся, в частности, редакторы ресурсов, работа с которыми рассматривается в главах 5 и 7. Из других инструментов большую пользу могут принести команда Find in Files и окно Context со списком нескольких последних вызовов функций.

Команда «Find in Files»

Доступ к команде Find in Files обеспечивает кнопка на панели инструментов, показанная на рис. П1.6.



Рис. П1.6. Кнопка Find in Files

Допустим, вы хотите найти все вхождения в файлы вашего проекта какого-либо имени. Наведите текстовый курсор на интересующее вас имя и нажмите кнопку Find in Files. Появится диалоговое окно Find in Files, в котором текстовое поле уже будет содержать интересующее вас имя. Вы можете дополнительно взвеси флагшки Match whole word only (искать только отдельные слова) и/или Match case (искать с учетом регистра букв). После этого остается нажать кнопку Find, и вы сразу получите в окне Output полный список вхождений заданного имени по всем файлам проекта.

Попробовав один раз, вы уже не сможете отказаться от регулярного общения с этим инструментом, ведь он так облегчает повседневную жизнь программиста!

Окно Context

Наиболее неприятной для программиста является ситуация с неожиданным крахом программы, при котором система выводит диалоговое окно с текстом «Access Violation». Это значит, что в результате какой-то ошибки ваша программа попыталась обратиться к недоступному для нее адресу оперативной памяти.

Окно Context, находящееся в верхней части окна Variables (см. рис. П1.5) и представляющее собой раскрывающийся список, помогает программисту в таких ситуациях быстро локализовать то место в программе, которое явилось причиной возникновения нештатной ситуации. В этом окне содержится список нескольких последних вызовов функций перед фатальным концом работы.

В этом окне надо найти самый последний вызов, который принадлежит вашему исходному коду. Если щелкнуть мышью по этому элементу списка, то стрелка на левом поле окна редактора укажет на строку программы, содержащую «кriminalnyy» вызов. Далее — уже дело техники (или аналитической работы ваших серых клеточек), чтобы сообразить, что же у вас «не так, как надо...».

Рассмотрим простой пример. Довольно частой ошибкой является использование указателя, для которого не выделена требуемая память, чтобы хранить значение соответствующего типа. Предположим, что в групповом проекте используется некий класс A, разрабатываемый двумя программистами¹. Допустим, что в классе A имеется член со следующим определением:

```
RECT* pRect;
```

Предположим далее, что первый программист, кодирующий конструктор класса A, забыл выделить память для указателя pRect. А в это время второй программист, разрабатывающий метод Show, будучи уверенным, что память уже выделена, напишет следующий фрагмент:

```
void A::Show() {
    int x0 = pRect->left;
    int y0 = pRect->top;
}
```

Все! «Аннушка уже пролила масло...»²

Любая программа, использующая объект класса A, при вызове метода Show завершится необработанным исключением (unhandled exception), о чем сообщает окно примерно такого вида (рис. П1.7).



Рис. П1.7. Печальная весть о кончине программы

¹ Мы здесь абстрагируемся от рекомендаций по надлежащей организации групповой разработки проекта и от рекомендаций по стилю кодирования. Нарушение этих рекомендаций как раз и является питательной средой для появления ошибок того типа, который мы рассматриваем.

Однако если вы запустили приложение с такой ошибкой в отладочном режиме (F5), то после щелчка на кнопке **OK** отладчик показывает картинку, приведенную на рис. П1.8.

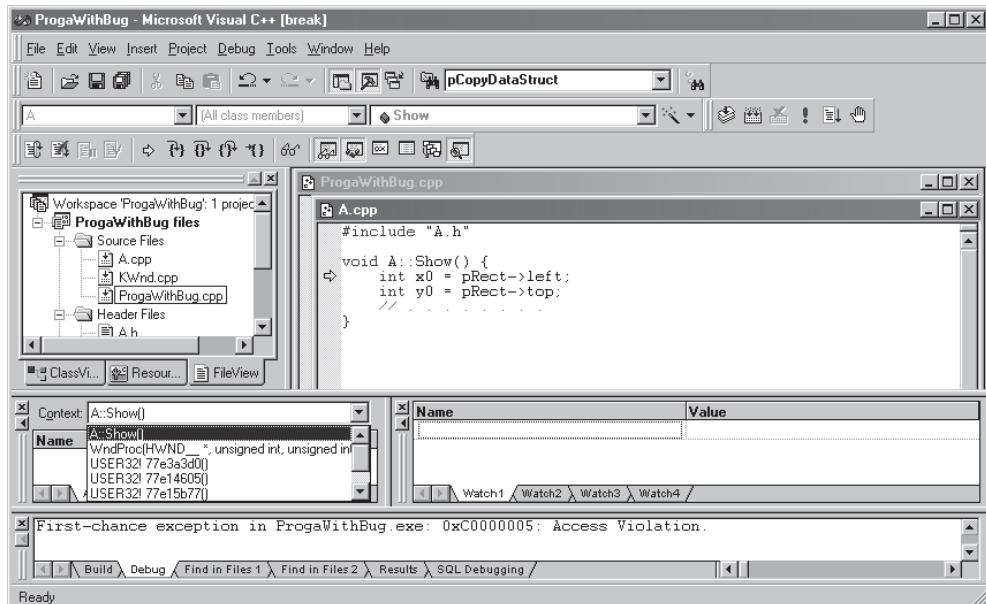


Рис. П1.8. Информация в окне Context, помогающая локализовать место краха программы

Последним вызовом функций в списке, который показывает окно **Context**, является вызов метода **A::Show()**. Цветная стрелка на левом поле окна редактора показывает инструкцию, при выполнении которой и случилось наше несчастье. Так как в этой инструкции используется указатель **pRect**, вполне обоснованным будет ваше желание проверить, а все ли с этим указателем в порядке?

Счастливого плавания!

Приложение 2

Интегрированная среда Visual Studio.NET

Мы предполагаем, что пакет Microsoft Visual Studio.NET уже установлен на вашем компьютере. Работа в IDE Visual Studio.NET имеет много общего с работой в IDE Visual Studio 6.0, описанной в приложении 1. Но в то же время имеются отличия как в интерфейсе пакета, так и в используемой терминологии.

В этом приложении описываются шаги по созданию нового проекта типа Win32 Application – empty project, компиляции, сборке и выполнению приложения.

После запуска Visual Studio.NET главное окно программы имеет вид, показанный на рис. П2.1. В зависимости от настроек для вашего пакета Visual Studio.NET вид главного окна может несколько отличаться от того, который показан на рисунке.

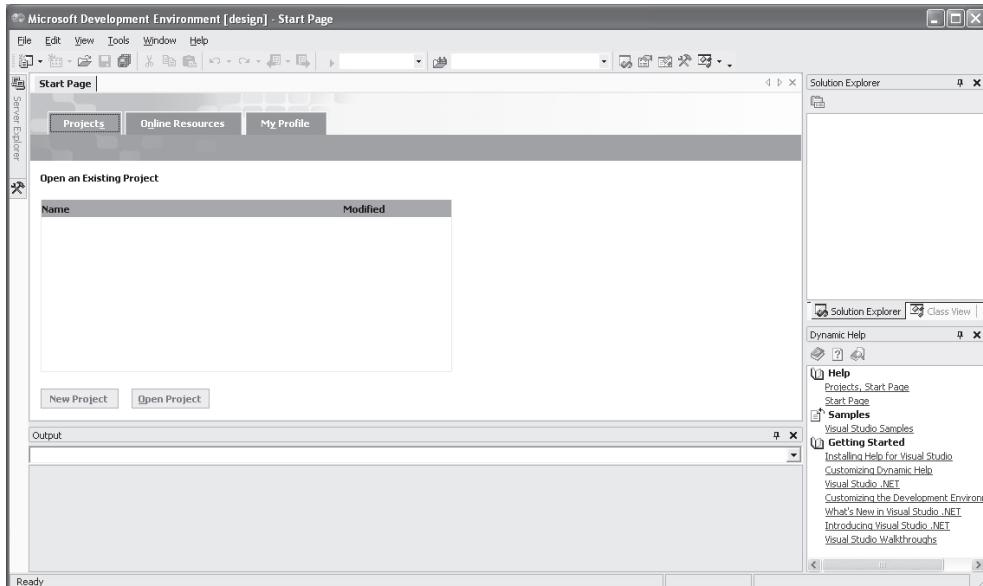


Рис. П2.1. Главное окно (рабочий стол) Visual Studio.NET

Конечно, уже сейчас можно найти первые отличия от Visual Studio 6.0.

Окно Project Workspace переместилось от левого края к правому краю главного окна и изменило свое название. Теперь оно называется **Solution Explorer**.

На месте окна **Editor** появилось окно с вкладками, которое сначала содержит только одну страницу **StartPage**. По мере построения проекта в этом окне могут появляться и другие страницы.

Таким образом, термином «Решение» теперь называется то, что раньше называлось «Рабочей областью». «Решение» также может содержать несколько проектов, но все наши программные примеры содержат ровно один проект.

Окно **Output** осталось на том же месте и по-прежнему служит для вывода сообщений о ходе компиляции, сборки и выполнения программы.

Создание нового проекта

Для создания нового проекта типа **Win32 Application — empty project** нужно выполнить следующую последовательность действий. Сначала надо выполнить команду меню **File ▶ New ▶ Project**. Это приведет к отображению окна **New Project**, показанного на рис. П2.2.

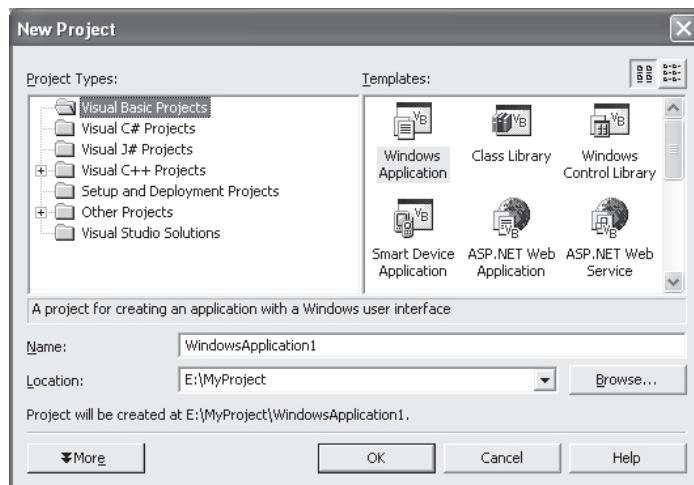


Рис. П2.2. Окно New Project

В списке **Project Types** следует открыть папку **Visual C++ Project**. Окно **New Project** примет вид, показанный на рис. П2.3.

В списке подтипов для **Visual C++ Project** выберите пиктограмму папки **Win32**. Окно **New Project** примет вид, показанный на рис. П2.4.

В окне **Templates** надо выделить шаблон **Win32 Project**. Затем надо указать имя проекта в текстовом поле **Name**, например, **HelloFromMsgBox**. Также введите имя папки для размещения проекта в поле **Location**, например, **E:\MyProject**. После нажатия кнопки **OK** будет отображено окно **Win32 Application Wizard – HelloFromMsgBox** (рис. П2.5).

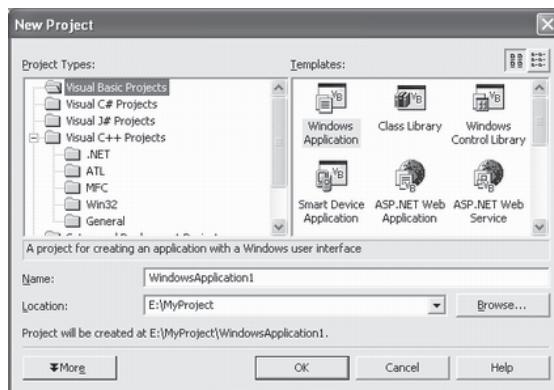


Рис. П2.3. Окно New Project после раскрытия папки Visual C++ Project

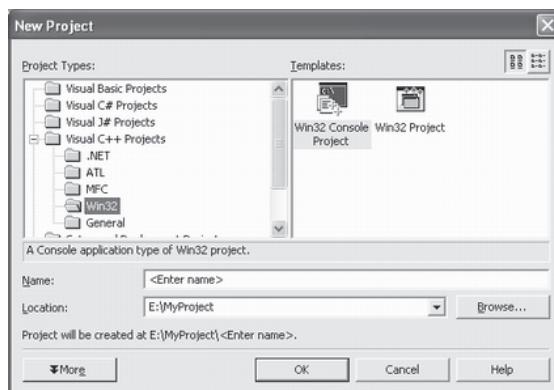


Рис. П2.4. Окно New Project после выбора папки Win32

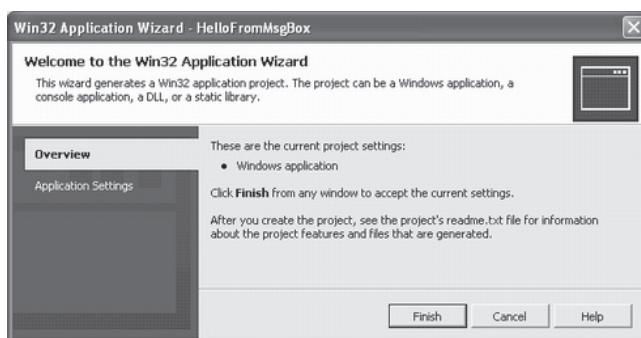


Рис. П2.5. Окно Win32 Application Wizard – HelloFromMessageBox

Шелкните мышью на пункте Application Settings. Окно мастера примет вид, показанный на рис. П2.6.

В группе флажков Additional options установите флажок Empty project и нажмите кнопку Finish. Окно Visual Studio.NET примет вид, показанный на рис. П2.7.

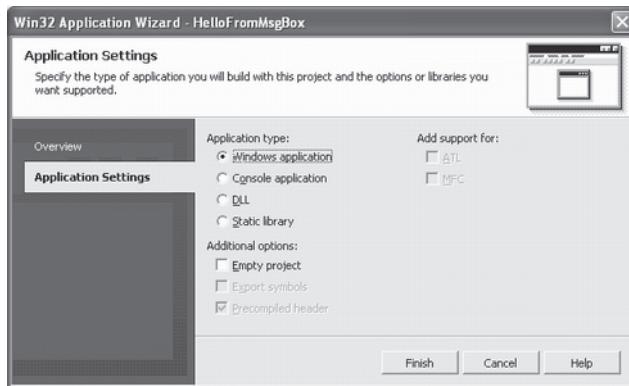


Рис. П2.6. Окно после выбора пункта Application Settings

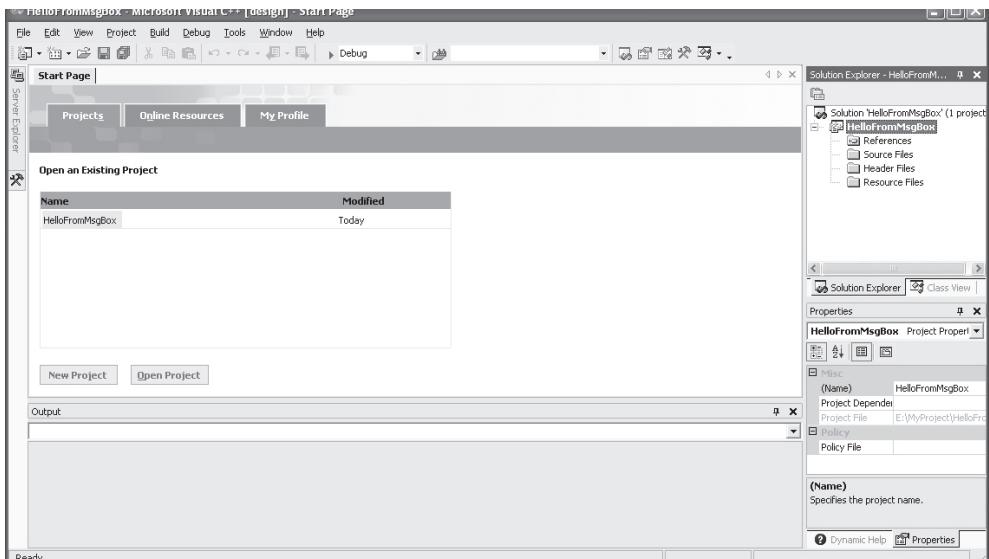


Рис. П2.7. Окно Visual Studio.NET после создания нового проекта

Создание нового проекта завершено.

Откроем папку проекта HelloFromMsgBox, чтобы посмотреть на служебные файлы, созданные мастером. Обратите внимание на два файла. Файл HelloFromMsgBox.sln — это файл «решения» (рабочей области), а HelloFromMsgBox.vcproj — это файл проекта. Таким образом, расширение .sln здесь используется вместо расширения .dsw для Visual C++ 6.0 и, соответственно, расширение .vcproj — вместо расширения .dsp.

Добавление к проекту нового файла

Чтобы добавить к проекту новый файл, нужно выполнить простую последовательность действий. Сначала щелкните правой кнопкой мыши на пиктограмме

папки Source Files в окне Solution Explorer. В появившемся контекстном меню надо выбрать команду Add ▶ Add New Item. В результате будет отображено окно Add New Item (рис. П2.8).

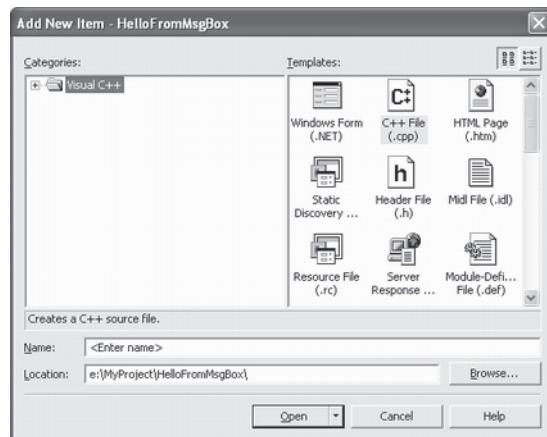


Рис. П2.8. Окно Add New Item

В окне Templates нужно выделить шаблон C++ File или Header File в зависимости от того, какого типа файл нужно присоединить к проекту. Введите имя файла в текстовое поле Name, например, HelloFromMsgBox, и нажмите кнопку Open. После этого в списке файлов Source Files окна Solution Explorer появится обозначение нового файла, например, HelloFromMsgBox.cpp, а также откроется окно текстового редактора.

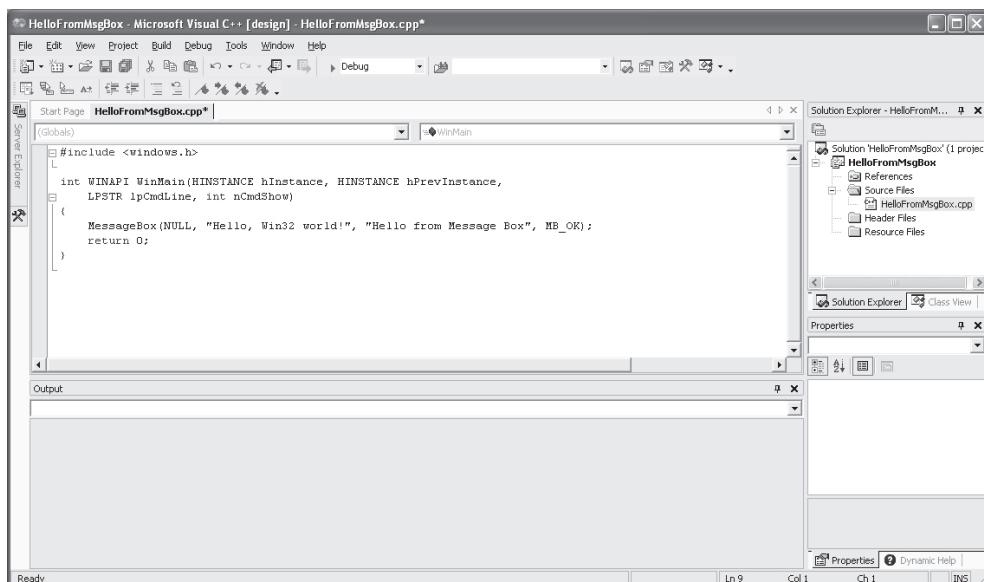


Рис. П2.9. Окно Visual Studio.NET после ввода текста в файл HelloFromMsgBox.cpp

Введите текст в окне текстового редактора, например:

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MessageBox(NULL, "Hello, Win32 world!", "Hello from Message Box", MB_OK);
    return 0;
}
```

После ввода текста основное окно Visual Studio.NET примет вид, показанный на рис. П2.9.

Сохраните набранный текст с помощью команды меню File ▶ Save.

Компиляция, сборка и выполнение

Выберите в меню главного окна команду Build ▶ Build Solution (Ctrl+Shift+B).

Сообщения о ходе компиляции и сборки будут поступать в окно Output. У нас ошибок нет, поэтому окно выглядит, как показано на рис. П2.10.

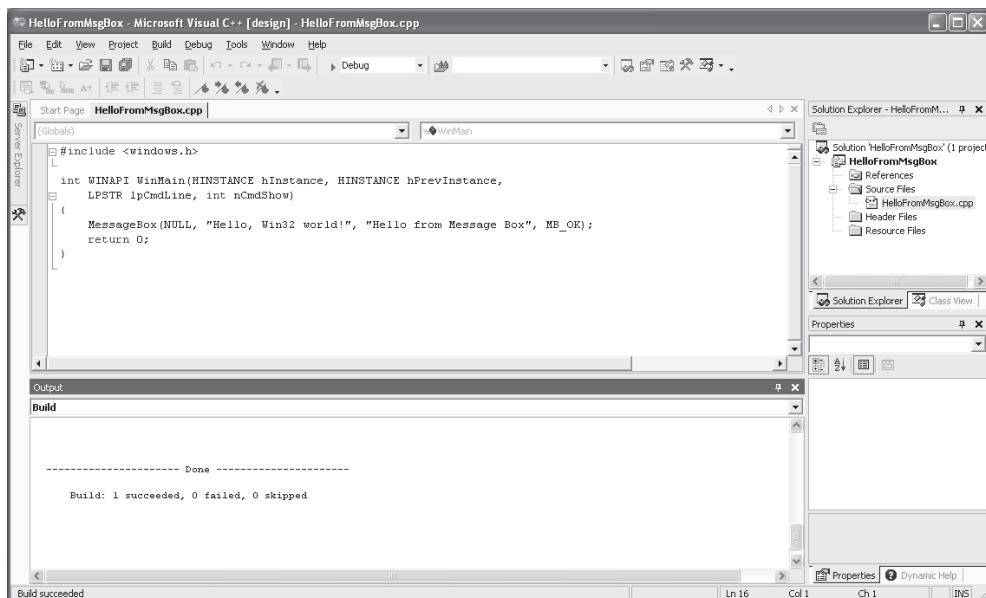


Рис. П2.10. Окно Visual Studio.NET после компиляции и сборки проекта HelloFromMsgBox

Откомпилированное приложение можно запустить на выполнение двумя способами.

Выполнение в обычном режиме производится с помощью команды меню Debug ▶ Start Without Debugging (Ctrl+F5).

Запуск в отладочном режиме производится с помощью команды меню Debug ▶ Start (F5).

Работа с редакторами ресурсов

Для добавления ресурсов в приложение вместо команды меню **Insert ▶ Resource**, которая применялась в Visual Studio 6.0, здесь используется команда **Project ▶ Add Resource**. После этой команды появляется диалоговое окно **Add Resource**. Выбор типа добавляемого ресурса делается так же, как в Visual Studio 6.0.

В Visual Studio.NET вместо панели инструментов **Controls** используется окно **Toolbox**, вызываемое с помощью команды меню **View ▶ Toolbox**. Это окно расположено в левой части главного окна Studio.NET.

Необходимый элемент управления в окне **Toolbox** выбирается щелчком мыши. Затем для размещения элемента управления на форме диалога делается повторный щелчок мыши в том месте, где следует расположить элемент.

Окно свойств для элементов управления здесь вызывать не нужно, так как свойства выбранного элемента управления отображаются в окне **Properties**, которое находится либо ниже окна **Solution Explorer**, либо рядом с ним.

Другие действия с проектами осуществляются примерно так же, как и при работе в среде Visual Studio 6.0.

Приложение 3

Работа с утилитой Spy++

В Microsoft Visual Studio 6.0 (а также в Visual Studio.NET) есть инструментальное средство под названием **Spy++**. Эта программа «шпионит» за окном другого приложения, чтобы разработчик мог иметь представление о том, какие сообщения проходят через это окно.

В этом приложении описывается работа с утилитой **Spy++** в составе Visual Studio 6.0. Запустить утилиту можно при помощи команды меню **Visual Studio Tools ▶ Spy++**. Окно утилиты **Spy++** имеет вид, показанный на рис. П3.1.

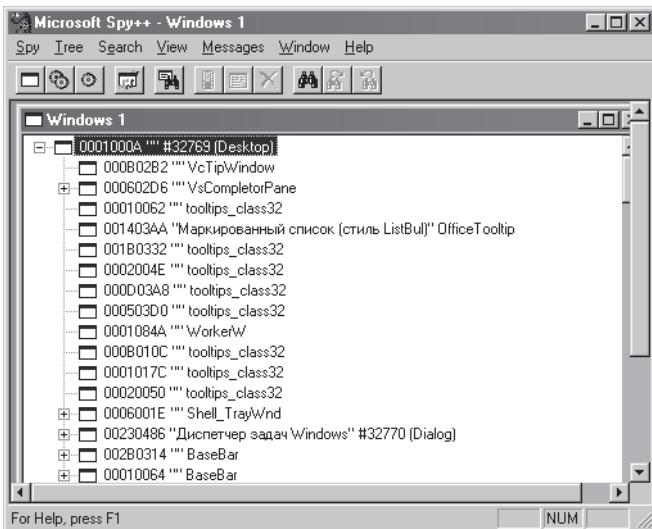


Рис. П3.1. Окно утилиты **Spy++**

Технология шпионского наблюдения за интересующим вас приложением **XXX** довольно проста. Сначала запустите на выполнение приложение **XXX**. Затем разместите на экране окно утилиты **Spy** и окно приложения **XXX** так, чтобы они были видны одновременно и не перекрывали друг друга. Выполните команду меню **Spy ▶ Messages**. В результате будет отображено диалоговое окно **Message Options** (рис. П3.2).

На вкладке **Windows** схватите левой кнопкой мыши пиктограмму **Finder Tool** и, удерживая кнопку, перетащите ее в то окно приложения **XXX**, за которым вы хотите наблюдать. После этого отпустите левую кнопку мыши.

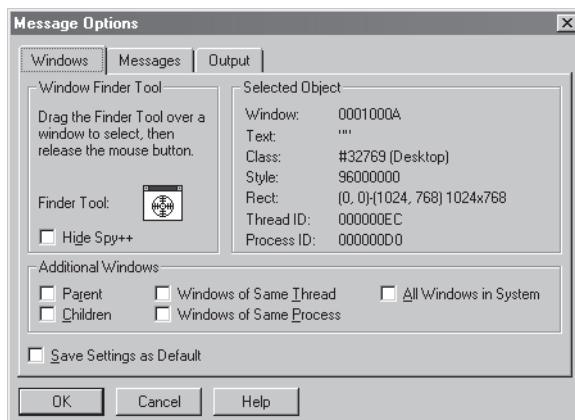


Рис. П3.2. Окно Message Options

В группе Selected Object будет отображена следующая информация:

- дескриптор окна (Window);
- заголовок окна (Text);
- имя класса;
- стиль;
- координаты окна;
- дескриптор потока (Thread ID);
- дескриптор процесса (Process ID).

Теперь надо нажать кнопку OK, и будет отображено окно Messages (Window...) (рис. П3.3).

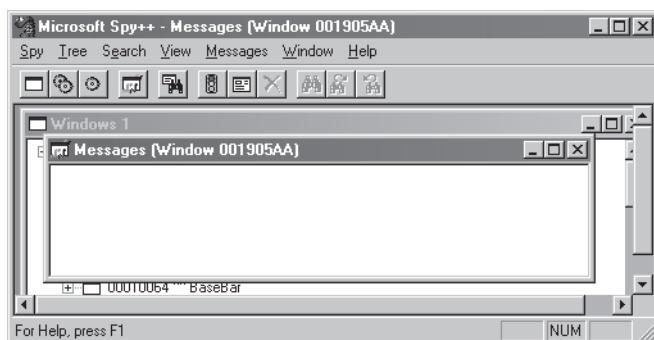


Рис. П3.3. Окно Messages (Window...)

После этого утилита Spy++ готова фиксировать все сообщения, поступающие в «помеченнное» окно. Сообщения фиксируются в окне Messages (Window...) в следующем формате:

Текущий номер сообщения	Дескриптор окна	Код сообщения	Идентификатор сообщения	Значения параметров
-------------------------	-----------------	---------------	-------------------------	---------------------

Код сообщения может иметь одно из значений, приведенных в табл. П3.1.

Таблица П3.1. Возможные коды сообщений

Код	Описание
P	Сообщение было отправлено в очередь сообщений с помощью функции PostMessage
S	Сообщение было отправлено в очередь сообщений с помощью функции SendMessage
s	Сообщение было отправлено с помощью SendMessage, но служба безопасности (security) не позволяет получить возвращаемое значение
R	Каждой строке с кодом S должна соответствовать строка с кодом R (return), которая содержит значение ответного сообщения. Иногда вызовы сообщений вложены друг в друга, когда обработчик одного сообщения посыпает другое сообщение

При первом сеансе использования Spy++ вас, по-видимому, поразит количество сообщений, протекающих через окно. Например, при нажатии кнопки закрытия окна в основном окне наблюдается более сорока сообщений.

Можно, однако, фильтровать поток сообщений, фиксируемых в окне Messages (Window ...), чтобы упростить анализ работы приложения. Для фильтрации сообщений вернитесь в диалоговое окно Message Options и откройте вкладку Messages (рис. П3.4).

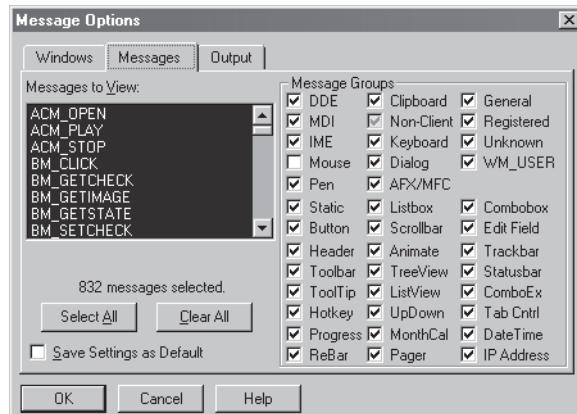


Рис. П3.4. Вкладка Messages

Здесь есть два механизма включения/исключения фиксируемых сообщений:

1. В группе **Message Groups** можно включить или исключить группу сообщений, относящихся либо к некоторому типу элемента управления, либо к некоторому внешнему источнику сообщений. Например, сообщения, поступающие от мыши.
2. В списке **Messages to View** содержится полный перечень сообщений, и каждое из них можно включить или исключить. По умолчанию все сообщения включены.

Использование утилиты Spy++ помогает иногда понять, почему ваше приложение работает не так, как вы ожидали.

Список литературы

1. *Петзольд Ч.* Программирование для Windows 95. Т. 1. — СПб: BHV — Санкт-Петербург, 1997. — 739 с.
2. *Мюррей У., Паппас К.* Создание переносимых приложений для Windows. — СПб: BHV — Санкт-Петербург, 1997. — 816 с.
3. *Саймон Р.* Microsoft Windows 2000 API. Энциклопедия программиста. — Киев: ДиаСофт, 2001. — 1008 с.
4. *Вильямс М.* Программирование в Windows 2000. Энциклопедия пользователя. — Киев: ДиаСофт, 2000. — 640 с.
5. *Рихтер Дж.* Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows. — СПб: Питер, 2001. — 752 с.
6. *Юань Фенъ.* Программирование графики для Windows. — СПб: Питер, 2002. — 1072 с.
7. *Румянцев П. В.* Азбука программирования в Win32 API. — М.: Горячая линия — Телеком, 2001. — 312 с.
8. *Касперски К.* Техника оптимизации программ. Эффективное использование памяти. — СПб: BHV — Санкт-Петербург, 2003. — 464 с.
9. *Павловская Т., Щупак Ю.* C++. Объектно-ориентированное программирование: Практикум. — СПб: Питер, 2004. — 265 с.

Алфавитный указатель

A

atomic access 459

B

bitmap 59

bitmap stretching mode 182

brush 59

Button 323

D

DDB 175

DDK 19

device space 67

dialog box 304

dialog procedure 304

dialog template units 308

DIB 175

DIB-секция 201

E

event 464

F

font 59

font mapping 128

G

GDI 19, 58

GUI 16

H

handle 20

hotkey 273

M

mailslots 470

mapping modes 71

mouse cursor 226

MSMQ 470

mutex 467

P

page file 471

page space 67

palette 59

pen 59

physical device space 67

primary thread 443

Progress bar 423

project workspace 563

R

region 59

ritical section 460

S

scan code 214

SDK 19

security descriptor 442

semaphore 466

shortcut menu 293

Slider 428

sockets 470

Spin 435

status bar 413

swap file 471

T

thread 443

thread context 443

Tooltip 391

TrueType 119

TSC 489

U

UTC-время 487

V

Virtual Memory Manager

471

VMM 471

W

Wait-функции 462

window procedure 42

world space 67

A

активное окно 18

Анонимный канал 470

атомарный доступ 459

B

библиотека динамической компоновки 19

Библиотеки динамической компоновки (DLL) 20

бинарные растровые операции 79

C

векторная графика 174

венгерская нотация 27

Видимый регион 60

Виртуальный код клавиши 216

виртуальный код клавиши 214

Время Windows 486

выключка 134

D

главное меню 272

горячая клавиша 273

E

двойная буферизация 543

дескриптор 20

дескриптор защиты 442
дескрипторы
 экземпляра приложения 55
диалоговая процедура 304
Диалоговое окно 304
динамическое связывание 20
Диспетчер виртуальной памяти 471
дочернее окно 18

И

Именованный канал 470
Индикатор процесса 423
информационный контекст 66

К

каскадное меню 274
квант 444
Кисть 35
кисть 59
класс окна 23
«классы
 KDib» 184
 KWndEx» 408
классы
 KDibSection 557
 KDocument 146
 KTimer 492
 KWnd 46
 QTimer 502
Клиент 473
Кнопка 323
Комбинированный список 356
Контекст потока 443
Контекст устройства 43, 59
Контекст устройства в памяти 65
Контекстное меню 293
контекстное меню 272
координаты Windows
 клиентские 28
 оконные 28
 экранные 28
Кривая Безье 86
Критическая секция 460
курсор мыши 226

Л

Латентность 491
Логический шрифт 124

М

макросы
 GetBValue 77
 GetGValue 77
 GetRValue 77
 HIWORD 55
 LOWORD 55
 MAKEINTRESOURCE 253
 PALETTEINDEX 77
 PALETTTERGB 77
 RGB 77
Метафайл 206
Мировая система координат 67
модальные диалоговые окна 304
мультимелейная библиотека winmm.lib 266
мьютекс 467

Н

немодальные диалоговые окна 304
немые клавиши 218

О

Обновляемый регион 60
окно Windows
 клиентская область 29
окно верхнего уровня 18
Окно подсказки 391
Окно редактирования 341
окно сообщений 26
оконная процедура 22, 42
оконный класс

 регистрация класса 32
 стандартный (предопределенный) 23
 стили 33
описатель 20
Отсечение 112

П

«палитра» 159
палитра 59
Панель инструментов 385

первичный поток 443
перо 59, 87
Пиксел 76
Пиктограмма 243
пиктограмма 28
поразрядные флаги 33
Поток 443
поток 17
Примитив синхронизации 459
Пропускная способность 491
Процесс 443
процесс 17

Р

Рабочая область 563
разрешение системного таймера 487
растеризация шрифта 120
растровая графика 174
Растровая кисть 99
растровая операция 79
растровое изображение 59
Регион 59, 109
Регион отсечения 60
Регулятор 428
редактор. См. См.
AppBrowser
режим масштабирования
растра 182
режим отображения 71
режим отображения MM_TEXT 72
режим пользователя 441
режим рисования 80
режим смешивания фона OPAQUE 81
TRANSPARENT 81
режим ядра 441
Ресурсы 242
родительское окно 18

С

семафор 466
Сервер 473
Система координат устройства 67
системная очередь сообщений 22
Системное время 487

- Скан-код 214
 скан-код 41
 Событие 464
 совместимый контекст 65
 Сокеты 470
 сообщение 21
 «сообщения»
 PBM_SETPOS» 424
 PBM_SETRANGE» 423
 PBM_SETSTEP» 424
 PBM_STEPIT» 424
 SB_GETRECT» 414
 SB_SETMINHEIGHT» 415
 SB_SetParts» 414
 SB_SetText» 414
 SB_SIMPLE» 414
 TB_ADDSTRING» 390
 TB_AUTOSIZE» 391
 TBM_GETPOS» 431
 TBM_SetLineSize» 430
 TBM_SetPageSize» 430
 TBM_SETRANGE» 430
 TBM_SetTicFreq» 430
 UDM_GetPos» 437
 UDM_SetRange» 437
 WM_CHAR» 218
 WM_COMMAND» 284
 WM_CONTEXTMENU» 293
 WM_CTLCOLOREDIT» 343
 WM_CTLCOLORSTATIC» 321
 WM_DRAWITEM» 325
 WM_HSCROLL» 430, 437
 WM_INITDIALOG» 318
 WM_INITMENU» 284
 WM_KEYDOWN» 216
 WM_KEYUP» 216
 WM_KILLFOCUS» 221
 WM_LBUTTONDOWNDBLCLK» 227
 WM_LBUTTONDOWN» 227
 WM_MENUCHAR» 284
 WM_MENUSELECT» 284, 416
 WM_MOUSEMOVE» 226
- «сообщения (продолжение)»
 WM_MOUSEWHEEL» 227
 WM_NOTIFY» 385
 WM_SETFOCUS» 221
 WM_SETFONT» 321
 WM_SYSCOMMAND» 284
 WM_VSCROLL» 430, 437
- сообщения
 WM_CLOSE 45, 51
 WM_COMMAND 51
 WM_COPYDATA 474
 WM_CREATE 51
 WM_DESTROY 46, 51
 WM_ERASEBKND 61
 WM_HSCROLL 144
 WM_INITDIALOG 51
 WM_MOVE 51
 WM_NCPAINT 61
 WM_PAINT 42, 51, 61
 WM_QUIT 24, 41
 WM_SIZE 51
 WM_TIMER 51
 WM_VSCROLL 144
- очередь сообщений
 приложения 22
 параметры сообщения 23
 синхронные и асинхронные 53
 цикл обработки сообщений 24, 40
- Список 344
 сплайн Безье 86
 сплошная кисть 97
 статическое связывание 19
 Страницчная система координат 67
 страницный файл 471
 Страна состояния 413
 «структуры»
 BITMAP» 193
 BITMAPFILEHEADER» 176
 BITMAPINFO» 183
 BITMAPINFOHEADER» 176
 CHOOSECOLOR» 377
 CHOOSEFONT» 377
 DRAWITEMSTRUCT» 325
- «структуры (продолжение)»
 INITCOMMONCONTROLSEX» 381
 LOGPALETTE» 171
 NMHDR» 392
 OPENFILENAME» 377
 PALETTEENTRY» 161
 RGBQUAD» 180
 TBBUTTON» 389
 TOOLINFO» 407
 TOOLTIPTEXT» 392
- структуры
 COPYDATASTRUCT 474
 LOGBRUSH 100
 LOGFONT 125
 MSG 40
 PAINTSTRUCT 61
 POINT 40
 PROCESS_INFORMATION 448
 RECT 44
 SCROLLINFO 143
 STARTUPINFO 447
 SYSTEMTIME 487
 TEXTMETRIC 121
 WNDCLASSEX 32
 XFORM 72
- счетчик 435
 счетчик меток реального времени 489
- Т**
- типы данных
 COLORREF 77
 HBRUSH 97
 HFONT 124
 HGDIOBJ 87
 HICON 253
 HPEN 87
 типы данных Windows 20
- Ф**
- файл подкачки 471
 Физическая система координат 67
 Фокус ввода 214
 фокус ввода 18
 «функции»
 AppendMenu» 281
 BitBlt» 197

«функции (*продолжение*)
CheckDlgButton» 332
CheckRadioButton» 333
CreateBitmap» 193
CreateBitmapIndirect» 193
CreateCaret» 219
CreateCompatibleBitmap» 194
CreateCompatibleDC» 197
CreateDialog» 366
CreateDIBitmap» 194
CreateDIBSection» 201
CreateEnhMetaFile» 207
CreateHalftonePalette» 168
CreateStatusWindow» 413
CreateToolBarEx» 390
DeleteMenu» 281
DestroyCaret» 220
DialogBox» 319
DrawMenuBar» 283
EndDialog» 319
GetDialogBaseUnits» 308
GetDlgItem» 306
GetDlgItemInt» 344
GetDlgItemText» 343
GetEnhMetaFileHeader» 211
GetMenu» 284
GetObject» 196
GetSubMenu» 284
HideCaret» 220
InitCommonControlsEx» 380
InsertMenu» 281
IsDialogMessage» 367
LoadAccelerators» 301
LoadBitmap» 195
LoadImage» 195
LoadLibrary» 382
LoadMenu» 281
MessageBox» 372
OutputDebugString» 413
PlayEnhMetaFile» 210
RealizePalette» 168
RemoveMenu» 281
SelectPalette» 168
SendDlgItemMessage» 342

«функции (*продолжение*)
SetDIBitsToDevice» 183
SetDlgItemText» 344
SetMenu» 281
SetMenuItemDefaultItem» 276
SetMenuItemInfo» 281
SetStretchBltMode» 182
SetWindowText» 321
ShowCaret» 220
ShowWindow» 312
StretchBlt» 198
StretchDIBits» 181
TrackPopupMenuEx» 294
TranslateAccelerator» 301
TranslateMessage» 218
функции
_beginthreadex 456
AngleArc 85
Arc 84
ArcTo 85
BeginPaint 43, 61
CheckMenuItem 282
CheckMenuRadioItem 283
Chord 106
CloseHandle 442
CombineRgn 111
CreateBrushIndirect 100
CreateCompatibleDC 65
CreateEllipticRgn 109
CreateEllipticRgnIndirect 109
CreateEvent 464
CreateFileMapping 472
CreateFont 125
CreateFontIndirect 128
CreateHatchBrush 98
CreateIC 66
CreateMutex 467
CreatePatternBrush 99
CreatePen 90
CreatePenIndirect 91
CreatePolygonRgn 110
CreatePolyPolygonRgn 110
CreateProcess 447
CreateRectRgn 109
CreateRectRgnIndirect 109
CreateRoundRectRgn 110
CreateSemaphore 466

функции (*продолжение*)
CreateSolidBrush 97
CreateThread 451
CreateWindow 35
CreateWindowEx 38
DefWindowProc 42
DeleteCriticalSection 461
DeleteObject 82
DestroyWindow 46
DispatchMessage 41
DrawFocusRect 105
DrawText 44, 136
DrawTextEx 136, 139
Ellipse 105
EnableMenuItem 283
EndPaint 62
EnterCriticalSection 460
ExtCreatePen 93
ExtTextOut 140
FillRect 104
FindResource 264
FindWindow 474
FrameRect 104
GetLocalTime 487
GetArcDirection 85
GetBkMode 81
GetClassLong 56
GetClientRect 44
GetDC 62
GetDeviceCaps 74
GetDlgItemID 306
GetMessage 41
GetModuleHandle 56, 255
GetPixel 78
GetStockObject 82, 88, 124
GetSystemTime 487
GetSystemTimeAdjustment 487
GetTextCharacterExtra 133
GetTextExtentPoint32 134
GetTextFace 129
GetTextMetrics 121
GetTickCount 487
GetWindowDC 62
GetWindowRect 68
InflateRect 102
InitializeCriticalSection 460
InterlockedDecrement 459

функции (*продолжение*)

InterlockedIncrement 459
 InvalidateRect 64, 260
 InvalidateRgn 64
 InvertRect 105
 KillTimer 504
 LeaveCriticalSection 461
 LineTo 83
 LoadIcon 253
 LoadImage 253
 LoadResource 264
 LoadString 270
 LockResource 264
 MapViewOfFile 472
 MessageBox 26
 ModifyMenu 283
 ModifyWorldTransform 74
 MoveToEx 83
 OffsetRect 102
 OffsetRgn 111
 OpenEvent 465
 OpenSemaphore 466
 PeekMessage 57
 Pie 106
 PlaySound 265
 PolyBezier 87
 PolyBezierTo 87
 Polygon 107
 Polyline 84
 PolylineTo 84
 PolyPolygon 108
 PolyPolyline 84
 PostMessage 54
 PostQuitMessage 46
 PtInRegion 110
 QueryPerformanceCounter 489

функции (*продолжение*)

QueryPerformanceFrequency 488
 Rectangle 103
 RegisterClass 35
 RegisterClassEx 32
 ReleaseDC 62
 ReleaseSemaphore 466
 RestoreDC 73
 RoundRect 106
 SaveDC 73
 ScrollWindow 145
 SelectClipRgn 113
 SelectObject 82
 SendDlgItemMessage 346
 SendMessage 54
 SendNotifyMessage 54
 SetArcDirection 85
 SetBkColor 45
 SetBkMode 45, 81
 SetBrushOrgEx 98
 SetClassLong 51
 SetCursor 258
 SetDCPenColor 89
 SetGraphicsMode 73
 SetMapMode 71
 SetPixel 78
 SetPixelV 78
 SetPolyFillMode 108
 SetPriorityClass 446
 SetRect 102
 SetROP2 80
 SetTextAlign 130
 SetTextCharacterExtra 133
 SetTextColor 45
 SetTextJustification 134

функции (*продолжение*)

SetThreadPriority 446
 SetTimer 504
 SetViewportExtEx 71
 SetViewportOrgEx 70
 SetWindowExtEx 70
 SetWindowOrgEx 70
 SetWindowRgn 112
 SetWorldTransform 67, 73
 ShellExecute 268
 ShowWindow 39
 Sleep 452, 498
 TabbedTextOut 131
 TextOut 130
 timeBeginPeriod 509
 timeEndPeriod 509
 timeGetDevCaps 509
 timeKillEvent 511
 timeSetEvent 510
 TranslateMessage 41
 TryEnterCriticalSection 461
 UnmapViewOfFile 472
 UpdateWindow 63
 WaitForMultipleObjects 463
 WaitForSingleObject 462
 обратного вызова (CALLBACK) 22
 поддержки окон 49
функция
 DispatchMessage 24
функция WinMain 25

Ш

шаблонные единицы 308
 Шрифт 59
 Штриховая кисть 98