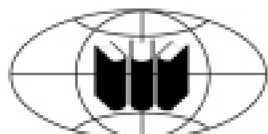
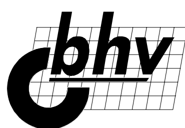


Ч. Петзолд

Программирование для Windows® 95

в двух томах

Том I



«BHV — Санкт-Петербург»

Дюссельдорф Киев
Москва Санкт-Петербург

Содержание

ЧАСТЬ I ВВЕДЕНИЕ **9**

ГЛАВА 1 README.TXT 11

Вызов, брошенный программистам *11*

Основные правила *12*

Краткая история Windows *13*

Краткая история этой книги *14*

Начнем *15*

ГЛАВА 2 HELLO, WINDOWS 95 17

Отличительная особенность Windows *17*

Графический интерфейс пользователя 17

Концепции и обоснование GUI 18

Содержимое интерфейса пользователя 18

Преимущество многозадачности 18

Управление памятью 19

Независимость графического интерфейса от оборудования 19

Соглашения операционной системы Windows 19

Вызовы функций 20

Объектно-ориентированное программирование 20

Архитектура, управляемая событиями 20

Оконная процедура 21

Ваша первая программа для Windows *21*

Что в этой программе неправильно? 22

Файлы HELLOWIN 22

Make-файл 25

Файл исходного текста программы на языке С 25

Вызовы функций Windows 26

Идентификаторы, написанные прописными буквами 26

Новые типы данных 27

Описатели 27

Венгерская нотация 28

Точка входа программы 28

Регистрация класса окна 29

Создание окна 31

Отображение окна 32

Цикл обработки сообщений 33

Оконная процедура 34

Обработка сообщений 34

Воспроизведение звукового файла 35

Сообщение WM_PAINT 35

Сообщение WM_DESTROY 36

Сложности программирования для Windows *37*

Не вызывай меня, я вызову тебя 37

Синхронные и асинхронные сообщения 38

Думайте о ближнем 39

Кривая обучения 39

ГЛАВА 3 РИСОВАНИЕ ТЕКСТА 41

Рисование и обновление *41*

Сообщение WM_PAINT 42

Действительные и недействительные прямоугольники 42

Введение в графический интерфейс устройства (GDI) *43*

Контекст устройства 43

Получение описателя контекста устройства. Первый метод 43

Структура информации о рисовании 44

Получение описателя контекста устройства. Второй метод 45

Функция *TextOut*. Подробности 46

Системный шрифт 47

Размер символа 47

Метрические параметры текста. Подробности 48

Форматирование текста 49

Соединим все вместе ... 50
Оконная процедура программы SYSMETS1.C ... 53
Не хватает места! ... 54
Размер рабочей области ... 55
Полосы прокрутки ... 55
Диапазон и положение полос прокрутки ... 56
Сообщения полос прокрутки ... 57
Прокрутка в программе SYSMETS ... 58
Структурирование вашей программы для рисования ... 62
Создание улучшенной прокрутки ... 62
Мне не нравится пользоваться мышью ... 67
ГЛАВА 4 ГЛАВНОЕ О ГРАФИКЕ ... 69
Концепция GDI ... 69
Структура GDI ... 70
Типы функций ... 70
Примитивы GDI ... 71
Другие аспекты ... 72
Контекст устройства ... 72
Получение описателя контекста устройства ... 72
Получение информации из контекста устройства ... 74
Программа DEVCAPS1 ... 74
Размер устройства ... 77
О цветах ... 77
Атрибуты контекста устройства ... 78
Сохранение контекста устройства ... 79
Рисование отрезков ... 80
Ограничивающий прямоугольник ... 84
Сплайны Безье ... 89
Использование стандартных перьев ... 93
Создание, выбор и удаление перьев ... 94
Закрашивание пустот ... 96
Режимы рисования ... 96
Рисование закрашенных областей ... 97
Функция *Polygon* и режим закрашивания многоугольника ... 98
Закрашивание внутренней области ... 99
Режим отображения ... 100
Координаты устройства (физические координаты) и логические координаты ... 101
Системы координат устройства ... 102
Область вывода и окно ... 102
Работа в режиме MM_TEXT ... 103
Метрические режимы отображения ... 105
Ваши собственные режимы отображения ... 106
Программа WHATSIZE ... 109
Прямоугольники, регионы и отсечение ... 112
Работа с прямоугольниками ... 112
Случайные прямоугольники ... 113
Создание и рисование регионов ... 117
Отсечения: прямоугольники и регионы ... 118
Программа CLOVER ... 118
Пути ... 121
Создание и воспроизведение путей ... 121
Расширенные перья ... 122
Bits and Blts ... 125
Цвета и битовые образы ... 126
Битовые образы, не зависящие от устройства (DIB) ... 126
Файл DIB ... 127
Упакованный формат хранения DIB ... 128
Отображение DIB ... 128
Преобразование DIB в объекты "битовые образы" ... 128
Битовый образ — объект GDI ... 129
Создание битовых образов в программе ... 129
Формат монохромного битового образа ... 130
Формат цветного битового образа ... 131
Контекст памяти ... 131

Мощная функция *BitBlt* ... 132
Перенос битов с помощью функции *BitBlt* ... 135
Функция *DrawBitmap* ... 136
Использование других ROP кодов ... 136
Дополнительные сведения о контексте памяти ... 138
Преобразования цветов ... 141
Преобразования режимов отображения ... 141
Растяжение битовых образов с помощью функции *StretchBlt* ... 141
Кисти и битовые образы ... 142
Метафайлы ... 143
Простое использование метафайлов памяти ... 144
Сохранение метафайлов на диске ... 147
Расширенные метафайлы ... 147
Делаем это лучше ... 147
Базовая процедура ... 148
Заглянем внутрь ... 151
Вывод точных изображений ... 153
Текст и шрифты ... 154
Вывод простого текста ... 154
Атрибуты контекста устройства и текст ... 156
Использование стандартных шрифтов ... 157
Типы шрифтов ... 157
Шрифты TrueType ... 158
Система EZFONT ... 158
Внутренняя работа ... 161
Форматирование простого текста ... 162
Работа с абзацами ... 163

ЧАСТЬ II СРЕДСТВА ВВОДА ... 169

ГЛАВА 5 КЛАВИАТУРА ... 171
Клавиатура. Основные понятия ... 171
Игнорирование клавиатуры ... 171
Фокус ввода ... 172
Аппаратные и символьные сообщения ... 172
Аппаратные сообщения ... 173
Системные и несистемные аппаратные сообщения клавиатуры ... 173
Переменная *lParam* ... 173
Виртуальные коды клавиш ... 175
Положения клавиш сдвига и клавиш-переключателей ... 177
Использование сообщений клавиатуры ... 178
Модернизация SYSMETS: добавление интерфейса клавиатуры ... 178
Логика обработки сообщений WM_KEYDOWN ... 178
Посылка асинхронных сообщений ... 179
Символьные сообщения ... 184
Сообщения WM_CHAR ... 185
Сообщения немых символов ... 186
Взгляд на сообщения от клавиатуры ... 186
Каретка (не курсор) ... 190
Функции работы с кареткой ... 190
Программа TYPER ... 191
Наборы символов Windows ... 196
Набор символов OEM ... 197
Набор символов ANSI ... 198
Наборы символов OEM, ANSI и шрифты ... 198
Международные интересы ... 199
Работа с набором символов ... 199
Связь с MS-DOS ... 199
Использование цифровой клавиатуры ... 200
Решение проблемы с использованием системы UNICODE в Windows NT ... 201
ГЛАВА 6 МЫШЬ ... 203
Базовые знания о мыши ... 203
Несколько кратких определений ... 203
Сообщения мыши, связанные с рабочей областью окна ... 204
Простой пример обработки сообщений мыши ... 205

Обработка клавиш <Shift> 208
Двойные щелчки клавиш мыши 209
Сообщения мыши нерабочей области 209
Сообщение теста попадания 210
Сообщения порождают сообщения 211
Тестирование попадания в ваших программах 211
Гипотетический пример 211
Пример программы 212
Эмуляция мыши с помощью клавиатуры 215
Добавление интерфейса клавиатуры к программе CHECKER 216
Использование дочерних окон для тестирования попадания 219
Дочерние окна в программе CHECKER 220
Захват мыши 223
Рисование прямоугольника 224
Решение проблемы — захват 227
Программа BLOKOUT2 227
ГЛАВА 7 ТАЙМЕР 231
Основы использования таймера 231
Система и таймер 232
Таймерные сообщения не являются асинхронными 232
Использование таймера: три способа 233
Первый способ 233
Второй способ 237
Третий способ 239
Использование таймера для часов 240
Позиционирование и изменение размеров всплывающего окна 243
Получение даты и времени 243
Обеспечение международной поддержки 243
Создание аналоговых часов 244
Стандартное время Windows 248
Анимация 249
ГЛАВА 8 ДОЧЕРНИЕ ОКНА УПРАВЛЕНИЯ 253
Класс кнопок 254
Создание дочерних окон 257
Сообщения дочерних окон родительскому окну 258
Сообщения родительского окна дочерним окнам 258
Нажимаемые кнопки 259
Флажки 259
Переключатели 260
Окна группы 260
Изменение текста кнопки 260
Видимые и доступные кнопки 261
Кнопки и фокус ввода 261
Дочерние окна управления и цвета 262
Системные цвета 262
Цвета кнопок 263
Сообщение WM_CTLCOLORBTN 263
Кнопки, определяемые пользователем 264
Класс статических дочерних окон 269
Класс полос прокрутки 270
Программа COLORS1 271
Интерфейс клавиатуры, поддерживаемый автоматически 276
Введение новой оконной процедуры 276
Закрашивание фона 277
Окрашивание полос прокрутки и статического текста 278
Класс редактирования 278
Стили класса редактирования 280
Коды уведомления управляющих окон редактирования 281
Использование управляющих окон редактирования 281
Сообщения управляющему окну редактирования 281
Класс окна списка 282
Стили окна списка 283
Добавление строк в окно списка 283
Выбор и извлечение элементов списка 284

Получение сообщений от окон списка ... 285
Простое приложение, использующее окно списка ... 285
Список файлов ... 288
Утилита Head для Windows ... 289

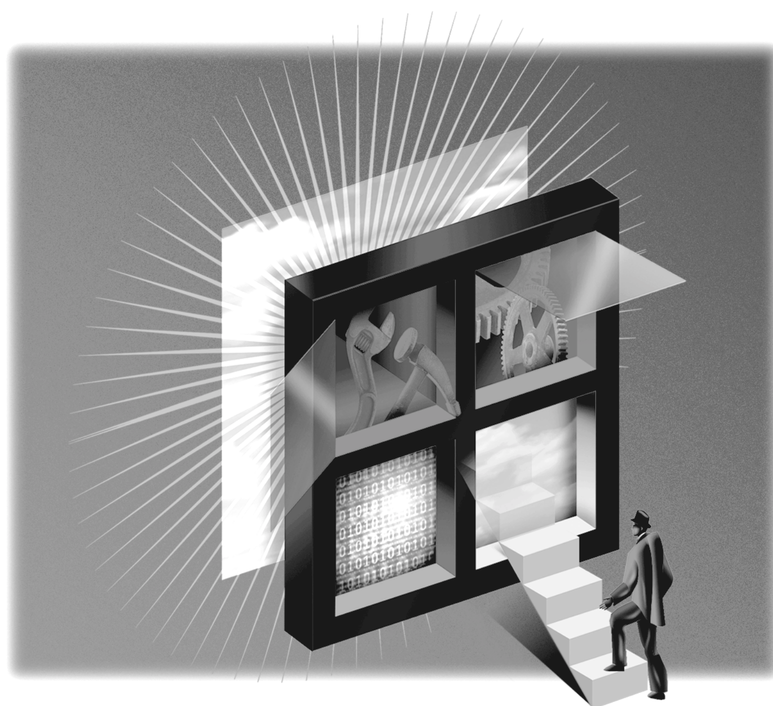
ЧАСТЬ III ИСПОЛЬЗОВАНИЕ РЕСУРСОВ ... 293

ГЛАВА 9 ЗНАЧКИ, КУРСОРЫ, БИТОВЫЕ ОБРАЗЫ И СТРОКИ ... 295
Компиляция ресурсов... 295
Значки и курсоры ... 296
Редактор изображений ... 298
Получение описателя значков ... 300
Использование значков в вашей программе ... 301
Использование альтернативных курсоров ... 301
Битовые образы: картинки в пикселях ... 302
Использование битовых образов и кистей ... 302
Символьные строки... 305
Использование ресурсов-символьных строк... 305
Использование ресурсов-строк в функции *MessageBox* ... 305
Ресурсы, определяемые пользователем ... 306
ГЛАВА 10 МЕНЮ И БЫСТРЫЕ КЛАВИШИ ... 313
Меню ... 313
Структура меню... 314
Шаблон меню... 314
Ссылки на меню в вашей программе... 315
Меню и сообщения... 316
Образец программы... 318
Этикет при организации меню ... 322
Сложный способ определения меню ... 322
Третий подход к определению меню... 324
Независимые всплывающие меню... 324
Использование системного меню ... 328
Изменение меню... 330
Другие команды меню ... 330
Нестандартный подход к меню ... 331
Использование в меню битовых образов... 335
Два способа создания битовых образов для меню ... 341
Контекст памяти ... 341
Создание битового образа, содержащего текст ... 341
Масштабирование битовых образов... 342
Соберем все вместе ... 343
Добавление интерфейса клавиатуры ... 344
Быстрые клавиши ... 344
Зачем нужны быстрые клавиши? ... 344
Некоторые правила назначения быстрых клавиш... 345
Таблица быстрых клавиш ... 345
Загрузка таблицы быстрых клавиш ... 346
Преобразование нажатий клавиш клавиатуры ... 346
Получение сообщений быстрых клавиш... 347
Программа POPPAD, имеющая меню и быстрые клавиши... 348
Разрешение пунктов меню... 352
Обработка опций меню ... 353
ГЛАВА 11 ОКНА ДИАЛОГА... 355
Модальные окна диалога... 355
Создание окна диалога About... 355
Шаблон окна диалога... 358
Диалоговая процедура... 359
Вызов окна диалога ... 360
Дополнительная информация о стиле окна диалога ... 361
Дополнительная информация об определении дочерних окон элементов управления ... 362
Более сложное окно диалога ... 364
Работа с дочерними элементами управления окна диалога ... 368
Кнопки OK и Cancel ... 370
Позиции табуляции и группы... 371
Рисование в окне диалога ... 373

Использование с окном диалога других функций ... 373
Определение собственных окон управления ... 373
Окна сообщений ... 378
Информация во всплывающих окнах ... 379
Немодальные окна диалога ... 379
Различия между модальными и немодальными окнами диалога ... 380
Новая программа COLORS ... 381
Программа HEXCALC: обычное окно или окно диалога? ... 385
Творческое использование идентификаторов дочерних окон элементов управления ... 390
Диалоговые окна общего пользования ... 391
Модернизированная программа POPPAD ... 391
Изменение шрифта ... 407
Поиск и замена ... 408
Программа для Windows, содержащая всего один вызов функции ... 408
ГЛАВА 12 СОВРЕМЕННЫЙ ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС ... 411
Основы элементов управления общего пользования ... 412
Инициализация библиотеки ... 413
Создание элементов управления общего пользования ... 413
Стили элементов управления общего пользования ... 414
Посылка сообщений элементам управления общего пользования ... 417
Уведомляющие сообщения от элементов управления общего пользования ... 418
Элементы управления главного окна ... 421
Панели инструментов ... 421
Создание панели инструментов ... 422
Строка состояния ... 434
Программа GADGETS ... 438
Наборы страниц свойств ... 465
Создание набора страниц свойств ... 466
Процедуры диалогового окна страницы свойств ... 470
Программа PROPERTY ... 475

Часть I

Введение



Глава 1 README.TXT

1

Эта книга для тех, кто будучи искусным программистом языка C, хотел бы научиться писать приложения для операционной системы Microsoft Windows 95. Близкое знакомство с языком C является первым из трех необходимых условий пользования этой книгой. Вторым условием является инсталляция 32-разрядной системы программирования Microsoft Visual C++ версии 4.0. И третье — реальное использование операционной системы Windows 95 и умение работать с ее пользовательским интерфейсом.

Как вы вероятно знаете, Windows 95 — это последнее воплощение графической операционной системы, впервые представленной в ноябре 1985 года для использования на компьютерах типа IBM PC и совместимых с ним. По мере проникновения на рынок, за последнее десятилетие, Windows почти полностью вытеснила всех имевшихся конкурентов и стала, фактически, эталоном операционной системы для персональных компьютеров. Теперь, если вы пишете программу для совместимых с IBM PC компьютеров, то вы пишете для Windows.

Считайте эту главу вашим первым днем в школе. Откажемся от склонности некоторых злобных учителей сразу бросаться в пучину учебного материала, ведь большинство из нас предпочитает более постепенное знакомство с ним. Поэтому, в этой главе будет рассказано о некоем историческом фоне Windows, об основных правилах пользования этой книгой и даже (с вашего позволения) немного об авторе и о том, как родилась эта книга.

Нельзя, однако, гарантировать, что в этой главе все будет столь безоблачно. Вы программист, вы инженер программного обеспечения, и, как и любой другой инженер, вы должны ставить трудные проблемы и решать их, делая мир удобнее и совершеннее. Вы строите дороги и мосты, которые служат людям, и эти конструкции должны быть крепки, устойчивы и непоколебимы.

Вызов, брошенный программистам

В фильме *Большой Каньон*, отец помогает пятнадцатилетнему сыну научиться управлять машиной и замечает: "Умение делать левый поворот в Лос-Анджелесе — это одна из наиболее трудных вещей в жизни, которую тебе следует научиться делать." То же самое он мог бы сказать о программировании для Windows.

Не будем здесь подробно касаться механики программирования для Windows. Займемся этим неприятным делом в следующей главе. Здесь мы побольше расскажем о философии составления программ, а, чтобы упростить их понимание, вернемся немного назад. Это относительно новая концепция.

В дни зарождения компьютеров программистами были сами пользователи. Первые компьютерные программы были неудобными, громоздкими и неэффективными. Простодушные пользователи в расчет не брались. Даже после того, как программное обеспечение компьютеров стало отчасти интерактивным и для ввода данных на телетайпе или дисплее появилась командная строка, часто пользователям необходимо было помнить множество команд и опций, которые не были представлены на экране.

Вероятно настоящая революция в составлении программ пришла с появлением первых интерактивных систем подготовки текстов (например, WordStar) и электронных таблиц (VisiCalc), которые объединили примитивные формы наиболее фундаментального элемента современного пользовательского интерфейса, а именно меню. Меню в тех ранних интерактивных приложениях было реализовано не слишком хорошо, но идея родилась, и она медленно развивалась и совершенствовалась. С нашей точки зрения, необходимость меню кажется очевидной: оно представляет пользователю все имеющиеся опции программы. Конечно, в те дни недостаток оперативной памяти ограничивал возможности программиста в создании хорошего пользовательского интерфейса. Программы, в которых интересы пользователя проигнорированы, короче и их легче писать; программы, в которых интересы пользователя учтены, длиннее и писать их труднее.

Здесь вполне уместна пословица о невозможности убить двух зайцев одним выстрелом. Должны страдать интересы либо пользователя, либо программиста, и кто-то должен заниматься более тяжким трудом. Вам, как программисту, суждено взять это бремя на себя.

К счастью, большая часть вашей трудной работы уже проделана конструкторами и программистами операционной системы Windows 95. Эти невоспетые герои уже реализовали большую часть кода, необходимого для создания объектов современного пользовательского интерфейса и для вывода программ на экран с использованием богатых возможностей оформления текста и графики. Занимаясь этим, они также создали развитой интерфейс программирования приложений (Application programming interface, API), который, однако, на первых порах вполне может испугать программистов, решивших работать с Windows. Это характерно не только для Windows, но и для любого современного графического интерфейса.

Раньше считалось, что для того, чтобы начать программировать для Windows, программисту нужно около 6 месяцев. (Иногда говорят, что при пользовании этой книгой, указанный срок мог бы сократиться до 26 недель или возможно даже до 180 дней.) За последние несколько лет Windows стала более обширной, но одновременно появились и дополнительные инструменты для решения трудных вопросов, поэтому, вероятно, правило шести месяцев остается вполне применимым.

Эта преамбула необходима, чтобы вы не почувствовали себя профессионально и умственно неполноценными, если не все сразу поймете. Если вы раньше никогда не программировали в графической среде, то, естественно, Windows окажется непривычной для вас. Однако, в конце концов вы добьетесь своего.

Основные правила

В этой книге мы хотели бы научить вас тому, что считается классическим программированием для Windows. Мы возьмем добрый старый язык программирования C (а не C++) и напрямую используем базовый интерфейс программирования приложений, а не какую бы то ни было другую оболочку, скрывающую под упрощенным, на первый взгляд, интерфейсом все тот же API. Несмотря на очевидные преимущества Windows 95 по сравнению с более ранними версиями Windows, большинство программ этой книги не слишком отличаются от программ, которые могли бы быть написаны (и писались) для Microsoft Windows версии 1.0 около десяти лет назад.

В определенном смысле эта книга описывает тяжелый путь создания программ для Windows, но вместе с тем это базовый, наиболее фундаментальный, гибкий и мощный путь. Используя другие подходы в программировании для Windows, вы не сможете реально добиться большего. Кроме того, изучая классическое программирование для Windows, использующее C и базовый API, вы сможете более отчетливо представить, как действуют Windows и ее приложения, и это знание может оказаться весьма полезным. Такой подход даст вам крепкий фундамент, о котором вы никогда не пожалеете. Поверьте.

Хотя книга учит классическому программированию для Windows, настоятельно рекомендуется этим не ограничиваться. В настоящее время существует множество средств, позволяющих облегчить программирование для Windows. Одним из таких популярных средств является язык C++, который используется в основном в сочетании с библиотеками классов, такими как Microsoft Foundation Classes (MFC) или Object Windows Library (OWL) фирмы Borland. Другими средствами являются Visual Basic фирмы Microsoft è Delphi фирмы Borland. В распоряжении программиста имеются также системы, которые генерируют коды и, таким образом, берут на себя некоторую часть работы по программированию для Windows. Существует даже возможность создавать приложения для Windows, используя простые описательные языки, например ToolBook фирмы Asymetrix.

Трудно сказать, какое из этих средств лучше; это сильно зависит от того, что за приложение создается, и насколько программист готов пожертвовать для этого своим временем.

Еще одной темой, которая не рассматривается в книге, является использование интегрированной среды разработчика (Integrated Development Environment, IDE), такой как Microsoft Visual C++ версии 4.0. Эта среда может облегчить вам работу при создании ресурсов (например, меню и окон диалога), обеспечит генерацию make-файлов (файлы, содержащие инструкции для компиляции и компоновки вашей программы, а также для создания исполняемого файла), и предоставит вам единую среду для компиляции, выполнения и отладки программ. IDE — это хорошо. Она превосходна. Однако, ее работа чаще вредит, чем помогает изучению классического программирования для Windows. Короче говоря, эта книга не научит вас формировать диалоговые окна в рамках IDE; она научит вас проектировать ваши собственные диалоговые окна и управлять процессом их заполнения.

По этой причине примеры программ в этой книге показаны так, как будто они создавались в обычном текстовом редакторе и компилировались из хорошо знакомой всем командной строки. К тому времени, когда мы доберемся до темы описания ресурсов, содержащих в виде инструкций сведения об окнах меню, диалога и других программных средств, такой подход даст большую наглядность. Описания ресурсов в книге даны так, что их можно прочитать, чего обычно не бывает в случае, если эти описания генерировались с помощью IDE. Так сделано для того, чтобы было понятно, что означает каждая инструкция описания ресурсов. Общеизвестно, что make-файлы становятся длиннее и сложнее, если они созданы с помощью IDE. Предлагаемые в книге файлы короче и проще. Нетекстовыми ресурсами в этой книге являются только иконки, указатели мыши и битовые картинки, для создания которых по самой их природе обычно требуются инструменты.

Преимущество такого подхода в том, что вы можете просто скопировать файл-источник на ваш жесткий диск, затем вызвать окно командной строки MS-DOS, запустить make-файл для получения исполняемого файла, а затем уже полученный файл немедленно запустить из командной строки.

Большинство программ этой книги очень короткие, предназначенные для демонстрации одного или двух конкретных положений. Иными словами, они максимально упрощены. Эти программы являются инструментами для обучения, а не образцами законченных программ для Windows. Например, каждая реальная программа для Windows должна обладать уникальной иконкой, что относится только к нескольким из приведенных программ, поскольку, создание для каждой программы своей иконки было бы просто топтанием на месте. Кроме этого, строки текста, использованные в программе, должны вставляться в файл описания ресурсов для его перевода на различные языки. Думается, что это было бы помехой в освоении тех принципов, которые отстаивает книга. Очевидно, что чем короче программа, тем проще читателю ее изучить.

Возможно эта книга не сможет охватить все аспекты программирования для Windows 95. Она также не заменит официальной технической документации. После усвоения материала книги, вам потребуется еще много часов на чтение документации, чтобы узнать о характеристиках и функциях Windows 95.

Это не единственная книга по программированию для Windows, опубликованная Microsoft Press. Как альтернативу программированию для Windows на С с использованием приложенного IDE, вы, возможно, захотите изучить программирование для Windows 95 с использованием MFC, представленное в книге Джефа Просиса "*Programming Windows 95 with MFC*" (публикация намечена на лето 1996 года). Чтобы лучше понять скрытую работу Windows 95, посмотрите книгу Адриана Кинга "*Inside Windows 95*" (имеется перевод этой книги: Адриан Кинг, "*Windows 95 изнутри*", Microsoft Press & Питер, 1995). Разнообразие возможностей пользовательского интерфейса, показанное в главе 12, более полно представлено в книге Нэнси Винник Клутс "*Programming the Windows 95 User Interface*". Поверхностное обсуждение OLE в главе 20 может быть дополнено книгой Крэйга Броксмита "*Inside OLE*". Разные аспекты программирования для Windows 95 можно найти в "*Programmer's Guide Microsoft Windows 95* ". Также интересными являются "*Inside Visual C++*" Дэвида Дж. Круглински, "*OLE Controls Inside Out* " Адама Денинга и "*Hardcore Visual Basic*" Брюса МакКини.

Последнее и основное правило этой книги состоит в том, что мы не будем копаться в недокументированных или неизученных аспектах программирования для Windows 95. Хотя эта информация и может оказаться интересной, она не является достаточно важной для всех приложений Windows, может быть только для каких-то наиболее необычных. Будучи опытным программистом, автор пытался по мере своих сил и возможностей не останавливаться на деталях реализации операционной системы, поскольку в ее будущих версиях детали могут измениться. Вместо этого он попытался так обойтись с API, как будто этот интерфейс полностью описан в технической документации. Это, конечно, далеко не всегда так, и часто документация не оправдывает возлагаемых на нее надежд. Также могут иметься и программные ошибки. Останавливаться на этих проблемах здесь мы не будем.

Краткая история Windows

Вскоре после появления в середине 1981 года IBM PC стало очевидно, что господствующей операционной системой для PC (включая совместимые) должна стать MS-DOS, что означает Microsoft Disk Operating System. Ранние версии MS-DOS обеспечивали для пользователя интерфейс командной строки, отображая такие команды как DIR и TYPE, которые могли загружать выполняемые программы в оперативную память и предлагали для этих программ определенный интерфейс для доступа к файлам, считывания информации с клавиатуры, и отображения на принтере и на экране дисплея (только в символьном режиме).

Из-за ограниченных возможностей программного и аппаратного обеспечения, псевдографическая среда пробивала себе дорогу медленно. Компьютеры Apple показали возможную альтернативу, когда в январе 1983 года была создана скандально известная Lisa, и затем в январе 1984 года Apple, разработав Macintosh, создала образцовую графическую среду, которая (несмотря на постепенную утрату этой моделью компьютера своих позиций на рынке) все еще рассматривается как эталон, по которому равняются создатели любой другой графической оболочки.

О работе над Windows корпорация Microsoft заявила в ноябре 1983 года (позже, чем появилась Lisa, но раньше, чем Macintosh) и реализовала ее двумя годами позже, в ноябре 1985 года. В течение двух следующих лет, Microsoft Windows версии 1.0 претерпела несколько модернизаций, необходимых для удовлетворения требований международного рынка. Кроме этого появились дополнительные драйверы для новых дисплеев и принтеров.

Windows версии 2.0 была создана в ноябре 1987 года. Эта версия содержала несколько изменений пользовательского интерфейса. Наиболее важное из этих изменений касалось использования перекрывающихся окон, вместо окон, расположенных рядом, что было характерно для Windows версии 1.x. Windows версии 2.0 содержала также улучшенный интерфейс клавиатуры и манипулятора мышь, а также, отчасти, окон меню и диалога.

В то время для Windows требовались только процессоры Intel 8086 или 8088, работающие в реальном режиме, при этом доступ осуществлялся к 1 мегабайту оперативной памяти. Windows/386 (созданная вскоре после Windows 2.0)

использовала виртуальный режим процессора Intel 80386 для запуска нескольких одновременно работающих с оборудованием программ MS-DOS в окнах. Для симметрии Windows версии 2.1 назвали Windows/286.

Windows версии 3.0 появилась 22 марта 1992 года. Здесь были объединены ранние версии Windows/286 и Windows/386. Главным изменением в Windows 3.0 была поддержка защищенного режима процессоров Intel 80286, 80386 и 80486. Это позволило Windows и ее приложениям получить доступ к 16 мегабайтам оперативной памяти. "Оболочка" программ Windows для запуска программ и поддержки файлов была полностью переделана. Windows 3.0 — это первая версия Windows, которая стала "родной" для множества пользовательских машин в домах и офисах.

Windows версии 3.1 появилась в апреле 1992 года. Сюда были включены такие важные свойства, как технология TrueType для шрифтов (что дало возможность масштабировать шрифты для Windows), multimedia (звук и музыка), OLE и диалоговые окна общего пользования. Кроме этого Windows 3.1 работала только в защищенном режиме и требовала процессора 80286 или 80386 и, по крайней мере, одного мегабайта оперативной памяти.

Любая история Windows была бы неполной без упоминания об операционной системе OS/2, альтернативной для DOS и Windows, которая на первом этапе развивалась корпорацией Microsoft в сотрудничестве с IBM. OS/2 версии 1.0 (только для символьного режима) работала на процессорах Intel 80286 (или более поздних) и появилась в конце 1987 года. Графическая оболочка Presentation Manager (PM) была реализована в OS/2 версии 1.1 в октябре 1988 года. PM, как изначально предполагалось, должна была стать версией защищенного режима Windows, но графический интерфейс программирования приложений так сильно изменился, что производителям программного обеспечения стало очень трудно поддерживать одновременно обе платформы.

К сентябрю 1990 года конфликт между IBM и Microsoft достиг своего апогея, что вынудило каждую компанию идти своим путем. IBM взяла на себя OS/2, а для Microsoft стало очевидно, что Windows должна стать основной стратегией развития операционных систем. Хотя у OS/2 все еще имеется немало горячих поклонников, ее популярность не идет ни в какое сравнение с популярностью Windows.

Windows NT, появившаяся в июле 1993 года, стала первой версией Windows, поддерживающей 32-разрядную модель программирования для процессоров Intel 80386 и 80486, а также Pentium. Windows NT имеет сплошное плоское (flat) 32-разрядное адресное пространство и 32-разрядные целые. Кроме этого Windows NT переносима и работает на нескольких моделях рабочих станций, основанных на RISC-технологии.

Windows 95 (первоначально условно названная Chicago, а иногда упоминающаяся и как Windows версии 4.0) появилась в августе 1995 года. Также как Windows NT, Windows 95 поддерживает 32-разрядную модель программирования (требуя, таким образом, для себя процессор 80386 и выше). Хотя у Windows 95 и нет некоторых возможностей Windows NT, таких как высокая степень безопасности и переносимость для работы с машинами, созданными по RISC-технологии; тем не менее она способна работать на компьютерах, имеющих всего 4 мегабайта оперативной памяти.

Очевидно, что программы, написанные до появления Windows NT и Windows 95 для 16-разрядных версий Windows, не вполне подходят для новейших 32-разрядных версий Windows; в первых главах будет рассказано о некоторых изменениях, необходимых для переносимости прежних программ.

Создавая API, Microsoft попыталась разделить различные реализации этого интерфейса. API Win16 поддерживается операционной системой Windows 3.1. API Win32 поддерживается системами Windows NT и Windows 95. Дальше — больше. Microsoft предоставила программистам возможность писать 32-разрядные приложения для Windows 3.1: с помощью динамически подключаемой библиотеки (Dynamic Link Library, DLL) вызовы 32-разрядных функций преобразуются в 16-разрядные вызовы. Такой API назвали Win32s (литера "s" означает "subset" — подмножество, поскольку этот API поддерживает только функции *Win16*). В то же время API для Windows 95 назвали Win32c ("c" от слова "compatible" — совместимый), но потом от этого названия отказались.

В настоящее время считается, что и Windows NT и Windows 95 поддерживают API Win32. Однако, у каждой из этих операционных систем имеются некоторые черты, которых нет у другой. Тем не менее общего у них гораздо больше, что позволяет писать программы, работающие в обеих системах. Кроме этого общеизвестно, что в ближайшем будущем эти системы объединятся.

Краткая история этой книги

В начале 1988 года, первая редакция книги "*Программирование для Windows*" стала одной из первых книг по программированию для Windows, положив начало их нынешнему изобилию. Когда была подготовлена новая версия книги "Программирование для Windows 95", то автор был поражен тем, что 1995 год — это десятая годовщина Windows и одновременно десятая годовщина той самой книги. Это одновременно и радостный, и пугающий факт.

Если вернуться к истории этой книги, то очевидно, что она появилась, в значительной степени, благодаря стечению обстоятельств и знакомствам автора со множеством прекрасных людей в отраслях, связанных с компьютерами. Теперь о том, как все начиналось.

Весной 1985 года автор работал над несколькими пространными статьями для журнала *PC Magazine* и проводил много времени в редакции на One Park Avenue в Нью-Йорке. Иногда здесь же появлялся Стив Балмер из Microsoft (теперь он исполнительный вице-президент по продажам) с очередной версией долгожданной операционной системы, о которой знали как о Windows. Те, кто интересовались Windows, вплоть до закрытия редакции могли оставаться в кабинете редактора Джона Дикинсона. Джон один из первых в *PC Magazine* установил у себя монитор EGA, и следовательно он мог работать с Windows в цвете. Желающие возились с этими первыми версиями Windows (обычно до тех пор, пока программа не рушилась), а затем ждали следующего раза, когда Балмер сумеет принести очередную, более устойчивую версию.

Однажды весной 1985 года автор спросил Джона Дикинсона о том, как реально идет написание программы при работе под Windows. Джон быстро выдвинул ящик стола, вывалил груду бумаги толщиной в несколько дюймов и около десятка дискет, которые Балмер оставил у него несколькими неделями раньше. Единственное, чего хотел Джон, это чтобы все это хозяйство больше у него в кабинете не появлялось.

Груда вываленных на стол бумаги и дискет было начальной версией пакета Microsoft Windows Software Development Kit (SDK) вместе с компилятором С. Автор забрал эту груду домой, инсталлировал SDK и, после почти шести месяцев непрерывных неудач, стал программистом для Windows. Во время этого эксперимента с обучением и борьбы с документацией, ему не раз приходила в голову мысль о том, что он мог бы объяснить содержимое этого пакета гораздо лучше, чем это делает Microsoft.

Windows версии 1.0 была окончательно готова к ноябрю 1985 года, но в то время никто не мог предположить, что со временем Windows может стать стандартом на рынке. Действительно, ее конкурентами в то время были TopView компании IBM, GEM компании Digital Research и DESQview компании Quarterdeck. 25 февраля 1985 года передовица *PC Magazine* провозгласила "Windows Wars!" Это была первая передовая статья, которую автор этой книги написал для журнала, хотя в то время многим TopView казался более перспективным, чем Windows.

Затем прошел почти целый год, пока не появилась твердая уверенность в необходимости обсуждения программирования для Windows в печати; это произошло в декабре 1986 года в журнале *Microsoft Systems Journal*. Считается, что эта статья, представляющая раннюю версию программы WHATSIZE, которую вы найдете в главе 4, — первая появившаяся в журнале статья о программировании для Windows. Автор узнал об этом от главного редактора *Microsoft Systems Journal* Джонатана Лазаруса, поскольку раньше он был вице-президентом Ziff-Davis — компании, которая выпускала *PC Magazine*. Позднее Джон перешел работать в Microsoft, где сейчас он остается на должности вице-президента по стратегической политике.

Но не только работы о программировании под Windows для *Microsoft Systems Journal* привлекли к автору внимание Microsoft Press. В октябре 1986 года в Редмонте, Вашингтон, на конференции фирмы Microsoft по языкам программирования он встретил Тенди Тровера (сейчас руководителя группы разработки пользовательского интерфейса в Microsoft) и рассказал ему, с каким подъемом писались для *Microsoft Systems Journal* статьи о программировании для Windows. Он сообщил имя автора этих статей некой Сьюзан Ламмерс, которая затем стала главным редактором Microsoft Press. Сотрудники Microsoft Press уже знали имя автора, поскольку вероятней всего, именно он начал рецензирование первой редакции "*Энциклопедии MS-DOS* " и сообщил им о наличии в ней бесчисленного количества изъянов и ошибок, что привело, в конце концов, к изъятию ее из обращения и тотальной переделке под строгим надзором Рэя Дункана.

В ноябре 1986 года в Лас-Вегасе автор этой книги неоднократно встречался с редактором Microsoft Клаудеттой Мур (сейчас она литературный агент в Массачусетсе), и вместе они набросали план будущего издания. Изначально "*Программирование для Windows*" было задумано очень небольшой книгой и предназначалось для программистов и квалифицированных пользователей. По мере работы над ней в течение следующего года (когда возникла нужда перестраиваться с Windows 1.0 на Windows 2.0), объем книги рос, а круг охватываемых ею проблем сужался.

То, что вы сейчас держите в руках — это четвертая редакция "*Программирования для Windows*". Она была исправлена для Windows 3.0, затем опять исправлена для Windows 3.1. С тех пор, как была опубликована первая редакция книги, многие программисты говорили, что "*Программирование для Windows*" стало для них отправной точкой в работе с этой достаточно необычной средой программирования. Ничто другое не было бы столь приятно, не могло бы принести большего удовлетворения автору, чем то, что его книги помогли кому-то в успешном освоении Windows.

Начнем

Итак, первый день обучения близок к завершению. В следующей главе мы начнем писать некоторые программы для Windows. Чтобы это делать, вам необходимо инсталлировать Microsoft Visual C++ версии 4.0. В Ваш файл AUTOEXEC.BAT необходимо включить инструкцию:

```
CALL \MSDEV\BIN\VCVARS32.BAT
```

Этот файл, включенный в VC++, устанавливает переменные среды DOS для компиляции программ с помощью командной строки MS-DOS. Он просто показывает путь к заголовочным, библиотечным и бинарным файлам. Кроме этого, чтобы запустить файл MSC.BAT, в вашем файле AUTOEXEC.BAT, необходимо будет использовать и вторую инструкцию CALL, показанную на рис. 1.1. Здесь также задаются переменные среды, используемые в make-файлах следующих глав.

MSC.BAT

```
REM -------------------------------------------------------------
REM  MSC.BAT -- Set up environment for Microsoft C/C++ 7.0 NMAKE
REM -------------------------------------------------------------
SET CC=cl
SET CFLAGS=-c -DSTRICT -G3 -Ow -W3 -Zp -Tp
SET CFLAGSMT=-c -DSTRICT -G3 -MT -Ow -W3 -Zp -Tp
SET LINKER=link
SET GUIFLAGS=-SUBSYSTEM:windows
SET DLLFLAGS=-SUBSYSTEM:windows -DLL
SET GUILIBS=-DEFAULTLIB:user32.lib gdi32.lib winmm.lib comdlg32.lib comctl32.lib
SET RC=rc
SET RCVARS=-r -DWIN32
```

Рис. 1.1 **Для компиляции программ из этой книги запустите этот файл**

Файл MSC.BAT можно найти в каталоге CHAP01 прилагаемой к книге дискеты CD-ROM.

О назначении представленных на рисунке инструкций, будет рассказано в следующих главах.

Глава 2 Hello, Windows 95

2

Если вы новичок в программировании для графической среды типа Microsoft Windows 95, то, вероятно, многое покажется вам совершенно отличным от всего, с чем вы сталкивались раньше. Windows имеет репутацию среды легкой для пользователя, и трудной для программиста. Новичков обычно сбивает с толку архитектура Windows и структура приложений, работающих в этой операционной системе. Если это происходит с вами, то пожалуйста не переживайте, что вам придется поломать голову, чтобы стать хорошим программистом для Windows. Этот первый конфуз вполне обычен, так бывает со многими.

Программирование для Windows — странное. Оно необычно, нестандартно, неудобно и часто запутывает. Оно абсолютно не очевидно, и может пройти немало времени, прежде чем ваши занятия завершатся победным "Эврика!" (то же самое, что студенческое "Ура, я это сделал!" — откровение, которое так любят преподаватели). Сделана обобщенная оценка, которая состоит в том, что программисты должны вытерпеть шестимесячную муку обучения, прежде чем стать сторонником составления программ для Windows, и даже после этого обучение не заканчивается и никогда не закончится. Можно только надеяться, что эта книга сократит на пару недель (а может быть на месяц, а может и на два) обычный ход обучения.

Тогда вы можете спросить: "Если программировать для Windows так трудно, зачем эти хлопоты?"

Ответ очевиден: "Вероятно, у вас нет другого выхода". В конце концов Windows так широко проникла на рынок PC-совместимых компьютеров, что необходимость программирования для "голой" MS-DOS (неважно в символьном режиме или графике) продлится недолго. Если вы пишете коммерческие приложения для широкого круга пользователей, обозреватели журналов по программному обеспечению компьютерной техники будут фактически игнорировать ваш товар, если он не работает под Windows. Если Вы пишете узкоспециализированные программы, то вашим пользователям (и вашим нанимателям!) не понравится тот факт, что ваши программы плохо сочетаются с существующими приложениями Windows, которыми они пользуются.

Отличительная особенность Windows

Windows обладает важными преимуществами и для пользователей, и для программистов по сравнению со средой MS-DOS. Выгоды для пользователей и выгоды для создателей программ на самом деле весьма схожи, поскольку задача создателя программы состоит в том, чтобы дать пользователю то, в чем он нуждается и то, что он хочет. Windows 95 делает это возможным.

Графический интерфейс пользователя

Windows — это графический интерфейс пользователя (Graphical User Interface, GUI), иногда его еще называют "визуальный интерфейс" или "графическая оконная среда". Концепции, давшие начало этому типу пользовательского интерфейса, берут свое начало в середине семидесятых годов, от первой работы, сделанной на Xerox Palo Alto Research Center (PARC) для таких машин как Alto и Star, и для такой среды как Smalltalk. Позднее эта работа была взята за основу и популяризована корпорацией Apple Computer, во-первых, в злополучной модели Lisa и, затем, год спустя в гораздо более удачной модели Macintosh, введенной в эксплуатацию в январе 1984 года.

После появления компьютера Macintosh, графические интерфейсы пользователя получили широкое распространение, причем как в сфере персональных компьютеров, так и не персональных компьютеров. Сейчас совершенно очевидно, что графический интерфейс пользователя является (по словам Чарльза Симони из Microsoft) наиболее важным "великим достижением" в сфере персональных компьютеров.

Концепции и обоснование GUI

Все графические интерфейсы пользователя делают возможным использование графики на растровом экране дисплея. Графика дает лучшее восприятие действительного положения вещей на экране, визуально богатую среду для передачи информации и возможность WYSIWYG (What you see is what you get, что вы видите, то и получите) как для графики, так и для форматированного для печати документов текста.

В первые дни своего существования дисплеи использовались исключительно для отображения на экране текста, который пользователь вводил с клавиатуры. В графическом интерфейсе пользователя дисплей сам становится источником, откуда в машину вводится информация. Дисплей показывает различные графические объекты в виде картинок и конструкций для ввода информации, таких как кнопки или полосы прокрутки. Используя клавиатуру (или, что проще, устройство с указателем, например, мышь), пользователь может непосредственно манипулировать этими объектами на экране. Графические объекты можно перетаскивать, кнопки можно нажимать, полосы прокрутки можно прокручивать.

Взаимодействие между пользователем и программой становится, таким образом, более тесным. Вместо последовательного ввода информации с клавиатуры в программу и на дисплей, пользователь взаимодействует с объектами непосредственно на дисплее.

Содержимое интерфейса пользователя

Пользователям теперь не надо тратить слишком много времени на то, чтобы научиться пользоваться компьютером и составлять новые программы. Система Windows способствует этому, поскольку все программы для Windows выглядят и воспринимаются одинаково. Любая программа для Windows имеет окно — прямоугольную область на экране. Окно идентифицируется заголовком. Большинство функций программы запускается посредством меню. Слишком большой для экрана объем информации может быть просмотрен с помощью полос прокрутки. Некоторые пункты меню вызывают появление окон диалога, в которые пользователь вводит дополнительную информацию. Одно из окон диалога, имеющееся почти в каждой программе для Windows, предназначено для открытия файла. Это окно выглядит одинаково (или очень похоже) для множества различных программ для Windows и почти всегда вызывается с помощью одной и той же опции меню.

После того, как вы научились работать с одной программой для Windows, вам будет легко научиться работать с другой. Меню и окна диалога дают возможность пользователю экспериментировать с новой программой и исследовать ее свойства. Большинство программ для Windows поддерживают и интерфейс клавиатуры, и интерфейс манипулятора мышь. Хотя большинством функций программ для Windows можно управлять с помощью клавиатуры, в большинстве случаев проще пользоваться мышью.

С точки зрения программиста содержимое интерфейса пользователя, необходимое для создания меню и окон диалога, является результатом использования программ, уже встроенных в Windows. У всех меню один и тот же интерфейс клавиатуры и мыши, поскольку именно Windows, а не программа-приложение, управляет их работой.

Преимущество многозадачности

Хотя некоторые еще продолжают спрашивать о том, действительно ли так необходима многозадачность для компьютера с одним пользователем, те, кто постоянно работает с машиной, определенно подготовлены к многозадачности и могут почувствовать ее выгоды. Популярность резидентных программ MS-DOS, таких, например, как Sidekick, доказала это много лет назад. Хотя резидентные программы не являются, строго говоря, многозадачными программами, они позволяют осуществлять быстрое переключение контекста. Такое переключение контекста в принципе, основано на тех же концепциях, что и многозадачность.

Под Windows любая программа становится резидентной. Одновременно несколько программ Windows могут иметь вывод на экран и выполняться. Каждая программа занимает на экране прямоугольное окно. Пользователь может перемещать окна по всему экрану, менять их размер, переключаться между разными программами и передавать данные от одной программы к другой. Поскольку это отчасти напоминает рабочий стол (это, конечно, относится к тому времени, когда компьютеров было гораздо меньше, чем столов), о Windows иногда говорят, как о системе, использующей для вывода на экран нескольких программ, образ "рабочего стола" (desktop).

Первые версии Windows использовали многозадачность, названную "невытесняющей" или "кооперативной". Этот термин означал, что Windows не использовала системный таймер для распределения процессорного времени между разными программами, работающими в системе. Чтобы у них была возможность работать, программы сами, "добровольно" должны были отдавать управление. В Windows 95 многозадачность является вытесняющей, программы сами по себе могут иметь несколько потоков, которые, как кажется, выполняются параллельно.

Управление памятью

Операционная система не сможет реализовать многозадачность без управления памятью. Так как одни программы запускаются, а другие завершаются, память фрагментируется. Система должна быть способной объединять свободное пространство. Для этого требуется, чтобы система перемещала в памяти блоки программ и данных.

Даже Windows 1, работающая на процессоре 8088, была способна реализовать такой тип управления памятью. В реальном режиме это можно рассматривать только как пример поразительного искусства создателей программного обеспечения. Программы, работающие под Windows могут перераспределять память; размер программы может быть больше, чем размер оперативной памяти в каждый момент времени. Windows может удалить часть кодов выполняемой программы из памяти, а позднее вновь загрузить эти коды из EXE-файла. Пользователь может запустить несколько копий, называемых "экземплярами" программы; и у всех этих экземпляров в памяти оказывается совместно используемый ими код программы. Программы, запущенные в Windows, могут использовать функции из других файлов, которые называются "динамически подключаемыми библиотеками" (DLL). Windows содержит механизм для связи программ во время их работы с функциями из динамически подключаемых библиотек. Сама по себе операционная система Windows по существу является набором динамически подключаемых библиотек.

Таким образом, даже Windows 1, несмотря на ограничение размера оперативной памяти емкостью 640 килобайт, эффективно работала, не требуя какой бы то ни было дополнительной памяти. Но на этом фирма Microsoft не остановилась: операционная система Windows 2 дала приложениям для Windows доступ к отображаемой памяти (EMS), а Windows 3, работая в защищенном режиме, дала приложениям для Windows доступ к 16 мегабайтам расширенной памяти. И наконец Windows 95 сняла эти прежние ограничения, предоставив 32-разрядную операционную систему с плоским адресным пространством.

Независимость графического интерфейса от оборудования

Windows — это графический интерфейс, и программы для Windows могут полностью использовать графику и форматированный текст как на дисплее, так и на принтере. Графический интерфейс не только более удобен для восприятия, но он может также обеспечить пользователю высококачественное отображение информации.

У программ, написанных для Windows, нет прямого доступа к аппаратной части устройств отображения информации, таких как экран и принтер. Вместо этого Windows включает в себя язык графического программирования, называемый графическим интерфейсом устройства (Graphics Device Interface, GDI), который облегчает создание графики и форматированного текста. Windows абстрагируется от конкретного устройства отображения информации. Программы, написанные для Windows, будут работать с любым типом дисплея и любым типом принтера, для которых имеется в наличии драйвер Windows. В программе нет необходимости задавать тип используемого в системе оборудования.

Установка на IBM PC интерфейса, независимого от устройства отображения информации, была для создателей Windows непростым делом. Конструкция PC была основана на принципе открытой архитектуры. Треть производителей аппаратуры для PC были ориентированы на производство периферийных устройств и создали множество их типов. Хотя и появилось несколько стандартов, общепринятые программы для PC должны были поддерживать каждую из множества очень разных конфигураций оборудования. Например, для программ текстовых редакторов MS-DOS было вполне обычно продавать их вместе с одной или двумя дискетами с небольшими файлами, каждый из которых предназначался для поддержки отдельного принтера. В Windows 95 эти драйверы не нужны, поскольку подобная функция поддерживается самой операционной системой.

Соглашения операционной системы Windows

Программирование для Windows 95 — это реализация принципа: все или ничего. Например, вы не сможете написать приложение для MS-DOS и при этом использовать Windows только для создания какой-нибудь графики. Если вы собираетесь использовать любую часть Windows, то вынуждены смириться с необходимостью написания полнокровной программы для Windows.

Смысл этого станет более понятен после того, как вы больше узнаете о структуре программ для Windows. В Windows все взаимосвязано. Если вы захотите переместить на дисплее какую-нибудь графическую картинку, то вам нужно получить соответствующий "описатель контекста устройства". Чтобы это сделать, вам нужен "описатель окна". А для этого вы должны создать окно и подготовиться к получению "сообщений" для окна. Чтобы получить и обработать сообщение, необходима "процедура окна". Таким образом пишется программа для Windows. Не оторвавшись от земли нельзя взлететь.

Вызовы функций

Windows 95 в настоящее время поддерживает свыше тысячи вызовов функций, которые можно использовать в приложениях. Вряд ли вы когда-нибудь сможете запомнить синтаксис всех этих вызовов. Большинство программистов, пишущих программы для Windows, тратят массу времени на поиски различных вызовов функций в отпечатанных первоисточниках или в системах контекстно-зависимой подсказки.

Каждая функция Windows имеет развернутое имя, написанное буквами как верхнего, так и нижнего регистров, например *CreateWindow*. Эта функция (как вы могли бы догадаться) создает для вашей программы окно. Другой пример: функция *IsClipboardFormatAvailable* определяет, хранятся ли в буфере обмена данные специального формата.

Все основные функции Windows объявляются в заголовочных файлах. Главный заголовочный файл называется WINDOWS.H, и в этом файле содержится множество ссылок на другие заголовочные файлы. Эти заголовочные файлы имеются в любой среде программирования, поддерживающей Windows 95 и основанной на использовании языка С. Заголовочные файлы являются важной частью технической документации для Windows. Вы можете распечатать копии заголовочных файлов или для скорости воспользоваться программой просмотра файлов.

В программе для Windows вы используете вызовы функций Windows примерно также, как использовали библиотечные функции С, например *strlen*. Основное отличие в том, что код библиотечных функций С связывается с кодом вашей программы, тогда как код функций Windows остается вне вашей программы в динамически подключаемых библиотеках (DLL).

Когда вы запускаете программу в Windows, она взаимодействует с Windows через процесс, называемый "динамическим связыванием". EXE-файлы Windows содержат ссылки на различные динамически подключаемые библиотеки, функции которых в них используются. Большая часть этих библиотек DLL расположено в подкаталоге SYSTEM вашего каталога Windows. Когда программа для Windows загружается в оперативную память, вызовы в программе настраиваются на точки входа функций в динамически подключаемых библиотеках, которые, если этого еще не произошло, тоже загружаются в оперативную память.

Когда вы компонуете программу для Windows, чтобы сделать ее исполняемой, вам необходимо компоновать ее с "библиотеками импорта", поставляемыми в составе вашей среды программирования. Библиотеки импорта содержат имена всех функций Windows из динамически подключаемых библиотек и ссылки на них. Компоновщик использует эту информацию для создания в EXE-файле таблицы, которую Windows использует при загрузке программы для настройки адресов функций Windows.

Объектно-ориентированное программирование

При программировании для Windows вы фактически занимаетесь одним из видов объектно-ориентированного программирования (Object Oriented Programming, OOP). Это наиболее очевидно для объекта, с которым вы в Windows будете большей частью работать, объекта, который дал Windows ее название, объекта, который, как кажется, вскоре приобретет человеческие свойства, объекта, который, быть может, даже будет вам мерещиться, объекта, который известен как "окно".

Как уже упоминалось, окна — это прямоугольные области на экране. Окно получает информацию от клавиатуры или мыши пользователя и выводит графическую информацию на своей поверхности.

Окно приложения обычно содержит заголовок (title bar), меню (menu), рамку (sizing border) и иногда полосы прокрутки (scroll bars). Окна диалога — это дополнительные окна. Больше того, в окне диалога всегда имеется еще несколько окон, называемых "дочерними" (child windows). Эти дочерние окна имеют вид кнопок (push buttons), переключателей (radio buttons), флажков (check boxes), полей текстового ввода или редактирования (text entry fields), списков (list boxes) и полос прокрутки (scroll bars).

Пользователь рассматривает окна на экране в качестве объектов и непосредственно взаимодействует с этими объектами, нажимая кнопки и переключатели, передвигая бегунок на полосах прокрутки. Достаточно интересно, что положение программиста аналогично положению пользователя. Окно получает от пользователя информацию в виде оконных "сообщений". Кроме этого окно обменивается сообщениями с другими окнами.

Понимание этих сообщений — это один из барьеров, которые нужно преодолеть, чтобы стать программистом для Windows.

Архитектура, управляемая событиями

Когда автор впервые увидел работу графического интерфейса пользователя, она поставила его в тупик. Демонстрация проводилась средствами устаревшего текстового редактора, работающего в окне. Всякий раз при изменении размеров окна, текстовый редактор переформатировал содержащийся в нем текст.

Было понятно, что сама операционная система управляет элементами логики изменения размеров окна, и что программа способна реагировать на эту функцию системы. Но как программа *узнавала*, что размер ее окна изменился? Какой механизм использовала операционная система, чтобы довести эту информацию до окна? Для того чтобы понять, как все это работает, прежнего опыта программирования оказывается недостаточно.

Отсюда следует, что ответ на эти вопросы является центральным в понимании архитектуры, использованной в графическом интерфейсе пользователя. В Windows, когда пользователь меняет размер окна, программе отправляется сообщение с данными о новом размере окна. После этого программа может поменять размеры своего окна на новые.

"Windows посылает программе сообщение". Трудно надеяться, что вы смогли, не моргнув глазом, прочесть это предложение. Что оно могло бы означать? Мы здесь говорим о программе, а не о системе электронной почты. Как может операционная система отправить программе сообщение?

Когда говорится: "Windows посылает программе сообщение," — имеется в виду, что Windows вызывает функцию внутри программы. Параметры этой функции описывают параметры сообщения. Эта функция, находящаяся в вашей программе для Windows, называется оконной процедурой (window procedure).

Оконная процедура

Вы, несомненно, уже привыкли к мысли, что программа делает вызовы операционной системы. Таким образом, например, программа открывает файл на жестком диске. К чему вы, наверное, еще не можете привыкнуть, так это к тому, что операционная система вызывает программу. Тем не менее, это суть объектно-ориентированной архитектуры Windows 95.

У каждого окна, создаваемого программой, имеется соответствующая оконная процедура. Эта процедура является функцией, которая может находиться либо в самой программе, либо в динамически подключаемой библиотеке. Windows посылает сообщение окну путем вызова оконной процедуры, на основе этого сообщения окно совершает какие-то действия и затем возвращает управление Windows.

Более точно, окно всегда создается на основе "класса окна". Класс окна определяет оконную процедуру, обрабатывающую поступающие окну сообщения. Использование класса окна позволяет создавать множество окон на основе одного и того же класса окна и, следовательно, использовать одну и ту же оконную процедуру. Например, все кнопки во всех программах для Windows созданы на основе одного и того же класса окна. Этот класс связан с оконной процедурой (расположенной в динамически подключаемой библиотеке Windows), которая управляет процессом передачи сообщений всем кнопкам всех окон.

В объектно-ориентированном программировании любой "объект" несет в себе сочетание кода и данных. Окно — это объект. Код — это оконная процедура. Данные — это информация, хранимая оконной процедурой, и информация, хранимая системой Windows для каждого окна и каждого класса окна, которые имеются в системе.

Оконная процедура обрабатывает сообщения, поступающие окну. Очень часто эти сообщения передают окну информацию о том, что пользователь осуществил ввод с помощью клавиатуры или мыши. Таким образом, например, кнопки "узнают" о том, что они нажаты. Другие сообщения говорят окну о том, что необходимо изменить размер окна или о том, что поверхность окна необходимо перерисовать.

Когда программа для Windows начинает выполняться, Windows строит для программы очередь сообщений (message queue). В этой очереди хранятся сообщения для любых типов окон, которые могли бы быть созданы программой. Небольшая часть программы, которая называется циклом обработки сообщений (message loop), выбирает эти сообщения из очереди и переправляет их соответствующей оконной процедуре. Другие сообщения отправляются непосредственно оконной процедуре, минуя очередь сообщений.

Если это слишком абстрактное описание архитектуры Windows начало вас утомлять, может быть вам станет понятнее, если вы увидите, как окно, класс окна, оконная процедура, очередь сообщений, цикл обработки сообщений и сами сообщения собраны все вместе в тексте реальной программы.

Начнем.

Ваша первая программа для Windows

В своем первом классическом труде *The C Programming Language* (2d ed., Prentice Hall, 1988), Брайан Керниган и Деннис Ритчи начали изучение С с этой, важной для нас, программы, которую они назвали "Hello, world":

```
#include <stdio.h>

main()
{
       printf("Hello, world\n");
}
```

В этой главе будет показана аналогичная программа, написанная для Microsoft Windows 95. Программа называется HELLOWIN, она создает окно, в котором выводится строка "Hello, Windows 95!" и воспроизводится звуковой файл с голосом, декламирующим те же слова.

Чтобы вас не хватил удар при виде программы HELLOWIN, предупреждаем заранее, что в тексте программы HELLOWIN.C свыше 80 строк. Большая часть этих 80 строк является надстройкой. Похожая надстройка будет почти в каждой программе для Windows.

Вместо того, чтобы спрашивать, почему программа "Hello, Windows 95!" столь длинная и сложная, давайте зададимся вопросом о том, почему привычная программа "Hello, world" столь короткая и простая.

Что в этой программе неправильно?

Модель вывода строки в программе "Hello, world" и в других традиционных программах C — это устаревший придаток аппарата, известного как телетайп. Телетайп напоминает пишущую машинку с непрерывной подачей бумаги. Прошло не слишком много времени, с тех пор как программисты сидели за телетайпом и набирали команды, которые воспроизводились на бумаге. Компьютер отзывался, печатая свои ответы на той же бумаге.

В начале, после появления терминалов мэйнфрэймов и персональных компьютеров, принцип телетайпа распространился и на экран дисплея. Экран дисплея стал "стеклянным телетайпом", который просто прокручивался, если текст доходил до нижней части экрана.

Как может традиционная программа "Hello, world" выводить свой текст на экран без получения операционной системой информации о конкретном устройстве вывода, на котором этот текст должен появиться? Очевидно, что это дисплей — единственное устройство вывода, используемое таким образом, как будто оно является телетайпом. В том случае, если пользователь хочет вывести информацию куда-нибудь еще, ему необходимо задать это в командной строке.

Как может программа выводить свой текст на экран без получения системой информации о том, где на устройстве вывода этот текст должен появиться? Поскольку текст всегда появляется там, где оказывается курсор, то вероятно, текст после выполнения программы окажется на следующей строке. В том случае, если вы хотите поместить слова "Hello, world" в центр экрана, то вам следует перевести начальную позицию курсора в нужное положение, воспользовавшись несколькими управляющими командами, конкретный вид которых зависит от используемого устройства вывода.

Давайте посмотрим, что появится на экране, если бы вы захотели одновременно выполнить несколько программ "Hello, world". Полная неразбериха! Копии программ стали бы мешать друг другу. В заложенном в основу телетайпа принципе нет ничего, что разделяло бы несколько работающих параллельно программ.

Следует также отметить, что вы видите слова "Hello, world" даже после того, как программа завершилась. Вместо того, чтобы их стереть, программа оставляет на экране пережиток своего существования.

Программа "Hello, world" выглядит так просто потому, что написана в простое время, для простых компьютеров и простых устройств вывода информации. В мире современных компьютеров произошли значительные изменения, и эти изменения диктуют создателям программного обеспечения новые правила игры.

Файлы HELLOWIN

Два из трех файлов, необходимых для создания программы "HELLOWIN", представлены на рис. 2.1. Это make-файл HELLOWIN.MAK и файл исходного текста HELLOWIN.C.

HELLOWIN.MAK

```
#------------------------
# HELLOWIN.MAK make file
#------------------------

hellowin.exe : hellowin.obj
     $(LINKER) $(GUIFLAGS) -OUT:hellowin.exe hellowin.obj $(GUILIBS)

hellowin.obj : hellowin.c
     $(CC) $(CFLAGS) hellowin.c
```

HELLOWIN.C

```
/*------------------------------------------------------------
   HELLOWIN.C -- Displays "Hello, Windows 95!" in client area
                 (c) Charles Petzold, 1996
  ------------------------------------------------------------*/
```

```
#include <windows.h>

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow)
{
        static char szAppName[] = "HelloWin";
        HWND        hwnd;
        MSG         msg;
        WNDCLASSEX  wndclass;

        wndclass.cbSize        = sizeof(wndclass);
        wndclass.style         = CS_HREDRAW | CS_VREDRAW;
        wndclass.lpfnWndProc   = WndProc;
        wndclass.cbClsExtra    = 0;
        wndclass.cbWndExtra    = 0;
        wndclass.hInstance     = hInstance;
        wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
        wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
        wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
        wndclass.lpszMenuName  = NULL;
        wndclass.lpszClassName = szAppName;
        wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

        RegisterClassEx(&wndclass);

        hwnd = CreateWindow(
                        szAppName,              // window class name
                        "The Hello Program",    // window caption
                        WS_OVERLAPPEDWINDOW,    // window style
                        CW_USEDEFAULT,          // initial x position
                        CW_USEDEFAULT,          // initial y position
                        CW_USEDEFAULT,          // initial x size
                        CW_USEDEFAULT,          // initial y size
                        NULL,                   // parent window handle
                        NULL,                   // window menu handle
                        hInstance,              // program instance handle
                        NULL
        );                                      // creation parameters

        ShowWindow(hwnd, iCmdShow);
        UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
        HDC         hdc;
        PAINTSTRUCT ps;
        RECT        rect;

        switch(iMsg)
        {
        case WM_CREATE:
                PlaySound("hellowin.wav", NULL, SND_FILENAME | SND_ASYNC);
                return 0;
```

```
        case WM_PAINT:
                hdc = BeginPaint(hwnd, &ps);

                GetClientRect(hwnd, &rect);

                DrawText(hdc, "Hello, Windows 95!", -1, &rect, DT_SINGLELINE | DT_CENTER | DT_VCENTER);

                EndPaint(hwnd, &ps);
                return 0;

        case WM_DESTROY:
               PostQuitMessage(0);
               return 0;
        }

        return DefWindowProc(hwnd, iMsg, wParam, lParam);
}
```

Рис. 2.1 Программа HELLOWIN

В главе 9 вы встретите другой тип файла, широко распространенный в программировании для Windows, который называется файлом описания ресурсов (resource script) и имеет расширение .RC. Но до тех пор для большинства простых программ будут использоваться только make-файл, файл с исходным текстом на языке С, а также, возможно, заголовочный файл.

Как уже упоминалось, большая часть файла HELLOWIN.C является надстройкой, которую можно обнаружить практически в каждой программе для Windows. В действительности никто полностью не запоминает текст этой надстройки; подавляющее большинство программистов, когда пишут программу для Windows, просто копируют существующую программу и делают в ней необходимые изменения. Вы можете, используя прилагаемую дискету, поступать аналогичным образом.

Если на вашем компьютере имеется операционная система Windows 95, инсталлирован пакет Microsoft Visual C++ 4.0, выполнены пакетные (batch) файлы VCVARS32.BAT, входящие в состав Visual C++, и MSC.BAT, описанный в главе 1, то введя из командной строки MS-DOS:

```
NMAKE HELLOWIN.MAK
```

вы должны, таким образом, создать файл HELLOWIN.EXE.

Если все идет нормально, вы можете просто запустить программу из командной строки MS-DOS, введя:

```
HELLOWIN
```

Программа создает обычное окно приложения, как показано на рис. 2.2. В окне, в центре рабочей области, выводится текст "Hello, Windows 95!". Если у вас установлена звуковая плата, вы также услышите звуковое сообщение. (Если у вас нет звуковой платы, то чего вы собственно ждете?)

Обратите внимание, что это окно предлагает просто потрясающее количество возможностей для своих 80 строк программы. Вы можете захватить указателем мыши заголовок окна и перемещать его по всему экрану. Вы можете захватить рамку окна и изменить размеры. При изменении размеров окна программа будет автоматически перемещать строку текста "Hello, Windows 95!" в новый центр рабочей области окна. Вы можете щелкнуть на кнопке развертывания окна и увеличить HELLOWIN до размеров всего экрана. Вы можете щелкнуть на кнопке свертывания окна и стереть его с экрана. Вы можете вызвать все эти действия из системного меню. Вы также можете, чтобы завершить программу, закрыть тремя разными способами окно: выбрав соответствующую опцию из системного меню, щелкнув на кнопке закрытия окна справа в строке заголовка, или дважды щелкнув на иконке слева в строке заголовка.

Отрадно видеть, что HELLOWIN имеет все возможности обычной программы для Windows, но настроение может резко измениться, когда вы изучите исходный код, необходимый для создания этой программы. Тем не менее не пугайтесь и держите себя в руках, пока не проанализируете всю программу до конца строку за строкой.

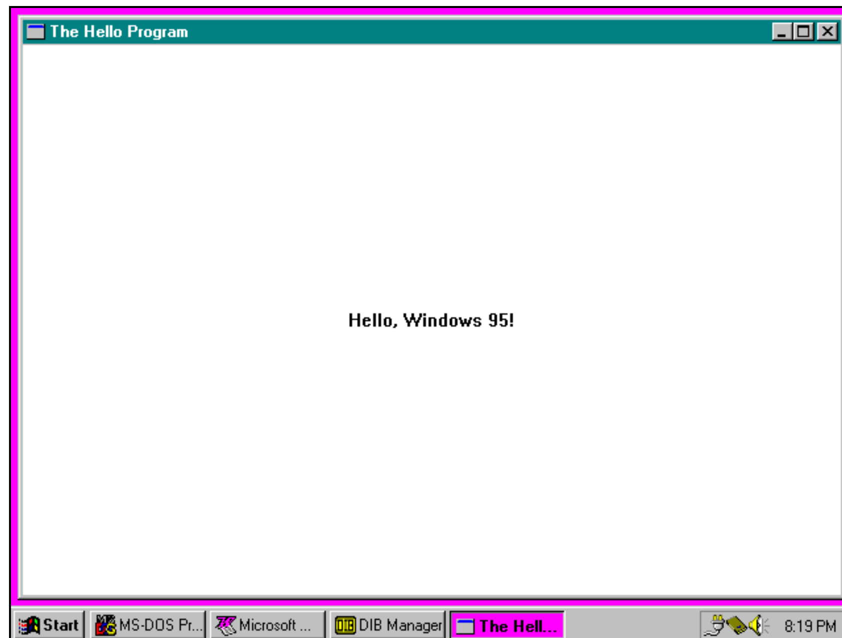


Рис. 2.2 Программа HELLOWIN, работающая в Windows 95

Make-файл

Для облегчения процесса компиляции программ для Windows, вы можете пользоваться утилитой NMAKE, поставляемой вместе с Microsoft Visual C++ 4.0. Если вы захотите что-нибудь изменить в файле с исходным текстом программы HELLOWIN.C, то все, что вам нужно сделать для создания нового исполняемого файла HELLOWIN.EXE — это запустить утилиту NMAKE так, как это было показано выше.

Make-файл состоит из одного или более разделов, каждый из которых начинается со строки, которая, в свою очередь, начинается с написания целевого файла (target file), потом идет двоеточие и далее перечисляются один или несколько файлов-источников (dependent file), из которых в итоге образуется целевой файл. За этой строкой следуют, с красной строки, одна или несколько командных строк. Эти команды собственно и предназначены для создания результирующего файла из файлов-источников. Если дата или время последней модификации любого из файлов-источников оказывается более поздней, чем дата или время последней модификации результирующего файла, то утилита NMAKE выполняет командные строки.

Обычно NMAKE модифицирует только тот результирующий файл, который расположен в первом разделе make-файла. Однако, если один из файлов-источников в другом разделе make-файла тоже оказывается результирующим файлом, то NMAKE первым модифицирует такой результирующий файл.

В make-файле HELLOWIN.MAK имеется два раздела. Если HELLOWIN.OBJ был изменен позже, чем HELLOWIN.EXE, то командная строка первого раздела запускает компоновщик. Если HELLOWIN.C был изменен позже, чем HELLOWIN.OBJ, то командная строка второго раздела запускает компилятор языка С. Поскольку HELLOWIN.OBJ в первом разделе является файлом-источником make-файла, а во втором разделе результирующим файлом, то утилита NMAKE, перед созданием нового файла HELLOWIN.EXE, проверит необходимость модификации HELLOWIN.OBJ. Таким образом, make-файл выполняется, фактически, снизу вверх. Вследствие работы компилятора языка С, из файла с исходным текстом программы HELLOWIN.C создается объектный модуль HELLOWIN.OBJ. Вследствие работы компоновщика, из объектного модуля HELLOWIN.OBJ создается исполняемый файл HELLOWIN.EXE.

В главе 1 было рассказано, как макроопределения в make-файле обеспечиваются переменными окружения, задаваемыми пакетными файлами, о которых там же шла речь. Это, по большей части, подключаемый набор различных флагов компилятора и имена библиотек компоновщика, поэтому, если вы хотите изучить их подробней, вернитесь к соответствующему разделу главы 1.

Файл исходного текста программы на языке С

Вторым файлом, показанным на рис. 2.1, является файл исходного текста программы HELLOWIN.C. Определение того, что эта программа написана на языке программирования С, может отнять у вас некоторое время!

Перед тем как заняться деталями, давайте рассмотрим HELLOWIN.C в целом. В файле имеется только две функции: *WinMain* и *WndProc*. *WinMain* — это точка входа в программу. Это аналог стандартной функции main языка С. В любой программе для Windows имеется функция *WinMain*.

WndProc — это "оконная процедура" для окна HELLOWIN. Каждое окно, независимо от того, является ли оно большим, как главное окно приложения для Windows, или маленьким, как кнопка, имеет соответствующую оконную процедуру. Оконная процедура — это способ инкапсулирования кода, отвечающего за ввод информации (обычно с клавиатуры или мыши) и за вывод информации на экран. Оконная процедура делает это, посылая "сообщения" окну. Не беспокойтесь о том, как именно это происходит. Позже у вас будет масса времени для того, чтобы попытаться решить эту проблему.

В HELLOWIN.C отсутствуют инструкции для непосредственного вызова *WndProc*: *WndProc* вызывается только из Windows. Однако, в *WinMain* имеется ссылка на *WndProc*, поэтому эта функция описывается в самом начале программы, еще до определения *WinMain*.

Вызовы функций Windows

HELLOWIN вызывает не менее 17 функций Windows. Здесь перечислены эти функции в порядке их появления в программе (с кратким описанием каждой функции):

- *LoadIcon* — загружает значок для использования в программе.
- *LoadCursor* — загружает курсор мыши для использования в программе.
- *GetStockObject* — получает графический объект (в этом случае для закрашивания фона окна используется кисть).
- *RegisterClassEx* — регистрирует класс окна для определенного окна программы.
- *CreateWindow* — создает окно на основе класса окна.
- *ShowWindow* — выводит окно на экран.
- *UpdateWindow* — заставляет окно перерисовать свое содержимое.
- *GetMessage* — получает сообщение из очереди сообщений.
- *TranslateMessage* — преобразует некоторые сообщения, полученные с помощью клавиатуры.
- *DispatchMessage* — отправляет сообщение оконной процедуре.
- *PlaySound* — воспроизводит звуковой файл.
- *BeginPaint* — инициирует начало процесса рисования окна.
- *GetClientRect* — получает размер рабочей области окна.
- *DrawText* — выводит на экран строку текста.
- *EndPaint* — прекращает рисование окна.
- *PostQuitMessage* — вставляет сообщение "завершить" в очередь сообщений.
- *DefWindowProc* — выполняет обработку сообщений по умолчанию.

Эти функции описаны в документации или системе контекстной подсказки, поставляемой с вашим компилятором, а описаны они в различных заголовочных файлах из WINDOWS.H.

Идентификаторы, написанные прописными буквами

В дальнейшем вы обратите внимание на использование в HELLOWIN.H нескольких идентификаторов, полностью написанных прописными буквами. Эти идентификаторы задаются в заголовочных файлах Windows. Некоторые из этих идентификаторов содержат двухбуквенный или трехбуквенный префикс, за которым следует символ подчеркивания:

CS_HREDRAW	DT_VCENTER	WM_CREATE
CS_VREDRAW	IDC_ARROW	WM_DESTROY
CW_USEDEFAULT	IDI_APPLICATION	WM_PAINT
DT_CENTER	SND_ASYNC	WS_OVERLAPPEDWINDOW
DT_SINGLELINE	SND_FILENAME	

Это просто числовые константы. Префикс показывает основную категорию, к которой принадлежат константы, как показано в данной таблице:

Префикс	Категория
CS	Опция стиля класса
IDI	Идентификационный номер иконки
IDC	Идентификационный номер курсора
WS	Стиль окна
CW	Опция создания окна
WM	Сообщение окна
SND	Опция звука
DT	Опция рисования текста

Программируя для Windows, почти никогда не нужно запоминать числовые константы. Фактически для любой числовой константы, которая используется в Windows, в заголовочных файлах имеется идентификатор.

Новые типы данных

Несколько других идентификаторов в HELLOWIN.C являются новыми типами данных; они также определяются в заголовочных файлах с помощью либо инструкций *typedef*, либо инструкций *#define*. Это изначально сделано для облегчения перевода программ для Windows с исходной 16-разрядной системы на будущие операционные системы, которые могли бы быть основаны на 32-разрядной (или иной) технологии. Эта работа не была столь гладкой и очевидной, как тогда многие думали, но идея оказалась основательной.

Иногда эти новые типы данных вполне условны. Например, тип данных UINT, использованный в качестве второго параметра *WndProc* — это просто беззнаковое целое, которое в Windows 95 является 32-разрядным. Тип данных PSTR, использованный в качестве третьего параметра *WinMain*, является указателем на строку символов, т. е. *char**.

Другие имена менее очевидны. Например, третий и четвертый параметры *WndProc* определяются как WPARAM и LPARAM соответственно. Происхождение этих имен требует небольшого экскурса в историю. Когда Windows была 16-разрядной системой, третий параметр *WndProc* определялся как WORD, что означало 16-разрядное *беззнаковое короткое* целое, а четвертый параметр определялся как LONG, что означало 32-разрядное знаковое *длинное* целое, и в этом смысл префиксов "W" и "L" у слова "PARAM". В Windows 95 имя WPARAM определяется как UINT, а LPARAM как LONG (что представляет из себя просто тип данных *длинное* целое языка С), следовательно оба параметра оконной процедуры являются 32-разрядными. Это может оказаться несколько неудобным, поскольку тип данных WORD в Windows 95 по-прежнему определяется как 16-разрядное *беззнаковое короткое* целое, следовательно префикс "W" у слова "PARAM" создает некоторую путаницу.

Функция *WndProc* возвращает значение типа LRESULT. Оно определено просто как LONG. Функция *WinMain* получает тип WINAPI (как и любая другая функция Windows, которая определяется в заголовочных файлах), а функция *WndProc* получает тип CALLBACK. Оба эти идентификатора определяются как stdcall, что является ссылкой на особую последовательность вызовов функций, которая имеет место между самой операционной системой Windows и ее приложением.

В HELLOWIN также использованы четыре структуры данных (о которых будет рассказано в конце этой главы), определяемых в заголовочных файлах Windows. Этими структурами данных являются:

Структура	Значение
MSG	Структура сообщения
WNDCLASSEX	Структура класса окна
PAINTSTRUCT	Структура рисования
RECT	Структура прямоугольника

Первые две структуры данных используются в *WinMain* для определения двух структур, названных *msg* и *wndclass*. Две вторые используются в *WndProc* для определения структур *ps* и *rect*.

Описатели

Наконец, имеется еще три идентификатора, которые пишутся прописными буквами и предназначены для разных типов описателей (handles):

Идентификатор	Значение
HINSTANCE	Описатель экземпляра (instance) самой программы
HWND	Описатель окна
HDC	Описатель контекста устройства

Описатели в Windows используются довольно часто. Перед тем как эта глава подойдет к концу, вы встретите описатель HICON (описатель иконки), описатель HCURSOR (описатель курсора мыши) и описатель HBRUSH (описатель графической кисти).

Описатель — это просто число (обычно длиной в 32 разряда), которое ссылается на объект. Описатели в Windows напоминают описатели файлов при программировании на традиционном C в MS-DOS. Программа почти всегда получает описатель путем вызова функции Windows. Программа использует описатель в других функциях Windows, чтобы сослаться на объект. Действительное значение описателя весьма важно для вашей программы, но модуль Windows, который обеспечивает программу описателем, "знает", как его использовать для ссылки на объект.

Венгерская нотация

Как вы могли заметить, некоторые переменные в HELLOWIN.C имеют своеобразные имена. Например, имя *szCmdLine* — параметр *WinMain*.

Многие программисты для Windows используют соглашения по именованию переменных, названные условно Венгерской нотацией, в честь легендарного программиста Microsoft Чарльза Симони. Все очень просто: имя переменной начинается со строчных буквы или букв, которые отмечают тип данных переменной. Например, префикс *sz* в *szCmdLine* означает, что строка завершается нулем (string terminated by zero). Префикс *h* в *hInstance* и *hPrevInstance* означает описатель (handle); префикс *i* в *iCmdShow* означает целое (integer). В двух последних параметрах *WndProc* также используется венгерская нотация, хотя, как уже говорилось раньше, *wParam* правильнее следовало бы назвать *uiParam* (беззнаковое целое — unsigned integer). Но поскольку эти два параметра определяются через типы данных WPARAM и LPARAM, было решено сохранить их прежние имена.

При обозначении переменных структуры удобно пользоваться именем самой структуры (или аббревиатурой имени структуры) и строчными буквами, используя их либо в качестве префикса имени переменной, либо как имя переменной в целом. Например, в функции *WinMain* в HELLOWIN.C переменная *msg* относится к структуре типа MSG; *wndclass* — к структуре типа WNDCLASSEX. В функции *WndProc*, переменная *ps* относится к структуре PAINTSTRUCT, *rect* — к RECT.

Венгерская нотация помогает избегать ошибок в программе еще до ее компоновки. Поскольку имя переменной описывает и саму переменную и тип ее данных, то намного снижается вероятность введения в программу ошибок, связанных с несовпадением типа данных у переменных.

В следующей таблице представлены префиксы переменных, которые в основном будут использоваться в этой книге:

Префикс	**Тип данных**
c	символ
by	BYTE (беззнаковый символ)
n	короткое целое
i	целое
x, y	целое (используется в качестве координат *x* и *y*)
cx, cy	целое (используется в качестве длины *x* и *y*), *c* означает "счет" — (count)
b или *f*	BOOL (булево целое); *f* означает "флаг" — (flag)
w	WORD (беззнаковое короткое целое)
l	LONG (длинное целое)
dw	DWORD (беззнаковое длинное целое)
fn	функция
s	строка
sz	строка, завершаемая нулем
h	описатель (handle)
p	указатель (pointer)

Точка входа программы

На этом глобальный обзор HELLOWIN.C заканчивается, и можно начать строку за строкой разбирать программу. Текст программы начинается с инструкции *#include*, которая позволяет включить в программу заголовочный файл WINDOWS.H:

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

WINDOWS.H включает в себя много других заголовочных файлов, содержащих объявления функций Windows, структур Windows, новые типы данных и числовые константы.

За инструкцией *#include* следует объявление *WndProc*:

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

Это объявление в начале программы необходимо потому, что в тексте функции *WinMain* имеются ссылки на функцию *WndProc*.

В программе на языке С, написанной для традиционной среды, точкой входа является функция *main*. С этого места программа начинает выполняться. (Фактически функция main является точкой входа в ту часть программы, которая пишется программистом. Обычно компилятор С должен вставить некоторый стартовый код в исполняемый файл. Этот код и вызывает функцию *main*.) Точкой входа программы для Windows является функция *WinMain*. *WinMain* всегда определяется следующим образом:

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow)
```

Эта функция использует последовательность вызовов WINAPI и, по своему завершению, возвращает операционной системе Windows целое. Функция называется *WinMain*. В ней есть четыре параметра.

Параметр *hInstance* называется описателем экземпляра (instance handle). Это уникальное число, идентифицирующее программу, когда она работает под Windows. Может так случиться, что пользователь запустит под Windows несколько копий одной и той же программы. Каждая копия называется "экземпляром" и у каждой свое значение *hInstance*. Описатель экземпляра можно сравнить с "идентификатором задачи" или "идентификатором процесса" — обычными терминами многозадачных операционных систем.

Параметр *hPrevInstance* — предыдущий экземпляр (previous instance) — в настоящее время устарел. В ранних версиях Windows он относился к самому последнему, предшествующему данному, описателю экземпляра той программы, которая все еще активна. Если в данный момент времени не было загружено никаких копий программы, то *hPrevInstance* = 0 или NULL. Под Windows 95 этот параметр всегда равен NULL.

Параметр *szCmdLine* — это указатель на оканчивающуюся нулем строку, в которой содержатся любые параметры, переданные в программу из командной строки. Вы можете запустить программу для Windows с параметром командной строки, вставив этот параметр после имени программы в командной строке MS-DOS или указать имя программы и параметр в окне диалога Run, которое вызывается из меню Start.

Параметр *iCmdShow* — число, показывающее, каким должно быть выведено на экран окно в начальный момент. Это число задается при запуске программы другой программой. Программисты достаточно редко обращаются к этому числу, но при необходимости такая возможность существует. В большинстве случаев число равно 1 или 7. Но лучше не думать об этом значении как о единице или как о семерке. Лучше думайте о них как об идентификаторе SW_SHOWNORMAL (заданном в заголовочных файлах Windows равным 1) или идентификаторе SW_SHOWMINNOACTIVE (заданном равным 7). Префикс SW в этих идентификаторах означает "показать окно" (show window). Параметр показывает, необходимо ли запущенную пользователем программу выводить на экран в виде окна нормального размера или окно должно быть изначально свернутым.

Регистрация класса окна

Окно всегда создается на основе класса окна. Класс окна идентифицирует оконную процедуру, которая выполняет процесс обработки сообщений, поступающих окну. Поскольку это важно, повторяем: окно всегда создается на основе класса окна. Класс окна идентифицирует оконную процедуру, которая выполняет процесс обработки сообщений, поступающих окну.

На основе одного класса окна можно создать несколько окон. Например, все окна-кнопки в Windows создаются на основе одного и того же класса окна. Класс окна определяет оконную процедуру и некоторые другие характеристики окон, создаваемых на основе этого класса. Когда вы создаете окно, вы определяете дополнительные характеристики окна, уникальные для него.

Перед созданием окна для вашей программы необходимо зарегистрировать класс окна путем вызова функции *RegisterClassEx*. Это расширенная (на что указывает окончание названия *Ex*, т. е. extended — расширенный) версия функции *RegisterClass* из предыдущих версий Windows. Тем не менее функция *RegisterClass* продолжает работать и под Windows 95.

У функции *RegisterClassEx* имеется один параметр: указатель на структуру типа WNDCLASSEX. Структура WNDCLASSEX определяется в заголовочных файла Windows следующим образом:

```
typedef struct tagWNDCLASSEX
{
       UINT          cbSize;
       UINT          style;
       WNDPROC       lpfnWndProc;
       int           cbClsExtra;
       int           cbWndExtra;
       HINSTANCE     hInstance;
       HICON         hIcon;
       HCURSOR       hCursor;
```

```
        HBRUSH         hbrBackground;
        LPCSTR         lpszMenuName;
        LPCSTR         lpszClassName;
        HICON          hIconSm;
} WNDCLASSEX;
```

Несколько замечаний о некоторых из представленных здесь типах данных и о венгерской нотации: префиксы *LP* и *lp* означают "длинный указатель" (long pointer), являющийся пережитком 16-разрядной Windows, в которой программисты могли различать короткие (или близкие, near) 16-разрядные указатели и длинные (или дальние, far) 32-разрядные указатели. В Windows 95 все указатели имеют длину в 32 разряда. В представленных в этой книге программах все префиксы l для типов указателей убраны, но несомненно вы встретите их где-нибудь в другом месте.

Обратите также внимание на некоторые новые случаи использования венгерской нотации: приставка *lpfn* означает "длинный указатель на функцию" (long pointer to a function). Приставка *cb* означает "счетчик байтов" (counter of bytes). Префикс *hbr* — это "описатель кисти" (handle to a brush).

В *WinMain* вы должны определить структуру типа WNDCLASSEX, обычно это делается следующим образом:

```
WNDCLASSEX wndclass;
```

Затем задаются 12 полей структуры и вызывается *RegisterClassEx*:

```
RegisterClassEx(&wndclass);
```

Наиболее важными являются второе от конца и третье поля. Второе от конца поле является именем класса окна (который в программах, создающих одно окно, обычно совпадает с именем программы). Третье поле (*lpfnWndProc*) является адресом оконной процедуры, которая используется для всех окон, созданных на основе этого класса (в HELLOWIN.C оконной процедурой является функция *WndProc*). Другие поля описывают характеристики всех окон, создаваемых на основе этого класса окна.

Поле *cbSize* равно длине структуры. Инструкция:

```
wndclass.style = CS_HREDRAW | CS_VREDRAW;
```

осуществляет объединение двух идентификаторов "стиля класса" (class style) с помощью поразрядной операции OR языка С. В заголовочных файлах Windows, идентификаторы, начинающиеся с префикса CS, задаются в виде 32-разрядной константы, только один из разрядов которой установлен в 1. Например, CS_VREDRAW задан как 0x0001, а CS_HREDRAW как 0x0002. Заданные таким образом идентификаторы иногда называют "поразрядными флагами" (bit flags). Объединяются поразрядные флаги с помощью операции OR языка С.

Эти два идентификатора стиля класса показывают, что все окна, созданные на основе данного класса должны целиком перерисовываться при изменении горизонтального (CS_HREDRAW) или вертикального (CS_VREDRAW) размеров окна. Если вы измените размер окна HELLOWIN, то увидите, что строка текста переместится в новый центр окна. Эти два идентификатора гарантируют, что это случится. Далее мы подробно рассмотрим, как оконная процедура уведомляется об изменении размера окна.

Третье поле структуры WNDCLASSEX инициализируется с помощью инструкции:

```
wndclass.lpfnWndProc = WndProc;
```

Эта инструкция устанавливает оконную *WndProc* как оконную процедуру данного окна, которая является второй функцией в HELLOWIN.C. Эта оконная процедура будет обрабатывать все сообщения всем окнам, созданным на основе данного класса окна. Как уже упоминалось, приставка *lpfn* означает "длинный указатель на функцию".

Следующие две инструкции:

```
wndclass.cbClsExtra = 0;

wndclass.cbWndExtra = 0;
```

резервируют некоторое дополнительное пространство в структуре класса и структуре окна, которое внутренне поддерживается операционной системой Windows. Программа может использовать это свободное пространство для своих нужд. В HELLOWIN эта возможность не используется, поэтому соответствующие значения равны 0. В противном случае, как следует из венгерской нотации, в этом поле было бы установлено "число байтов" резервируемой памяти.

В следующем поле находится просто описатель экземпляра программы (который является одним из параметров *WinMain*):

```
wndclass.hInstance = hInstance;
```

Инструкции:

```
wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
```

и

```
wndclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
```

устанавливают значок для всех окон, созданных на основе данного класса окна. Значок — это просто маленькая битовая картинка, которая появляется на панели задач Windows и слева в заголовке окна. Позже из этой книги вы узнаете, как создавать пользовательские значки для ваших программ под Windows. Сейчас для простоты воспользуемся стандартным значком.

Для получения описателя стандартного значка, вы вызываете *LoadIcon*, установив первый параметр в NULL. (При загрузке вашего собственного пользовательского значка, этот параметр должен быть установлен равным описателю экземпляра программы.) Второй идентификатор, начинающийся с префикса IDI ("идентификатор для значка" — ID for icon) определяется в заголовочных файлах Windows. Значок IDI_APPLICATION — это просто маленькое изображение окна. Функция *LoadIcon* возвращает описатель этого значка. Фактически нам не важно конкретное значение этого описателя. Оно просто используется для установки значений полей *wndclass.hIcon* и *wndclass.hIconSm*. Эти поля определяются в структуре WNDCLASSEX как поля типа HICON, что означает "описатель значка" (handle to an icon).

Инструкция:

```
wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
```

очень похожа на две предыдущие инструкции. Функция *LoadCursor* загружает стандартный курсор IDC_ARROW и возвращает описатель курсора. Этот описатель присваивается полю *hCursor* структуры WNDCLASSEX. Когда курсор мыши оказывается в рабочей области окна, созданного на основе данного класса, он превращается в маленькую стрелку.

Следующее поле задает цвет фона рабочей области окон, созданных на основе данного класса. Префикс *hbr* имени поля *hbrBackground* означает "описатель кисти" (handle to a brush). Кисть — это графический объект, который представляет собой шаблон пикселей различных цветов, используемый для закрашивания области. В Windows имеется несколько стандартных, или предопределенных (stock) кистей. Вызов *GetStockObject*, показанный здесь, возвращает описатель белой кисти:

```
wndclass.hbrBackground = GetStockObject(WHITE_BRUSH);
```

Это означает, что фон рабочей области окна будет плотного белого цвета, что является стандартным выбором.

Следующее поле задает меню класса окна. В приложении HELLOWIN меню отсутствует, поэтому поле установлено в NULL:

```
wndclass.lpszMenuName = NULL;
```

На последнем этапе классу должно быть присвоено имя. Для простой программы оно может быть просто именем программы, которым в нашем случае является строка "HelloWin", хранящаяся в переменной *szAppName*:

```
wndclass.lpszClassName = szAppName;
```

После того как инициализированы все 12 полей структуры, HELLOWIN регистрирует класс окна путем вызова функции *RegisterClassEx*. Единственным параметром функции является указатель на структуру WNDCLASSEX:

```
RegisterClassEx(&wndclass);
```

Создание окна

Класс окна определяет основные характеристики окна, что позволяет использовать один и тот же класс для создания множества различных окон. Когда вы, вызывая функцию *CreateWindow*, фактически создаете окно, вы, таким образом, детализируете информацию об окне.

Новички в программировании для Windows иногда путают понятия класс окна и окно, и не понимают, почему нельзя все характеристики окна задать за один раз. Фактически же, делить таким образом информацию очень удобно. Например, все окна-кнопки создаются на основе одного и того же класса окна. Оконная процедура, связанная с этим классом окна, находится в самой операционной системе Windows. Класс окна определяет для этих кнопок процесс ввода информации с клавиатуры, а также с помощью мыши, и одновременно регламентирует отображение кнопок на экране. Все кнопки, таким образом, работают одинаково. Но, в то же время, сами кнопки не одинаковы. Они могут отличаться по размеру, располагаться в разных местах экрана, иметь отличные друг от друга надписи. Эти последние характеристики и задаются при определении окна, а не класса окна.

Вместо использования структуры данных, как это делается в случае использования функции *RegisterClassEx*, вызов функции *CreateWindow* требует, чтобы вся информация передавалась функции в качестве параметров. Далее представлен вызов функции *CreateWindow* в HELLOWIN.C:

```
hwnd = CreateWindow(
       szAppName,                  //имя класса окна
       "The Hello Program",        //заголовок окна
       WS_OVERLAPPEDWINDOW,        //стиль окна
       CW_USEDEFAULT,              //начальное положение по x
       CW_USEDEFAULT,              //начальное положение по y
       CW_USEDEFAULT,              //начальный размер по x
       CW_USEDEFAULT,              //начальный размер по y
       NULL,                       //описатель родительского окна
       NULL,                       //описатель меню окна
       hInstance,                  //описатель экземпляра программы
       NULL
);                                 //параметры создания
```

Для удобства восприятия, использовались символ // и однострочные комментарии для описания параметров функции *CreateWindow*.

Параметр с комментарием "имя класса окна" — *szAppName* содержит строку "HelloWin", являющуюся именем только что зарегистрированного класса окна. Таким образом, этот параметр связывает окно с классом окна.

Окно, созданное нашей программой, является обычным перекрывающимся окном с заголовком, системным меню слева на строке заголовка, иконками для сворачивания, разворачивания и закрытия окна справа на строке заголовка и рамкой окна. Это стандартный стиль окон, он называется WS_OVERLAPPEDWINDOW и помечен комментарием "стиль окна". Комментарием "заголовок окна" отмечен текст, который появится в строке заголовка.

Параметры с комментариями "начальное положение по *x*" и "начальное положение по *y*" задают начальные координаты верхнего левого угла окна относительно левого верхнего угла экрана. Устанавливая для этих параметров идентификатор CW_USEDEFAULT, мы сообщаем Windows, что хотим использовать для перекрывающегося окна задаваемое по умолчанию начальное положение. (CW_USEDEFAULT задается равным 0x80000000.) По умолчанию Windows располагает следующие друг за другом перекрывающиеся окна, равномерно отступая по горизонтали и вертикали от верхнего левого угла экрана. Примерно также задают ширину и высоту окна параметры с комментариями "начальный размер по *x*" и "начальный размер по *y*". CW_USEDEFAULT снова означает, что мы хотим, чтобы Windows использовала задаваемый по умолчанию размер окна.

Параметр с комментарием "описатель родительского окна" устанавливается в NULL, поскольку у нашего окна отсутствует родительское окно. (Если между двумя окнами существует связь типа родительское-дочернее, дочернее окно всегда появляется только на поверхности родительского.) Параметр с комментарием "описатель меню окна" также установлен в NULL, поскольку у нашего окна нет меню. В параметр с комментарием "описатель экземпляра программы" помещается описатель экземпляра, переданный программе в качестве параметра функции *WinMain*. И наконец, параметр с комментарием "параметры создания" установлен в NULL. При необходимости этот параметр используется в качестве указателя на какие-нибудь данные, на которые программа в дальнейшем могла бы ссылаться.

Вызов *CreateWindow* возвращает описатель созданного окна. Этот описатель хранится в переменной *hwnd*, которая имеет тип HWND (описатель окна — handle to a window). У каждого окна в Windows имеется описатель. В вашей программе описатель используется для того, чтобы ссылаться на окно. Для многих функций Windows в качестве параметра требуется *hwnd*, благодаря этому Windows знает, к какому окну применить функцию. Если программа создает несколько окон, то каждое из них имеет свой описатель. Описатель окна — это один из важнейших описателей, которыми оперирует программа для Windows.

Отображение окна

К тому времени, когда функция *CreateWindow* возвращает управление программе, окно уже создано внутри Windows. Однако, на экране монитора оно еще не появилось. Необходимы еще два вызова. Первый из них:

```
ShowWindow(hwnd, iCmdShow);
```

Первым параметром является описатель только что созданного функцией *CreateWindow* окна. Вторым параметром является величина *iCmdShow*, передаваемая в качестве параметра функции *WinMain*. Он задает начальный вид окна на экране. Если *iCmdShow* имеет значение SW_SHOWNORMAL (т. е. 1), на экран выводится обычное окно. Если *iCmdShow* имеет значение SW_SHOWMINNOACTIVE (т. е. 7), то окно не выводится, а на панели задач появляются его имя и иконка.

Функция *ShowWindow* выводит окно на экран. Если второй параметр *ShowWindow* имеет значение SW_SHOWNORMAL, то фон рабочей области окна закрашивается той кистью, которая задана в классе окна. Вызов функции:

```
UpdateWindow(hwnd);
```

вызывает затем перерисовку рабочей области. Для этого в оконную процедуру (функция *WndProc* в HELLOWIN.C) посылается сообщение WM_PAINT. Вскоре мы изучим, как *WndProc* обрабатывает это сообщение.

Цикл обработки сообщений

После вызова функции *UpdateWindow*, окно окончательно выведено на экран. Теперь программа должна подготовить себя для получения информации от пользователя через клавиатуру и мышь. Windows поддерживает "очередь сообщений" (message queue) для каждой программы, работающей в данный момент в системе Windows. Когда происходит ввод информации, Windows преобразует ее в "сообщение", которое помещается в очередь сообщений программы.

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

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

Переменная *msg* — это структура типа MSG, которая определяется в заголовочных файлах Windows следующим образом:

```
typedef struct tagMSG
{
       HWND   hwnd;
       UINT   message;
       WPARAM wParam;
       LPARAM lParam;
       DWORD  time;
       POINT  pt;
} MSG;
```

Тип данных POINT — это тип данных другой структуры, которая определяется так:

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

Вызов функции *GetMessage*, с которого начинается цикл обработки сообщений, извлекает сообщение из очереди сообщений:

```
GetMessage(&msg, NULL, 0, 0)
```

Этот вызов передает Windows указатель на структуру *msg* типа MSG. Второй, третий и четвертый параметры, NULL или 0, показывают, что программа получает все сообщения от всех окон, созданных этой программой. Windows заполняет поля структуры сообщений информацией об очередном сообщении из очереди сообщений. Поля этой структуры следующие:

- *hwnd* — описатель окна, для которого предназначено сообщение. В программе HELLOWIN, он тот же, что и *hwnd*, являющийся возвращаемым значением функции *CreateWindow*, поскольку у нашей программы имеется только одно окно.
- *message* — идентификатор сообщения. Это число, которое идентифицирует сообщение. Для каждого сообщения имеется соответствующий ему идентификатор, который задается в заголовочных файлах Windows и начинается с префикса WM (оконное сообщение — window message). Например, если вы установите указатель мыши в рабочей области программы HELLOWIN и нажмете левую кнопку мыши, Windows поставит сообщение в очередь сообщений с полем *message* равным WM_LBUTTONDOWN, значение которого 0x0201.
- *wParam* — 32-разрядный параметр сообщения (message parameter), смысл и значение которого зависят от особенностей сообщения.
- *lParam* — другой 32-разрядный параметр, зависящий от сообщения.
- *time* — время, когда сообщение было помещено в очередь сообщений.
- *pt* — координаты курсора мыши в момент помещения сообщения в очередь сообщений.

Если поле *message* сообщения, извлеченного из очереди сообщений, равно любому значению, кроме WM_QUIT (т. е., 0x0012), то функция *GetMessage* возвращает ненулевое значение. Сообщение WM_QUIT заставляет программу прервать цикл обработки сообщений. На этом программа заканчивается, возвращая число *wParam* структуры *msg*.

Инструкция:

```
TranslateMessage(&msg);
```

передает структуру *msg* обратно в Windows для преобразования какого-либо сообщения с клавиатуры. (Более подробно об этом будет рассказано в главе 5.)

Инструкция:

```
DispatchMessage(&msg);
```

также передает структуру *msg* обратно в Windows. Windows отправляет сообщение для его обработки соответствующей оконной процедуре — таким образом, Windows вызывает оконную процедуру. Такой оконной процедурой в HELLOWIN является функция *WndProc*. После того, как *WndProc* обработает сообщение, оно возвращается в Windows, которая все еще обслуживает вызов функции *DispatchMessage*. Когда Windows возвращает управление в программу HELLOWIN к следующему за вызовом *DispatchMessage* коду, цикл обработки сообщений в очередной раз возобновляет работу, вызывая *GetMessage*.

Оконная процедура

Обо всем, о чем так долго говорилось, можно кратко сказать так: класс окна зарегистрирован, окно создано, окно выведено на экран, и для извлечения сообщений из очереди сообщений программа вошла в цикл обработки сообщений.

Реальная работа начинается в оконной процедуре, которую программисты обычно называют "window proc". Оконная процедура определяет то, что выводится в рабочую область окна и то, как окну реагировать на пользовательский ввод.

В программе HELLOWIN оконной процедурой является функция *WndProc*. Оконной процедуре можно назначить любое имя (любое, конечно, в той степени, в которой оно не будет конфликтовать с другими именами). В программе для Windows может содержаться более одной оконной процедуры. Оконная процедура всегда связана с определенным классом окна, который вы регистрируете, вызывая *RegisterClassEx*. Функция *CreateWindow* создает окно на основе определенного класса окна. На основе одного и того же класса можно создать несколько окон.

Оконная процедура всегда определяется следующим образом:

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
```

Отметьте, что четыре параметра оконной процедуры идентичны первым четырем полям структуры MSG.

Первым параметром является *hwnd*, описатель получающего сообщение окна. Это тот же описатель, который возвращает функция *CreateWindow*. Для программы типа HELLOWIN, в которой создается только одно окно, имеется только один известный программе описатель окна. Если же в программе создается несколько окон на основе одного и того же класса окна (и следовательно одной и той же оконной процедуры), тогда *hwnd* идентифицирует конкретное окно, которое получает сообщение.

Вторым параметром является число (точнее 32-разрядное беззнаковое целое или UINT), которое идентифицирует сообщение. Два последних параметра (*wParam* типа WPARAM и *lParam* LPARAM) представляют дополнительную информацию о сообщении. Они называются "параметрами сообщения" (message parameters). Конкретное значение этих параметров определяется типом сообщения.

Обработка сообщений

Каждое получаемое окном сообщение идентифицируется номером, который содержится в параметре *iMsg* оконной процедуры. В заголовочных файлах Windows определены идентификаторы, начинающиеся с префикса WM ("window message") для каждого типа сообщений.

Обычно программисты для Windows используют конструкции *switch* и *case* для определения того, какое сообщение получила оконная процедура и то, как его обрабатывать. Если оконная процедура обрабатывает сообщение, то ее возвращаемым значением должен быть 0. Все сообщения, не обрабатываемые оконной процедурой, должны передаваться функции Windows, которая называется *DefWindowProc*. Значение, возвращаемое функцией *DefWindowProc*, должно быть возвращаемым значением оконной процедуры.

В HELLOWIN функция *WndProc* обрабатывает только три сообщения: WM_CREATE, WM_PAINT и WM_DESTROY. Оконная процедура выглядит следующим образом:

```
switch(iMsg)
{
case WM_CREATE:
       [process WM_CREATE message]
       return 0;

case WM_PAINT:
       [process WM_PAINT message]
       return 0;

case WM_DESTROY:
       [process WM_DESTROY message]
       return 0;
}

return DefWindowProc(hwnd, iMsg, wParam, lParam);
```

Здесь важно отметить то, что вызов функции *DefWindowProc* обрабатывает по умолчанию все сообщения, которые не обрабатывает ваша оконная процедура.

Воспроизведение звукового файла

Самое первое сообщение, которое получает оконная процедура — и первое, которое обрабатывает функция *WndProc* — это WM_CREATE. *WndProc* получает это сообщение тогда, когда Windows обрабатывает функцию *CreateWindow* в *WinMain*. Таким образом, когда HELLOWIN вызывает *CreateWindow*, Windows делает то, что должна делать, т. е. Windows вызывает *WndProc* с описателем окна в качестве первого параметра и с WM_CREATE в качестве второго. *WndProc* обрабатывает сообщение WM_CREATE и передает управление обратно в Windows. Теперь Windows может вернуться после вызова *CreateWindow* обратно в HELLOWIN, чтобы продолжить работу в *WinMain*.

Часто оконная процедура выполняет разовую инициализацию окна, когда обрабатывается сообщение. WM_CREATE. HELLOWIN предпочитает обрабатывать это сообщение путем воспроизведения звукового файла HELLOWIN.WAV. Это делается с помощью функции *PlaySound*. Первым параметром этой функции является имя файла. Это также может быть другое имя (sound alias name), которое задается в секции Sounds панели управления (Control Panel) или определяется ресурсом программы. Второй параметр используется только при условии, что звуковой файл является ресурсом. Третий параметр задает две опции. В нашем случае, когда первый параметр — это имя файла, звук должен воспроизводиться асинхронно, т. е. функция *PlaySound* возвратит свое значение как только начнет воспроизводиться звуковой файл, не ожидая окончания воспроизведения.

WndProc завершает обработку WM_CREATE с нулевым возвращаемым значением.

Сообщение WM_PAINT

Сообщение WM_PAINT функция *WndProc* обрабатывает вторым. Это сообщение крайне важно для программирования под Windows. Оно сообщает программе, что часть или вся рабочая область окна недействительна (invalid), и ее следует перерисовать.

Как рабочая область становится недействительной? При первом создании окна недействительна вся рабочая зона, поскольку программа еще ничего в окне не нарисовала. Сообщение WM_PAINT (которое обычно посылается, когда программа вызывает *UpdateWindow* в *WinMain*) заставляет оконную процедуру что-то нарисовать в рабочей области.

Когда вы изменяете размер окна, рабочая область также становится недействительной. Вспомните, что в параметр *style* структуры *wndclass* программы HELLOWIN помещены флаги CS_HREDRAW и CS_VREDRAW. Они заставляют Windows при изменении размеров окна считать недействительным все окно. Затем оконная процедура получает сообщение WM_PAINT.

Когда вы минимизируете окно программы HELLOWIN, а затем снова его восстанавливаете до начального размера, то в Windows содержимое рабочей области не сохраняется. В графической среде это привело бы к тому, что пришлось бы хранить слишком много данных. Вместо этого, Windows делает недействительным все окно. Оконная процедура получает сообщение WM_PAINT и сама восстанавливает содержимое окна.

Когда вы перемещаете окна так, что они перекрываются, Windows не сохраняет ту часть окна, которая закрывается другим окном. Когда эта часть окна позже открывается, Windows помечает его как недействительное. Оконная процедура получает сообщение WM_PAINT для восстановления содержимого окна.

Обработка сообщения WM_PAINT почти всегда начинается с вызова функции *BeginPaint*:

```
hdc = BeginPaint(hwnd, &ps);
```

и заканчивается вызовом функции *EndPaint*:

```
EndPaint(hwnd, &ps);
```

В обеих функциях первый параметр — это описатель окна программы, а второй — это указатель на структуру типа PAINTSTRUCT. В структуре PAINTSTRUCT содержится некоторая информация, которую оконная процедура может использовать для рисования в рабочей области. (В следующей главе будет рассказано о полях этой структуры.)

При обработке вызова *BeginPaint*, Windows обновляет фон рабочей области, если он еще не обновлен. Обновление фона осуществляется с помощью кисти, заданной в поле *hbrBackground* структуры WNDCLASSEX, которая использовалась при регистрации класса окна. В случае нашей программы HELLOWIN подготовлена белая кисть и это означает, что Windows обновит фон окна, закрасив его белым цветом. Вызов *BeginPaint* делает всю рабочую область действительной (не требующей перерисовки) и возвращает описатель контекста устройства. Контекст устройства описывает физическое устройство вывода информации (например, дисплей) и его драйвер. Описатель контекста устройства необходим вам для вывода в рабочую область окна текста и графики. Используя описатель контекста устройства, возвращаемого функцией *BeginPaint*, вы не сможете рисовать вне рабочей области, даже не пытайтесь. Функция *EndPaint* освобождает описатель контекста устройства, после чего его значение нельзя использовать.

Если оконная процедура не обрабатывает сообщения WM_PAINT (что бывает крайне редко), они должны передаваться в *DefWindowProc*. Функция *DefWindowProc* просто по очереди вызывает *BeginPaint* и *EndPaint* и, таким образом, рабочая область устанавливается в действительное состояние, т. е. состояние, не требующее перерисовки.

После того, как *WndProc* вызвала *BeginPaint*, она вызывает *GetClientRect*:

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

Первый параметр — это описатель окна программы. Второй параметр — это указатель на переменную *rect*, для которой в *WndProc* задан тип RECT.

RECT — это структура "прямоугольник" (rectangle), определенная в заголовочных файлах Windows. Она имеет четыре поля типа LONG, имена полей: *left, top, right* и *bottom*. *GetClientRect* помещает в эти четыре поля размер рабочей области окна. Поля *left* и *top* всегда устанавливаются в 0. В полях *right* и *bottom* устанавливается ширина и высота рабочей области в пикселях.

WndProc никак не использует структуру RECT, за исключением передачи указателя на нее в качестве четвертого параметра функции *DrawText*:

```
DrawText(hdc, "Hello, Windows 95!", -1, &rect, DT_SINGLELINE | DT_CENTER | DT_VCENTER);
```

DrawText (как подсказывает ее имя) рисует текст. Поскольку эта функция что-то рисует, то первый параметр — это описатель контекста устройства, возвращенный функцией *BeginPaint*. Вторым параметром является рисуемый текст, а третий параметр установлен в —1, чтобы показать, что строка текста заканчивается нулевым символом.

Последний параметр — это набор флагов, значения которых задано в заголовочных файлах Windows. Флаги показывают, что текст следует выводить в одну строку, по центру относительно горизонтали и вертикали и внутри прямоугольной области, размер которой задан четвертым параметром. Вызов этой функции приводит, таким образом, к появлению строки "Hello, Windows 95!" в центре рабочей области.

Когда рабочая область становится недействительной (как это происходит при изменении размеров окна), *WndProc* получает новое сообщение WM_PAINT. Новый размер окна *WndProc* получает, вызвав функцию *GetClientRect*, и снова рисует текст в центре окна.

Сообщение WM_DESTROY

Еще одним важным сообщением является сообщение WM_DESTROY. Это сообщение показывает, что Windows находится в процессе ликвидации окна в ответ на полученную от пользователя команду. Пользователь вызывает поступление этого сообщения, если щелкнет на кнопке Close, или выберет Close из системного меню программы, или нажмет <Alt>+<F4>.

HELLOWIN стандартно реагирует на это сообщение, вызывая:

```
PostQuitMessage(0);
```

Эта функция ставит сообщение WM_QUIT в очередь сообщений программы. Как уже упоминалось, функция *GetMessage* возвращает ненулевое значение при любом сообщении, полученном из очереди сообщений за исключением WM_QUIT. Когда *GetMessage* получает сообщение WM_QUIT, функция возвращает 0. Это заставляет *WinMain* прервать цикл обработки сообщений и выйти в систему, закончив программу.

Сложности программирования для Windows

Даже с учетом авторских пояснений структура и принципы работы программы Hellowin, вполне вероятно, так и останутся для вас немного загадочными. В короткой программе на С, написанной для обычной среды, вся программа целиком может поместиться в функции *main*. В HELLOWIN *WinMain* содержит только надстройку программы, необходимую для регистрации класса окна, создания окна и получения и передачи сообщений из/в очередь сообщений.

Все основные действия программы происходят в оконной процедуре. В HELLOWIN этих действий немного — это просто воспроизведение звука и вывод в окно строки текста. Но в следующих главах вы обнаружите, что почти в каждой программе для Windows работа с сообщениями происходит в оконной процедуре. Это основная концептуальная особенность, которую вы должны усвоить, чтобы начать писать программы для Windows.

Не вызывай меня, я вызову тебя

Как уже упоминалось, программисты хорошо знакомы с понятием вызова операционной системы для выполнения каких-то действий. Например, программисты на С используют функцию *fopen* для открытия файла. Библиотечные функции, поставляемые с компилятором, содержат код, который фактически вызывает для открытия файла операционную систему. Здесь все просто.

Но операционная система Windows ведет себя иначе. Хотя в Windows имеется свыше тысячи доступных программисту функций, Windows также и сама посылает вызовы вашей программе, особенно оконной процедуре, которую мы назвали *WndProc*. Оконная процедура связана с классом окна, который программа регистрирует с помощью вызова функции *RegisterClassEx*. Окно, создаваемое на основе этого класса, использует оконную процедуру для обработки всех сообщений окна. Windows посылает сообщения окну, вызывая оконную процедуру.

Windows вызывает *WndProc* первый раз при создании окна. Windows вызывает *WndProc* при последующем удалении окна. Windows вызывает *WndProc* при изменении размеров окна, при его перемещении, при его свертывании. Windows вызывает *WndProc* при выборе пункта меню. Windows вызывает *WndProc* при манипуляциях с полосами прокрутки или с мышью. Windows вызывает *WndProc*, чтобы сообщить ей о необходимости перерисовать рабочую область.

Все эти вызовы имеют форму сообщений. В большинстве программ для Windows, основная часть программы направлена на обработку этих сообщений. Свыше 200 различных сообщений, которые Windows может отправить оконной процедуре, идентифицируются именами, которые начинаются с букв "WM" и определяются в заголовочных файлах Windows.

Фактически, идея функции, находящейся в программе, но которая вызывается не из самой программы, не является абсолютно новой в традиционном программировании. Функция *signal* в С может перехватить <Ctrl>+<Break>. Вы можете иметь опыт с перехватом аппаратных прерываний с помощью языка ассемблера или одной из конструкций ON в Microsoft BASIC. Драйвер мыши фирмы Microsoft позволяет работать с этой мышью программам, сделанным не для Windows.

В Windows эта идея расширена и пронизывает всю систему. Любое событие, относящееся к окну, передается оконной процедуре в виде сообщения. Затем оконная процедура соответствующим образом реагирует на это сообщение или передает сообщение в *DefWindowProc* для обработки его по умолчанию.

Параметры *wParam* и *lParam* оконной процедуры не используются в HELLOWIN кроме как параметры для *DefWindowProc*. Эти параметры дают оконной процедуре дополнительную информацию о сообщении. Значение этих параметров зависит от самого сообщения.

Давайте рассмотрим пример. Когда меняется размер рабочей области окна, Windows вызывает оконную процедуру. Параметр *hwnd* оконной процедуры — это описатель окна, изменившего размер. Параметр *iMsg* равен WM_SIZE. Параметр *wParam* для сообщения WM_SIZE равен одной из величин SIZENORMAL, SIZEICONIC, SIZEFULLSCREEN, SIZEZOOMSHOW или SIZEZOOMHIDE (определяемых в заголовочных файлах Windows как числа от 0 до 4). Параметр *wParam* показывает, будет ли окно свернуто, развернуто или скрыто (в результате развертывания другого окна). Параметр *lParam* определяет новый размер окна. Новая ширина (16-разрядное значение) и новая высота (16-разрядное значение) объединяются вместе в 32-разрядный параметр *lParam*. В заголовочных файлах Windows имеется макрос, который позволяет выделить оба эти значения из *lParam*. Мы это проделаем в следующей главе.

Иногда, в результате обработки сообщения функцией *DefWindowProc*, генерируются другие сообщения. Например, предположим, что вы запускаете HELLOWIN и выбираете Close из системного меню программы, используя клавиатуру или мышь. *DefWindowProc* обрабатывает эту информацию. Когда она определяет, что вы выбрали опцию Close, то отправляет сообщение WM_SYSCOMMAND оконной процедуре. *WndProc* передает это сообщение *DefWindowProc*. *DefWindowProc* реагирует на него, отправляя сообщение WM_CLOSE оконной процедуре. *WndProc* снова передает это сообщение *DefWindowProc*. *DefWindowProc* реагирует на сообщение

WM_CLOSE, вызывая функцию *DestroyWindow*. *DestroyWindow* заставляет Windows отправить сообщение WM_DESTROY оконной процедуре. И наконец, *WndProc* реагирует на это сообщение, вызывая функцию *PostQuitMessage* путем постановки сообщения WM_QUIT в очередь сообщений. Это сообщение прерывает цикл обработки сообщений в *WinMain* и программа заканчивается.

Синхронные и асинхронные сообщения

Ранее было рассказано о передаче окну сообщений, что означает вызов операционной системой Windows оконной процедуры. Но в программах для Windows имеется цикл обработки сообщений, который берет сообщения из очереди сообщений, вызывая функцию *GetMessage*, и отправляет их оконной процедуре, вызывая функцию *DispatchMessage*.

Так что же, буферизуются ли сообщения для Windows-программы (так же как в обычной программе буферизуется ввод с клавиатуры) и затем пересылаются дальше, или она (программа для Windows) получает сообщения непосредственно снаружи? И так, и этак.

Одни и те же сообщения могут быть и "синхронные" (queued), и "асинхронные" (nonqueued)[1]. Синхронными сообщениями называются сообщения, которые Windows помещает в очередь сообщений программы, и которые извлекаются и диспетчеризуются в цикле обработки сообщений. Асинхронные сообщения передаются непосредственно окну, когда Windows вызывает оконную процедуру. В результате оконная процедура получает все предназначенные для окна сообщения, как синхронные, так и асинхронные. Структура программ для Windows очень проста, поскольку у них имеется только одно центральное место обработки сообщений. Говорят, что синхронные сообщения помещаются в очередь сообщений (*post*), а асинхронные посылаются прямо в оконную процедуру (*send*).

Синхронными становятся сообщения, в основном, тогда, когда они являются результатом пользовательского ввода путем нажатия клавиш (например, WM_KEYDOWN и WM_KEYUP), это символы, введенные с клавиатуры (WM_CHAR), результат движения мыши (WM_MOUSEMOVE) и щелчков кнопки мыши (WM_LBOTTONDOWN). Кроме этого синхронные сообщения включают в себя сообщение от таймера (WM_TIMER), сообщение о необходимости плановой перерисовки (WM_PAINT) и сообщение о выходе из программы (WM_QUIT). Сообщения становятся асинхронными во всех остальных случаях. Часто асинхронные сообщения являются результатом синхронных. При передаче асинхронного сообщения в *DefWindowProc* из оконной процедуры, Windows часто обрабатывает сообщение, отправляя оконной процедуре другие асинхронные сообщения.

Очевидно, что процесс этот сложен, но к счастью большая часть сложностей ложится на Windows, а не на наши программы. С позиции оконной процедуры, эти сообщения проходят через нее упорядочено или синхронно. Оконная процедура может что-то сделать с этими сообщениями, а может и проигнорировать их. По этой причине оконную процедуру назвали "конечным пунктом обработки" (ultimate hook). Сообщения извещают оконную процедуру почти обо всем, что влияет на окно.

Часто асинхронные сообщения являются результатом вызова определенных функций Windows или непосредственным результатом вызова функции *SendMessage*. (Кроме этого, сообщения могут помещаться в очередь сообщений посредством вызова функции *PostMessage*.)

Например, когда *WinMain* вызывает функцию *CreateWindow*, Windows создает окно и для этого отправляет оконной процедуре асинхронное сообщение WM_CREATE. Когда *WinMain* вызывает *ShowWindow*, Windows отправляет оконной процедуре асинхронные сообщения WM_SIZE и WM_SHOWWINDOW. Когда *WinMain* вызывает *UpdateWindow*, Windows отправляет оконной процедуре асинхронное сообщение WM_PAINT.

Сообщения не похожи на аппаратные прерывания. Во время обработки в оконной процедуре одного сообщения программа не может быть прервана другим сообщением. Только в том случае, если оконная процедура вызвала функцию, которая сама стала источником нового сообщения, то оконная процедура начнет обрабатывать это новое сообщение еще до того, как функция вернет управление программе.

---

[1] Необходимо уточнить дальнейшее использование атрибутов сообщения "синхронное" и "асинхронное". Ставится сообщение в очередь или не ставится, определяется способом его отправки. В оригинальном тексте книги для указания конкретного способа отправки сообщения используются слова send (функция *SendMessage*) и post (функция *PostMessage*). Если для отправки сообщения используется функция *SendMessage*, то оно не ставится в очередь, оконная процедура вызывается непосредственно, а функция возвращает управление только после обработки сообщения оконной процедурой. Если для отправки сообщения используется функция *PostMessage*, то оно ставится в очередь, а функция возвращает управление немедленно.

Таким образом, используя терминологию автора, можно сказать:

- если сообщение отправляется с помощью функции *SendMessage*, то оно является асинхронным;
- если сообщение отправляется с помощью функции *PostMessage*, то оно является синхронным.

(*Прим. перев.*).

Цикл обработки сообщений и оконная процедура работают не параллельно. Когда оконная процедура обрабатывает сообщение, то это результат вызова функции *DispatchMessage* в *WinMain*. *DispatchMessage* не завершается до тех пор, пока оконная процедура не обработала сообщение.

Но заметьте, что оконная процедура должна быть повторно-входимой (reentrant). Это означает, что Windows часто вызывает *WndProc* с новым сообщением, как результат вызова функции *DefWindowProc* в *WndProc* с предыдущим сообщением. В большинстве случаев повторная входимость оконной процедуры не создает проблем, но об этом следует знать.

Например, предположим, что вы вводите переменную в процесс обработки сообщения оконной процедурой и затем вызываете функцию Windows. Можете ли вы быть уверены в том, что после возврата функцией своего значения, ваша переменная осталась той же самой? Конечно, нет — в том случае, если конкретная функция Windows, которую вы вызвали, стала источником другого сообщения, и оконная процедура изменила вашу переменную при обработке этого второго сообщения. Это одна из причин того, что при компиляции программ для Windows необходимо отключать некоторые возможности оптимизации.

Часто возникает необходимость того, чтобы оконная процедура сохраняла информацию, полученную в сообщении, и использовала ее при обработке другого сообщения. Тогда эту информацию следует описывать в оконной процедуре в виде статических переменных, либо хранить в глобальных переменных.

Все вышеизложенное станет более понятным, когда вы прочитаете в следующих главах об оконных процедурах, способных обрабатывать большое число сообщений.

Думайте о ближнем

Windows 95 — это вытесняющая многозадачная среда. Это означает, что если программа работает слишком долго, то Windows может разрешить пользователю переключиться на другую программу. Это удобная вещь и одно из преимуществ Windows 95 по сравнению с предыдущими, основанными на DOS, версиями Windows.

Однако, поскольку Windows сконструирована определенным образом, эта вытесняющая многозадачность не всегда работает так, как бы вам хотелось. Например, предположим, что ваша программа тратит на обработку отдельного сообщения минуту или больше. Да, пользователь может переключиться на другую программу. Но пользователь ничего не в состоянии сделать с *вашей* программой. Пользователь не может переместить окно вашей программы, изменить его размер, закрыть его, т. е. вообще ничего. Так происходит потому, что, хотя ваша оконная процедура и должна выполнять все эти задачи, но она занята выполнением слишком долгой работы. Конечно, может показаться, что оконная процедура не выполняет операций по изменению размера и перемещению собственного окна, но, тем не менее, она это делает. Это часть работы функции *DefWindowProc*, которую, в свою очередь, необходимо считать частью оконной процедуры.

Если для вашей программы необходимо долго обрабатывать отдельные сообщения, то в главе 14 описан удобный способ сделать это. Даже при наличии вытесняющей многозадачности, не слишком хорошо оставлять ваше окно безвольно висящим на экране. Оно мешает пользователям, и они просто вашу программу будут ругать.

Кривая обучения

Да, как вы, несомненно, поняли из этой главы, программирование для Windows определенно отличается от программирования для общепринятой среды типа MS-DOS. Никто не станет утверждать, что программировать для Windows легко.

При первом изучении программирования для Windows обычно делают то, что всегда делается при изучении новой операционной системы или нового языка программирования — пишется простая программа для вывода на экран содержимого файла. В общепринятой среде MS-DOS такая программа включает в себя обработку командной строки, простейший файловый Ввод/Вывод и форматирование вывода на экран. В отличии от этого, аналогичная программа для Windows превращается в монстра. Она требует изучения меню, окон диалога, полос прокрутки и т. д. Так как это ваша первая программа для Windows, то из-за необходимости сразу усвоить слишком большой объем материала, она обычно оказывается совершенно неправильной.

Когда работа над программой подходит к концу, она уже совершенно отличается от любой программы, которую вы когда-либо писали. Вместо командной строки для ввода имени файла, программа WINDUMP (назовем ее так) предлагает на экране список всех файлов текущего каталога. Вместо последовательной печати на экране содержимого файла, как это происходит в обычном телетайпе, WINDUMP имеет полосы прокрутки, и можно перемещаться в любую часть файла. И как особая награда, можно запустить две копии WINDUMP и сравнить два находящихся рядом файла. Короче говоря, из всех ранее написанных программ для вывода содержимого файла на экран WINDUMP первая, которой можно действительно гордиться.

Выясните для себя следующий вопрос: нужно ли, чтобы в ваших программах использовался современный и удобный пользовательский интерфейс, включающий в себя меню, окна диалога, полосы прокрутки и графику?

Если да, тогда еще один вопрос: хотите ли вы сами программировать все эти меню, окна диалога, полосы прокрутки и графику? Или было бы лучше воспользоваться уже подготовленными для этого программами Windows? Другими словами, что проще: узнать, как использовать более 1000 функций, или писать их самому? Что проще: направить свои силы на изучение управляемой сообщениями архитектуры Windows, или бороться с разнообразными вариантами организации пользовательского ввода в традиционной модели программирования?

Если вы собираетесь создавать собственную логику интерфейса, вам было бы лучше закрыть эту книгу и заняться своим делом. Тем временем другим читателям предстоит в следующей главе выяснить, как вывести на экран окно с текстом и просмотреть там этот текст.

Глава 3 Рисование текста

3

В предыдущей главе вы познакомились с простой программой для Windows 95, в оторой в центре окна или, если быть более точным, в центре рабочей области окна, выводилась одна текстовая строка. Очень важно понимать разницу между окном приложения и его рабочей областью: рабочая область — это часть всего окна приложения, в верхней части которой нет строки заголовка, у которой нет рамки окна, нет строки меню, нет полос прокрутки. Короче говоря, рабочая область — это часть окна, на которой программа может рисовать и представлять визуальную информацию для пользователя.

Вы можете делать с рабочей областью вашей программы почти все, что захотите — все, за исключением того, что у вас отсутствует возможность задать ей определенный размер или оставить этот размер неизменным во время работы вашей программы. Если вы приспособились писать программы для MS-DOS, эти условия могут вас слегка огорчить. Вы можете больше не думать о рамках экранного пространства, т. е. о 25 (или 43, или 50) строках текста и 80 символов. Ваша программа должна делить экран с другими программами. Пользователь Windows управляет тем, как расположить на экране окна программ. Хотя вполне можно создать окно фиксированного размера (как это делается в программе калькулятора и сходных с ней программах), в подавляющем большинстве случаев размер окна — это величина переменная. Ваша программа должна учитывать размер окна и использовать рациональные способы работы с ним.

Здесь возможны два варианта. Рабочая область может оказаться недостаточной даже для того, чтобы поместилось только слово "Hello". Точно также она может оказаться на большом экране видеосистемы с высоким разрешением, и оказаться настолько большой, что в ней могли бы поместиться две полные страницы текста и, кроме этого, на экране еще осталось бы множество участков свободного пространства. Умение разумно поступать в обеих этих ситуациях — это важная часть программирования для Windows.

Хотя в Windows имеется очень широкий набор функций графического интерфейса устройства (GDI) для вывода графики на экран, в этой главе будет рассказано только о выводе простых текстовых строк. Не будут также рассматриваться различные шрифты и их размеры, имеющиеся в Windows, и будет использоваться только системный шрифт (system font), который Windows использует по умолчанию. Этого может показаться недостаточно, но на самом деле вполне хватает. Задачи, с которыми мы в этой главе столкнемся, и которые решим, относятся к программированию для Windows в целом. Если вы выводите на экран сочетание текста и графики (как, например, это происходит в программе Windows Calculator), размеры символов, задаваемые по умолчанию системным шрифтом Windows, часто определяют и размеры графики.

На первый взгляд, эта глава посвящена тому, как научиться рисовать, но в действительности в ней изучаются основы программирования, независимого от используемого оборудования. В программах для Windows можно не задумываться о размерах рабочей области окна или даже о размерах текстовых символов. Вместо этого в них нужно использовать те возможности, которые предлагает операционная система для получения информации о среде, в которой работает программа.

Рисование и обновление

При работе в MS-DOS программа, использующая полноэкранный режим вывода информации, может выводить текст в любую часть экрана. То, что программа вывела на экран, там и останется и никуда таинственно не исчезнет. Программе уже не нужна информация, необходимая для повторного вывода информации на экран. Если другая программа (например, резидентная) закроет часть экрана, тогда эта резидентная программа и должна, после завершения своей работы, восстановить содержимое экрана.

В Windows можно выводить информацию только в рабочую область окна, и вы не можете быть уверены в том, что там что-нибудь будет оставаться, до тех пор пока ваша программа специально не выведет что-нибудь поверх. Например, окно диалога другого приложения может перекрыть часть вашей рабочей области. Хотя Windows будет

пытаться сохранить и восстановить область экрана под окном диалога, это не всегда получается. После того как окно диалога удаляется с экрана, Windows выдаст запрос, требующий, чтобы ваша программа перерисовала эту часть рабочей области.

Windows — это операционная система, управляемая сообщениями. Windows уведомляет приложения о различных событиях путем постановки синхронных сообщений в очередь сообщений приложения или путем отправки асинхронных сообщений соответствующей процедуре окна. Посылая синхронное сообщение WM_PAINT, Windows уведомляет оконную процедуру о том, что часть рабочей области окна необходимо обновить.

Сообщение WM_PAINT

Большинство программ для Windows вызывают функцию *UpdateWindow* при инициализации в *WinMain*, сразу перед входом в цикл обработки сообщений. Windows использует эту возможность для асинхронной отправки в оконную процедуру первого сообщения WM_PAINT. Это сообщение информирует оконную процедуру о том, что рабочая область готова к рисованию. После этого оконная процедура должна быть готова в любое время обработать дополнительные сообщения WM_PAINT и даже перерисовать, при необходимости, всю рабочую область окна. Оконная процедура получает сообщение WM_PAINT при возникновении одной из следующих ситуаций:

- Предварительно скрытая область окна открылась, когда пользователь передвинул окно или выполнил какие-то действия, в результате которых окно вновь стало видимым.
- Пользователь изменил размера окна (если в стиле класса окна установлены биты CS_HREDRAW и CS_HVREDRAW).
- В программе для прокрутки части рабочей области используются функции *ScrollWindow* или *ScrollDC*.
- Для генерации сообщения WM_PAINT в программе используются функции *InvalidateRect* или *InvalidateRgn*.

В некоторых случаях, когда часть рабочей области временно закрывается, Windows пытается сначала ее сохранить, а затем восстановить. Это не всегда возможно. В некоторых случаях Windows может послать синхронное сообщение WM_PAINT. Обычно это происходит, когда:

- Windows удаляет диалоговое окно или окно сообщения, которое перекрывало часть окна программы.
- Раскрывается пункт горизонтального меню и затем удаляется с экрана.

В некоторых случаях Windows всегда сохраняет перекрываемую область, а потом восстанавливает ее. Происходит это всегда, когда:

- Курсор мыши перемещается по рабочей области.
- Иконку перемещают по рабочей области.

Работа с сообщением WM_PAINT требует, чтобы вы изменили свое представление о том, как выводить информацию на экран. Ваша программа должна быть структурирована таким образом, чтобы в ней была собрана вся информация, необходимая для рисования в рабочей области, но рисование осуществлялось бы только "по запросу" — когда Windows отправит оконной процедуре синхронное сообщение WM_PAINT. Если вашей программе необходимо перерисовать свою рабочую область, она может заставить Windows сгенерировать сообщение WM_PAINT. Это может показаться не самым простым способом вывода информации на экран, но структура вашей программы от этого только улучшится.

Действительные и недействительные прямоугольники

Хотя оконная процедура должна быть готова в любой момент, при получении сообщения WM_PAINT, перерисовать всю рабочую область, часто бывает необходимо перерисовать только небольшую ее часть (обычно прямоугольную область внутри рабочей зоны окна). Это наиболее очевидно, когда часть рабочей области закрыта диалоговым окном. Перерисовка требуется только для прямоугольной области, вновь открывающейся при удалении окна диалога.

Эту область называют "недействительном регионом" (invalid region) или "регионом обновления" (update region). Появление недействительного региона в рабочей области вынуждает Windows поместить синхронное сообщение WM_PAINT в очередь сообщений приложения. Ваша оконная процедура получает сообщение WM_PAINT только тогда, когда часть вашей рабочей области недействительна, т. е. требует обновления.

В Windows для каждого окна поддерживается "структура информации о рисовании" (paint information structure). В этой структуре содержатся (помимо другой информации) координаты минимально возможного прямоугольника, содержащего недействительную область. Эта информация носит название "недействительного прямоугольника"

(invalid rectangle); иногда — "недействительного региона" (invalid region). Если еще один регион рабочей области становится недействительным перед обработкой сообщения WM_PAINT, Windows рассчитывает новый недействительный регион, который содержит оба эти региона и запоминает эту новую информацию в структуре информации о рисовании. Windows не помещает в очередь сообщений сразу несколько сообщений WM_PAINT.

Оконная процедура, вызывая функцию *InvalidateRect*, может задать недействительный прямоугольник в своей рабочей области. Если в очереди сообщений уже содержится сообщение WM_PAINT, Windows рассчитывает новый недействительный прямоугольник. В противном случае Windows помещает новое сообщение WM_PAINT в очередь сообщений. При принятии сообщения WM_PAINT (как мы позже увидим в этой главе), оконная процедура может получить координаты недействительного прямоугольника. Она также может получить эти координаты в любое другое время, вызвав функцию *GetUpdateRect*.

После того как оконная процедура вызывает функцию *BeginPaint* при обработке сообщения WM_PAINT, вся рабочая область становится действительной. Программа, вызвав функцию *ValidateRect*, также может сделать действительной любую прямоугольную зону в рабочей области. Если этот вызов делает действительной всю рабочую область, тогда любое сообщение WM_PAINT, имеющееся в это время в очереди сообщений, удаляется из нее и не обрабатывается.

Введение в графический интерфейс устройства (GDI)

Для рисования в рабочей области вашего окна, вы используете функции графического интерфейса устройства. (Обзор GDI ждет нас в следующей главе). В Windows имеется несколько функций GDI для вывода строк текста в рабочей области окна. В главе 2 вы уже встречались с функцией *DrawText*, но гораздо более популярной функцией является *TextOut*. Формат этой функции следующий:

```
TextOut(hdc, x, y, psString, iLength);
```

Функция *TextOut* выводит на экран строку символов. Параметр *psString* — это указатель на строку символов, а *iLength* — длина строки символов. Параметры *x* и *y* определяют начальную позицию строки символов в рамках рабочей области. (Более подробно об этом будет рассказано в дальнейшем.) Параметр *hdc* — это "описатель контекста устройства" (handle to a device context), являющийся важной частью GDI. Практически каждой функции GDI в качестве первого параметра необходим этот описатель.

Контекст устройства

Вспомните, описатель — это просто число, которое Windows использует для внутренней ссылки на объект. Вы получаете описатель от Windows и затем используете этот описатель в разных функциях. Описатель контекста устройства — это паспорт вашего окна для функций GDI. Этот описатель дает вам полную свободу при рисовании в рабочей области вашего окна, и вы можете сделать ее такой, как пожелаете.

Контекст устройства фактически является структурой данных, которая внутренне поддерживается GDI. Контекст устройства связан с конкретным устройством вывода информации, таким как принтер, плоттер или дисплей. Что касается дисплея, то в данном случае контекст устройства обычно связан с конкретным окном на экране.

Некоторые значения в контексте устройства являются графическими "атрибутами" (attributes). Эти атрибуты определяют некоторые особенности работы функций рисования GDI. Например, для функции *TextOut* эти атрибуты контекста устройства задают цвет текста, цвет фона для текста, процедуру преобразования координат *x* и *y*, передаваемых функции *TextOut* в координаты рабочей области, а также шрифт, используемый для вывода текста.

Когда программе необходимо начать рисование, она должна получить описатель контекста устройства. После окончания рисования программа должна освободить описатель. Когда программа освободит описатель, он становится недействительным и не должен далее использоваться. Во время обработки каждого отдельного сообщения программа должна получить и освободить описатель. За исключением описателя контекста устройства, созданного функцией *CreateDC* (эта функция не будет рассматриваться в данной главе), вам не следует хранить описатель контекста устройства в промежутке между обработкой различных сообщений.

В приложениях для Windows при подготовке процесса рисования на экране, обычно используются два метода получения описателя контекста устройства.

Получение описателя контекста устройства. Первый метод

Этот метод используется при обработке сообщений WM_PAINT. Применяются две функции: *BeginPaint* и *EndPaint*. Для этих двух функций требуется описатель окна (передаваемый в оконную процедуру в качестве параметра) и адрес переменной типа структуры PAINTSTRUCT. Программисты, пишущие программы для Windows, обычно называют эту структурную переменную *ps* и определяют ее внутри оконной процедуры следующим образом:

```
PAINTSTRUCT ps;
```

Во время обработки сообщения WM_PAINT оконная процедура сначала вызывает *BeginPaint* для заполнения полей структуры *ps*. Возвращаемым значением функции *BeginPaint* является описатель контекста устройства. Обычно он передается переменной с именем *hdc*. Эта переменная определяется в оконной процедуре следующим образом:

```
HDC hdc;
```

Тип данных HDC определяется как 32-разрядное беззнаковое целое. Затем программа может использовать функции GDI, например *TextOut*. Вызов функции *EndPaint* освобождает описатель контекста устройства.

Типовой процесс обработки сообщения WM_PAINT выглядит следующим образом:

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

При обработке сообщения WM_PAINT в оконной процедуре функции *BeginPaint* и *EndPaint* должны обязательно вызываться парой. Если в оконной процедуре сообщения WM_PAINT не обрабатываются, то они должны передаваться в *DefWindowProc* (процедура обработки сообщений по умолчанию), реализованной в Windows.

DefWindowProc обрабатывает сообщения WM_PAINT следующим образом:

```
case WM_PAINT:
       BeginPaint(hwnd, &ps);
       EndPaint(hwnd, &ps);
       return 0;
```

Следующие одна за другой функции *BeginPaint* и *EndPaint* просто превращают ранее недействительный регион в действительный. Но нельзя делать следующее:

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

Windows помещает сообщение WM_PAINT в очередь сообщений, поскольку часть рабочей области окна недействительна (требует перерисовки). До тех пор пока вы не вызовете функции *BeginPaint* и *EndPaint* (или *ValidateRect*), Windows не сделает эту область действительной. Вместо этого Windows снова отправит вам сообщение WM_PAINT. И снова, и снова, и снова ...

Структура информации о рисовании

Ранее уже говорилось о структуре информации о рисовании, которая поддерживается в Windows для каждого окна. Это PAINTSTRUCT. Структура определяется так:

```
typedef struct tagPAINTSTRUCT
{
       HDC            hdc;
       BOOL           fErase;
       RECT           rcPaint;
       BOOL           fRestore;
       BOOL           fIncUpdate;
       BYTE           rgbReserved[32];
} PAINTSTRUCT;
```

Windows заполняет поля этой структуры, когда ваша программа вызывает *BeginPaint*. Вам в программе можно использовать только первые три поля структуры. Остальные используются Windows.

Поле *hdc* — это описатель контекста устройства. Возвращаемым значением функции *BeginPaint* также является описатель контекста устройства, и такая избыточность характерна для Windows.

В подавляющем большинстве случаев, в поле *fErase* установлен флаг TRUE (т. е., ненулевое значение), означающий, что Windows обновит фон недействительного прямоугольника. Windows перерисует фон, используя кисть, заданную в поле *hbrBackground* структуры WNDCLASSEX, которую вы использовали при регистрации класса окна во время инициализации *WinMain*. Многие программы для Windows используют белую кисть:

```
wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
```

Однако, если вы, вызывая функцию *InvalidateRect*, делаете недействительным прямоугольник рабочей зоны вашей программы, то последний параметр этой функции определяет, хотите ли вы стирать фон. Если этот параметр равен FALSE (т. е. 0), Windows не будет стирать фон и поле *fErase* также будет равно FALSE.

Поле *rcPaint* структуры PAINTSTRUCT — это структура типа RECT. Как вы знаете из главы 2, структура RECT определяет прямоугольник. В ней имеется четыре поля: *left*, *top*, *right* и *bottom*. Поле *rcPaint* структуры PAINTSTRUCT определяет границы недействительного прямоугольника, как показано на рис. 3.1. Значения заданы в пикселях относительно левого верхнего угла рабочей области. Недействительный прямоугольник — это та область, которую вы хотите перерисовать. Хотя программа для Windows может просто перерисовать всю рабочую область окна при получении сообщения WM_PAINT, перерисовка только той области окна, которая задана этим прямоугольником, экономит время.

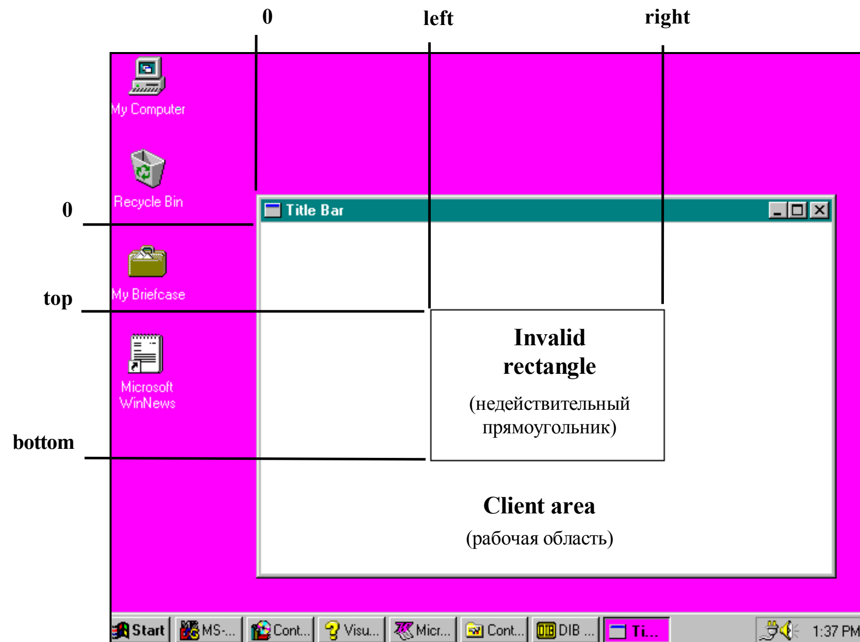


Рис. 3.1 Границы недействительного прямоугольника

Прямоугольник *rcPaint* в PAINTSTRUCT — это не только недействительный прямоугольник, это также и "отсекающий" (clipping) прямоугольник. Это означает, что Windows рисует только внутри отсекающего прямоугольника. (Или точнее, если недействительная зона не является прямоугольником, Windows рисует только внутри этой зоны.) Когда вы используете описатель контекста устройства из структуры PAINTSTRUCT, Windows не будет рисовать вне прямоугольника *rcPaint*.

Чтобы при обработке сообщения WM_PAINT рисовать вне прямоугольника *rcPaint*, вы можете сделать вызов:

```
InvalidateRect(hWnd, NULL, TRUE);
```

перед вызовом *BeginPaint*. Это сделает недействительной всю рабочую область и обновит ее фон. Если же значение последнего параметра будет равно FALSE, то фон обновляться не будет. Все что там было, останется неизменным.

В программе HELLOWIN из главы 2 мы не думали о недействительных или отсекающих прямоугольниках при обработке сообщения WM_PAINT. Если оказывалось, что область вывода текста на экран находится внутри недействительного прямоугольника, то функция *DrawText* перерисовывала ее. Если при обработке вызова *DrawText* Windows не находит такие области, на экран ничего не выводится. Но такой поиск требует времени. Программист, заботящийся об эффективности и быстродействии, захочет при обработке сообщений WM_PAINT использовать размеры недействительного прямоугольника, чтобы не обращаться лишний раз к вызовам GDI.

Получение описателя контекста устройства. Второй метод

Вы также можете получить описатель контекста устройства, если хотите рисовать в рабочей области при обработке отличных от WM_PAINT сообщений, или если вам необходим описатель контекста устройства для других целей, например, для получения информации о самом контексте устройства. Вызывайте *GetDC* для получения описателя контекста устройства и *ReleaseDC*, если он вам больше не нужен:

```
hdc=GetDC(hwnd);
[использование функций GDI]
ReleaseDC(hwnd, hdc);
```

Также как *BeginPaint* и *EndPaint*, функции *GetDC* и *ReleaseDC* следует вызывать парой. Если вы при обработке сообщения вызываете *GetDC*, то перед выходом из оконной процедуры необходимо вызвать *ReleaseDC*. Не вызывайте *GetDC* при обработке одного сообщения, а *ReleaseDC* при обработке другого.

В отличии от описателя контекста устройства, полученного из структуры PAINTSTRUCT, в описателе контекста устройства, возвращаемом функцией *GetDC*, определен отсекающий прямоугольник, равный всей рабочей области. Вы можете рисовать в любом месте рабочей области, а не только в недействительном прямоугольнике (если недействительный прямоугольник вообще определен). В отличии от *BeginPaint*, *GetDC* не делает действительными какие-либо недействительные зоны.

Как правило, вы будете использовать вызовы функций *GetDC* и *ReleaseDC* в ответ на сообщения от клавиатуры (например, в программах текстовых редакторов) или на сообщения от манипулятора мышь (например, в программах рисования). Они позволяют обновлять рабочую область непосредственно в ответ на пользовательский ввод информации с клавиатуры или с помощью мыши, при этом специально делать недействительной часть окна для выдачи сообщений WM_PAINT. Однако, в ваших программах должно содержаться достаточно информации для обновления экрана в любой момент, когда вы получаете сообщение WM_PAINT.

Функция *TextOut*. Подробности

Когда вы получаете описатель контекста устройства, Windows заполняет внутреннюю структуру контекста устройства задаваемыми по умолчанию значениями. Как вы увидите в следующих главах, эти задаваемые по умолчанию значения можно изменить с помощью функций GDI. Из тех функций GDI, которые нас интересуют прямо сейчас, рассмотрим *TextOut*:

```
TextOut(hdc, x, y, psString, iLength);
```

Давайте исследуем эту функцию более подробно.

Первый параметр — это описатель контекста устройства, являющийся возвращаемым значением либо функции *GetDC*, либо функции *BeginPaint*, полученный при обработке сообщения WM_PAINT.

Атрибуты контекста устройства управляют характеристиками выводимого на экран текста. Например, один атрибут контекста устройства задает цвет текста. Цвет, задаваемый по умолчанию — черный. Контекст устройства по умолчанию также определяет цвет фона — белый. Когда программа выводит текст на экран, Windows использует этот цвет фона для заполнения прямоугольной зоны вокруг каждого символа, эта зона называется знакоместом (character box).

Цвет фона текста не является цветом фона, который вы установили при определении класса окна. Фон в классе окна — это кисть, являющаяся шаблоном, которая может иметь, а может и не иметь чистый (без полутонов) цвет, и которую Windows использует для закрашивания рабочей области. Фон в классе окна не имеет отношения к структуре контекста устройства. При определении структуры класса окна в большинстве приложений Windows используется кисть WHITE_BRUSH, поэтому задаваемый по умолчанию в контексте устройства цвет фона оказывается таким же, как и цвет кисти, используемой Windows для закрашивания фона рабочей области.

Параметр *psString* — это указатель на символьную строку, а *iLength* — длина строки, т. е., число символов в строке. Строка не должна содержать никаких управляющих символов ASCII, таких как возврат каретки, перевод строки, табуляция или забой. Windows выводит такие управляющие символы в виде прямоугольников или закрашенных блоков. *TextOut* не определяет конца строки по нулевому символу, и поэтому для задания ее длины необходим параметр *iLength*.

Значения *x* и *y* в *TextOut* определяют точку начала строки текста внутри рабочей области. Значение *x* в горизонтальном направлении, значение *y* в вертикальном. Левый верхний угол первого символа строки имеет координаты
(*x, y*). В контексте устройства по умолчанию исходной точкой отсчета (когда *x* и *y* равны 0) является левый верхний угол рабочей области. Если *x* и *y* в *TextOut* равны 0, строка текста начнет выводиться, начиная с левого верхнего угла рабочей области.

Координаты GDI в документации упоминаются как "логические координаты" (logical coordinates). Что именно это означает, мы более подробно изучим в следующей главе. Сейчас достаточно знать, что в Windows имеются различные "режимы отображения" (mapping mode), которые определяют, как логические координаты, заданные в функциях GDI, преобразуются в реальные физические координаты дисплея. Режим отображения определяется в контексте устройства. Задаваемый по умолчанию режим отображения называется MM_TEXT (идентификатор, заданный в заголовочных файлах Windows). В режиме отображения MM_TEXT логические единицы соответствуют физическим единицам, каковыми являются пиксели, задаваемые относительно левого верхнего угла рабочей области. Значения по координате *x* увеличиваются при движении вправо по рабочей области, а по *y* — при движении вниз. (См. рис. 3.2.) Система координат режима MM_TEXT эквивалентна системе координат, которую Windows использует для определения недействительного прямоугольника в структуре PAINTSTRUCT. Это очень удобно. (Однако это не относится к другим режимам отображения).

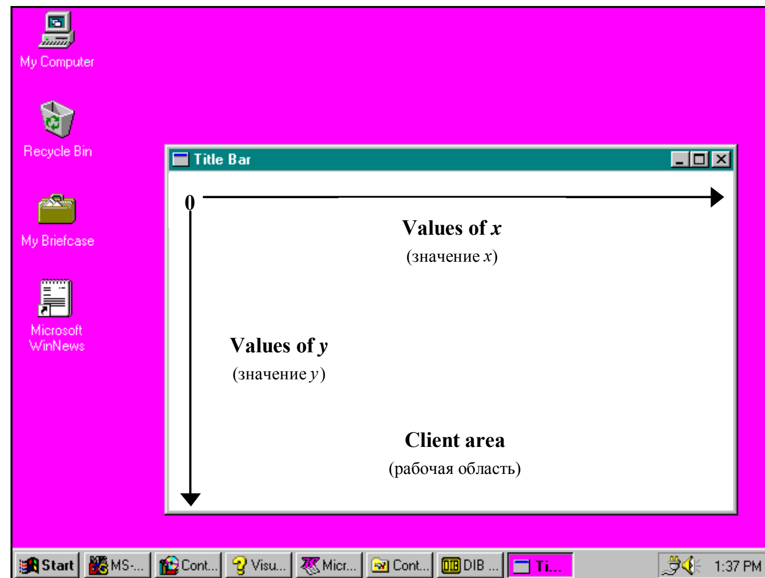


Рис. 3.2 Координаты x и y в режиме отображения MM_TEXT

Контекст устройства также определяет "регион отсечения" (clipping region). Как вы уже узнали, задаваемый по умолчанию регион отсечения — это или вся рабочая область, если описатель контекста устройства получен из функции *GetDC*, или недействительный регион, если описатель контекста устройства получен из функции *BeginPaint*. Windows не выведет на экран ту часть строки символов, которая лежит вне региона отсечения. Если часть символа находится внутри отсекающей зоны, Windows выводит на экран только эту часть. Вывести что-нибудь за пределы рабочей области вашего окна очень непросто, поэтому не беспокойтесь о том, что это случится по невнимательности.

Системный шрифт

Контекст устройства также определяет шрифт, который Windows использует при выводе текста в рабочую область. По умолчанию задается так называемый "системный шрифт" (system font) или (используя идентификатор заголовочных файлов Windows) SYSTEM_FONT. Системный шрифт — это шрифт, который Windows использует для текста заголовков, меню и окон диалога.

В ранних версиях Windows системный шрифт был фиксированным, т. е. у всех символов была одинаковая ширина, также как на пишущей машинке. Однако, начиная с Windows 3.0 (и до Windows 95 включительно), системный шрифт стал пропорциональным, т. е. разные символы имеют разную ширину. Например, символ W шире символа i. Так было сделано потому, что пропорциональный шрифт гораздо лучше читается, чем фиксированный. Но, как вы могли бы догадаться, изменение задаваемого по умолчанию шрифта с фиксированного на пропорциональный привело в негодность массу программ для первых версий Windows и потребовало от программистов изучить некоторые новые приемы работы с текстом.

Системный шрифт является "растровым шрифтом" (raster font). Это означает, что все символы определяются как пиксельные шаблоны. Распространяемые для продажи версии Windows включают в себя несколько системных шрифтов различных размеров для использования с различными видеоадаптерами. Когда производители новых видеоплат создают новый дисплейный драйвер, они также ответственны и за разработку системного шрифта, соответствующего разрешению дисплея. В противном случае, производителю пришлось бы указывать, какой именно системный шрифт из входящих в комплект Windows, может использоваться. Системный шрифт должен быть разработан таким образом, чтобы на экране помещалось, по крайней мере, 25 строк и 80 символов текста. Только это гарантирует, что имеется некоторое соответствие между размером экрана и размером шрифта в Windows.

Размер символа

Для вывода на экран нескольких строк текста с использованием функции *TextOut*, вам необходимо задать размеры символов шрифта. Следующие друг за другом строки текста вы размещаете на экране с учетом высоты символа, а колонки текста в рабочей области — исходя из усредненной ширины символов шрифта.

Размеры символов можно получить с помощью вызова функции *GetTextMetrics*. Для функции *GetTextMetrics* требуется описатель контекста устройства, поскольку ее возвращаемым значением является информация о шрифте, выбранном в данное время в контексте устройства. Windows копирует различные значения метрических параметров текста в структуру типа TEXTMETRIC. Значения определяются в единицах, зависящих от выбранного

в контексте устройства режима отображения. Выбранным по умолчанию в контексте устройства режимом отображения является режим MM_TEXT, а единицами измерения являются пиксели.

Для использования функции *GetTextMetrics*, во-первых, необходимо определить структурную переменную (которую обычно называют *tm*):

```
TEXTMETRIC tm;
```

Далее нужно получить описатель контекста устройства и вызвать *GetTextMetrics*:

```
hdc = GetDC(hwnd);
GetTextMetrics(hdc, &tm);
ReleaseDC(hwnd, hdc);
```

Теперь вы можете проанализировать значения в структуре текстовых размеров и, возможно, сохранить несколько из них для использования в будущем.

Метрические параметры текста. Подробности

Структура TEXTMETRIC обеспечивает полную информацию о выбранном в данный момент в контексте устройства шрифте. Вертикальный размер шрифта определяется только пятью величинами, как показано на рис. 3.3.

Эти величины вполне объяснимы. Значение *tmInternalLeading* — это величина пустого пространства, отведенного для указания специальных знаков над символом. Если это значение равно 0, то помеченные прописные буквы делаются немного меньше, чтобы специальный символ поместился внутри верхней части символа. Значение *tmExternalLeading* — это величина пустого пространства, которое разработчик шрифта установил для использования между строками символов. Вы можете согласиться или отклонить предложение разработчика при распределении пространства между строками текста.

В структуре TEXTMETRIC имеется два поля, описывающих ширину символа: *tmAveCharWidth* (усредненная ширина символов строки) и *tmMaxCharWidth* (ширина самого широкого символа шрифта). Для фиксированного шрифта эти две величины одинаковы.

В примерах программ этой главы используется другое значение ширины символа — средняя ширина прописных букв. Она может быть точно рассчитана как 150% от *tmAveCharWidth*.

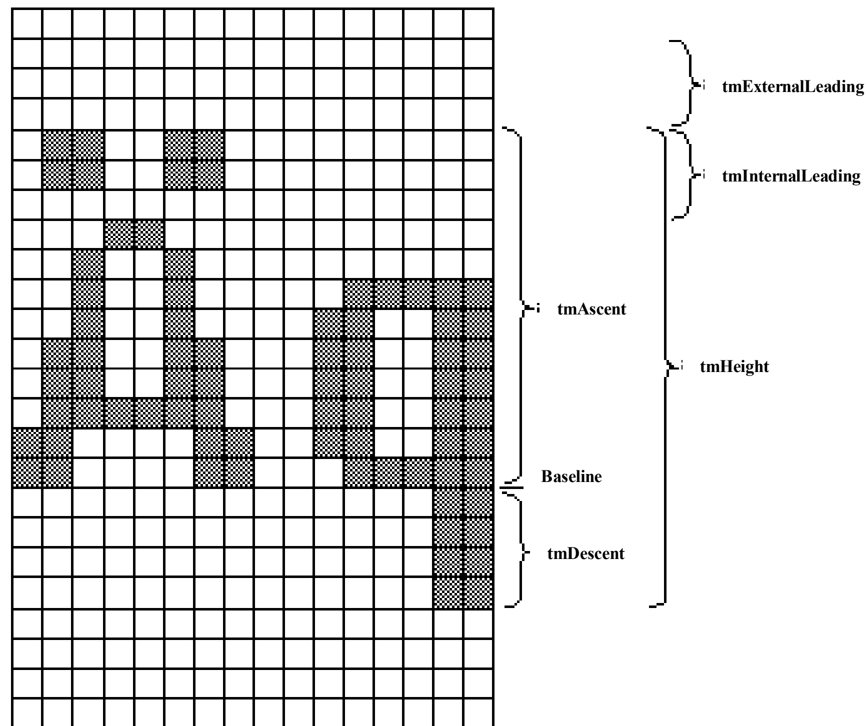


Рис. 3.3 Пять значений, определяющих вертикальный размер шрифта

Важно понимать, что размеры системного шрифта зависят от разрешающей способности дисплея, на котором работает Windows. Система Windows обеспечивает независимый от оборудования графический интерфейс, но, тем не менее, вам необходимо помочь ей. Не пишите программы для Windows, опирающиеся на конкретные размеры

символов. Не задавайте никаких фиксированных значений. Используйте функцию *GetTextMetrics* для получения этой информации.

Форматирование текста

Поскольку размеры системного шрифта не меняются в рамках одного сеанса работы с системой Windows, вам необходимо вызвать функцию *GetTextMetrics* только один раз при выполнении программы. Хорошо сделать этот вызов при обработке сообщения WM_CREATE в оконной процедуре. Сообщение WM_CREATE — это первое сообщение, которое принимает оконная процедура. Windows вызывает вашу оконную процедуру с сообщением WM_CREATE, когда вы вызываете в *WinMain* функцию *CreateWindow*.

Предположим, что вы пишите программу для Windows, которая выводит на экран несколько строк текста, появляющихся друг за другом в направлении к нижней границе рабочей области. Вы хотите получить значения высоты и ширины символов. Внутри оконной процедуры вы можете определить две переменные для хранения средней ширины символов (*cxChar*) и полной высоты символов (*cyChar*):

```
static int cxChar, cyChar;
```

Префикс *c*, добавленный к именам переменных, означает "счетчик" (count) и в сочетании с *x* или *y* относится к длине или ширине. Эти переменные определяются как статические, поскольку они должны оставаться без изменений при обработке оконной процедурой других сообщений (таких как WM_PAINT). Если эти переменные определяются как глобальные вне какой бы то ни было функции, то нет необходимости определять их как статические.

Здесь представлен код обработки сообщения WM_CREATE:

```
case WM_CREATE:
       hdc = GetDC(hwnd);

       GetTextMetrics(hdc, &tm);
       cxChar = tm.tmAveCharWidth;
       cyChar = tm.tmHeight + tm.tmExternalLeading;

       ReleaseDC(hwnd, hdc);
       return 0;
```

Если вы не хотите учитывать межстрочное пространство (external leading), то можно использовать:

```
cyChar = tm.tmHeight;
```

От вас зависит, как использовать полученный размер символов для расчета координат при выводе информации. Самое простое — оставить пустое поле *cyChar* в верхней части рабочей области, и *cxChar* в ее левой части. Для вывода на экран нескольких строк текста с выравниванием по левому краю при вызове функции *TextOut* используйте следующее значение координаты *x*:

```
cxChar
```

Значение координаты *y* в *TextOut*:

```
cyChar *(1 + i)
```

где *i* — это номер строки, начиная с 0.

Вы часто будете сталкиваться с необходимостью выводить отформатированные числа как простые символьные строки. Если вы программировали для MS-DOS с использованием стандартных библиотечных функций языка С, вы, вероятно, для форматирования применяли функцию *printf*. В Windows функция *printf* не работает, поскольку *printf* предназначена для стандартного устройства вывода, а установка в Windows не имеет смысла.

В Windows есть аналогичная функция *sprintf*. Она работает точно также, как и функция *printf*, отличие заключается в том, что форматируемая строка помещается в символьный массив. Для вывода строки на экран можно затем использовать функцию *TextOut*. Очень удобно то, что возвращаемым значением *sprintf* является длина строки — вы можете передать это значение в *TextOut* в качестве параметра *iLength*. Следующий фрагмент программы показывает типовое использование функций *sprintf* и *TextOut*:

```
int iLength;
char szBuffer[40];
[другие инструкции программы]
iLength = sprintf(szBuffer, "Сумма %d и %d равна %d", nA, nB, nA + nB);
TextOut(hdc, x, y, szBuffer, iLength);
```

Для достаточно простых задач вы могли бы избежать вычисления *iLength* и объединить два оператора в один:

```
TextOut(hdc, x, y, szBuffer, sprintf(szBuffer, "Сумма %d и %d равна %d", nA, nB, nA + nB));
```

Это не очень красиво, но вполне работоспособно.

Если вам не нужно выводить числа с плавающей точкой, то вместо *sprintf* вам лучше использовать функцию *wsprintf*. Синтаксис функции *wsprintf* такой же, как и *sprintf*, но она включена в Windows, поэтому ее использование не увеличит размер вашего EXE-файла.

Соединим все вместе

Теперь, как кажется, у нас есть все необходимое, чтобы написать простую программу для вывода на экран нескольких строк текста. Мы знаем, как получить описатель контекста устройства, как использовать функцию *TextOut*, и как разместить текст, основываясь на размере одного символа. Осталось только придумать, что бы такое интересное вывести на экран.

Информация, получаемая при вызове функции Windows *GetSystemMetrics*, достаточно полезна. Эта функция возвращает информацию о размере различных графических элементов Windows, таких как значки, курсоры, панели заголовков и полосы прокрутки. В функции *GetSystemMetrics* имеется один параметр, называемый "индекс" (index). Этот индекс — один из 73 целых идентификаторов, определяемых в заголовочных файлах Windows. Возвращаемым значением *GetSystemMetrics* является целое, обычно это размер элемента, указанного в параметре.

Давайте напишем программу для вывода на экран некоторой части информации, получаемой при вызове функции *GetSystemMetrics* в простом формате: на каждой строке по одному элементу. Работать с этой информацией будет легче, если создать заголовочный файл, в котором нужно определить массив структур, содержащих идентификаторы индексов *GetSystemMetrics* и текст, который мы хотим выводить на экран для каждого возвращаемого функцией значения. Заголовочный файл назовем SYSMETS.H, он показан на рис. 3.4.

SYSMETS.H

```
/*-----------------------------------------------
   SYSMETS.H -- System metrics display structure
  -----------------------------------------------*/

#define NUMLINES((int)(sizeof sysmetrics / sizeof sysmetrics [0]))

struct
{
        int  iIndex;
        char *szLabel;
        char *szDesc;
} sysmetrics [] =
{
        SM_CXSCREEN,          "SM_CXSCREEN",          "Screen width in pixels",
        SM_CYSCREEN,          "SM_CYSCREEN",          "Screen height in pixels",
        SM_CXVSCROLL,         "SM_CXVSCROLL",         "Vertical scroll arrow width",
        SM_CYHSCROLL,         "SM_CYHSCROLL",         "Horizontal scroll arrow height",
        SM_CYCAPTION,         "SM_CYCAPTION",         "Caption bar height",
        SM_CXBORDER,          "SM_CXBORDER",          "Window border width",
        SM_CYBORDER,          "SM_CYBORDER",          "Window border height",
        SM_CXDLGFRAME,        "SM_CXDLGFRAME",        "Dialog window frame width",
        SM_CYDLGFRAME,        "SM_CYDLGFRAME",        "Dialog window frame height",
        SM_CYVTHUMB,          "SM_CYVTHUMB",          "Vertical scroll thumb height",
        SM_CXHTHUMB,          "SM_CXHTHUMB",          "Horizontal scroll thumb width",
        SM_CXICON,            "SM_CXICON",            "Icon width",
        SM_CYICON,            "SM_CYICON",            "Icon height",
        SM_CXCURSOR,          "SM_CXCURSOR",          "Cursor width",
        SM_CYCURSOR,          "SM_CYCURSOR",          "Cursor height",
        SM_CYMENU,            "SM_CYMENU",            "Menu bar height",
        SM_CXFULLSCREEN,      "SM_CXFULLSCREEN",      "Full screen client area width",
        SM_CYFULLSCREEN,      "SM_CYFULLSCREEN",      "Full screen client area height",
        SM_CYKANJIWINDOW,     "SM_CYKANJIWINDOW",     "Kanji window height",
        SM_MOUSEPRESENT,      "SM_MOUSEPRESENT",      "Mouse present flag",
        SM_CYVSCROLL,         "SM_CYVSCROLL",         "Vertical scroll arrow height",
        SM_CXHSCROLL,         "SM_CXHSCROLL",         "Horizontal scroll arrow width",
        SM_DEBUG,             "SM_DEBUG",             "Debug version flag",
        SM_SWAPBUTTON,        "SM_SWAPBUTTON",        "Mouse buttons swapped flag",
        SM_RESERVED1,         "SM_RESERVED1",         "Reserved",
        SM_RESERVED2,         "SM_RESERVED2",         "Reserved",
```

```
        SM_RESERVED3,          "SM_RESERVED3",          "Reserved",
        SM_RESERVED4,          "SM_RESERVED4",          "Reserved",
        SM_CXMIN,              "SM_CXMIN",              "Minimum window width",
        SM_CYMIN,              "SM_CYMIN",              "Minimum window height",
        SM_CXSIZE,             "SM_CXSIZE",             "Minimize/Maximize icon width",
        SM_CYSIZE,             "SM_CYSIZE",             "Minimize/Maximize icon height",
        SM_CXFRAME,            "SM_CXFRAME",            "Window frame width",
        SM_CYFRAME,            "SM_CYFRAME",            "Window frame height",
        SM_CXMINTRACK,         "SM_CXMINTRACK",         "Minimum window tracking width",
        SM_CYMINTRACK,         "SM_CYMINTRACK",         "Minimum window tracking height",
        SM_CXDOUBLECLK,        "SM_CXDOUBLECLK",        "Double click x tolerance",
        SM_CYDOUBLECLK,        "SM_CYDOUBLECLK",        "Double click y tolerance",
        SM_CXICONSPACING,      "SM_CXICONSPACING",      "Horizontal icon spacing",
        SM_CYICONSPACING,      "SM_CYICONSPACING",      "Vertical icon spacing",
        SM_MENUDROPALIGNMENT, "SM_MENUDROPALIGNENT",  "Left or right menu drop",
        SM_PENWINDOWS,         "SM_PENWINDOWS",         "Pen extensions installed",
        SM_DBCSENABLED,        "SM_DBCSENABLED",        "Double-Byte Char Set enabled",
        SM_CMOUSEBUTTONS,      "SM_CMOUSEBUTTONS",      "Number of mouse buttons",
        SM_SHOWSOUNDS,         "SM_SHOWSOUNDS",         "Present sounds visually"
};
```

Рис. 3.4 SYSMETS.H

Программа для вывода этой информации называется SYSMETS1. Файлы, необходимые для создания SYSMETS.EXE (make-файл и файл с исходным текстом на С) приведены на рис. 3.5. Большинство операторов должны быть теперь вам понятны. За исключением имени программы, make-файл идентичен make-файлу программы HELLOWIN. В SYSMETS1.С функция *WinMain* фактически идентична HELLOWIN.

SYSMETS1.MAK

```
#------------------------
# SYSMETS1.MAK make file
#------------------------

sysmets1.exe : sysmets1.obj
     $(LINKER) $(GUIFLAGS) -OUT:sysmets1.exe sysmets1.obj $(GUILIBS)

sysmets1.obj : sysmets1.c sysmets.h
     $(CC) $(CFLAGS) sysmets1.c
```

SYSMETS1.C

```
/*----------------------------------------------------
   SYSMETS1.C -- System Metrics Display Program No. 1
                 (c) Charles Petzold, 1996
  ----------------------------------------------------*/

#include <windows.h>
#include <string.h>
#include "sysmets.h"

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow)
{
        static char szAppName[] = "SysMets1";
        HWND        hwnd;
        MSG         msg;
        WNDCLASSEX  wndclass;

        wndclass.cbSize         = sizeof(wndclass);
        wndclass.style          = CS_HREDRAW | CS_VREDRAW;
        wndclass.lpfnWndProc    = WndProc;
        wndclass.cbClsExtra     = 0;
        wndclass.cbWndExtra     = 0;
        wndclass.hInstance      = hInstance;
```

```
        wndclass.hIcon          = LoadIcon(NULL, IDI_APPLICATION);
        wndclass.hCursor        = LoadCursor(NULL, IDC_ARROW);
        wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
        wndclass.lpszMenuName  = NULL;
        wndclass.lpszClassName = szAppName;
        wndclass.hIconSm        = LoadIcon(NULL, IDI_APPLICATION);

        RegisterClassEx(&wndclass);

        hwnd = CreateWindow(
                        szAppName,
                        "Get System Metrics No. 1",
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        NULL, NULL, hInstance, NULL
        );

        ShowWindow(hwnd, iCmdShow);
        UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
        static int  cxChar, cxCaps, cyChar;
        char        szBuffer[10];
        HDC         hdc;
        int         i;
        PAINTSTRUCT ps;
        TEXTMETRIC  tm;

        switch(iMsg)
        {
        case WM_CREATE :
                hdc = GetDC(hwnd);
                GetTextMetrics(hdc, &tm);
                cxChar = tm.tmAveCharWidth;
                cxCaps =(tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2;
                cyChar = tm.tmHeight + tm.tmExternalLeading;

                ReleaseDC(hwnd, hdc);
                return 0;

        case WM_PAINT :
                hdc = BeginPaint(hwnd, &ps);

                for(i = 0; i < NUMLINES; i++)
                {
                        TextOut(
                                hdc, cxChar, cyChar *(1 + i),
                                sysmetrics[i].szLabel,
                                strlen(sysmetrics[i].szLabel)
                        );

                        TextOut(
                                hdc, cxChar + 22 * cxCaps, cyChar *(1 + i),
                                sysmetrics[i].szDesc,
```

```
                              strlen(sysmetrics[i].szDesc)
                    );

                    SetTextAlign(hdc, TA_RIGHT | TA_TOP);

                    TextOut(
                              hdc, cxChar + 22 * cxCaps + 40 * cxChar,
                              cyChar *(1 + i), szBuffer,
                              wsprintf(
                                        szBuffer, "%5d",
                                        GetSystemMetrics(sysmetrics[i].iIndex)
                              )
                    );

                    SetTextAlign(hdc, TA_LEFT | TA_TOP);
          }

          EndPaint(hwnd, &ps);
          return 0;

     case WM_DESTROY :
          PostQuitMessage(0);
          return 0;
     }

     return DefWindowProc(hwnd, iMsg, wParam, lParam);
}
```

Рис. 3.5 Программа SYSMETS1

На рис. 3.6 показано окно программы SYSMETS1, работающей на VGA. Из информации в окне программы следует, что ширина экрана составляет 640 пикселей, а высота 480 пикселей. Эти два значения, также как и многие другие значения, выведенные в окне, могут отличаться для различных типов дисплеев.

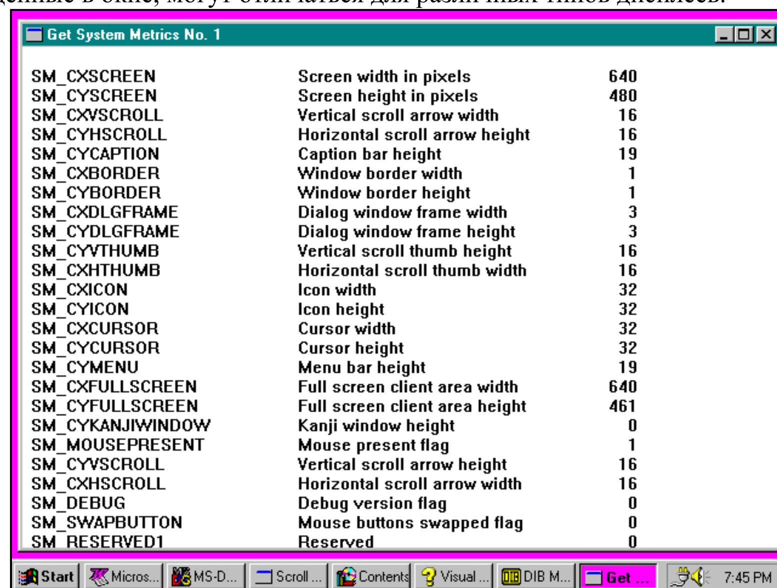


Рис. 3.6 Вывод информации программой SYSMETS1

Оконная процедура программы SYSMETS1.C

Оконная процедура *WndProc* программы SYSMETS1.C обрабатывает три сообщения: WM_CREATE, WM_PAINT и WM_DESTROY. Сообщение WM_DESTROY обрабатывается точно также, как и в программе HELLOWIN, рассмотренной в главе 2.

Сообщение WM_CREATE является первым сообщением, получаемым оконной процедурой. Оно генерируется операционной системой Windows, когда функция *CreateWindow* создает окно. При обработке сообщения WM_CREATE SYSMETS1 получает контекст устройства для окна путем вызова функции *GetDC*, а также размеры

текста для системного шрифта по умолчанию путем вызова функции *GetTextMetrics*. SYSMETS1 сохраняет усредненное значение ширины символа в *cxChar* и полную высоту символов (включая поле *tmExternalLeading*) в *cyChar*.

SYSMETS1 также сохраняет среднюю ширину символов верхнего регистра в статической переменной *cxCaps*. Для фиксированного шрифта *cxCaps* была бы равна *cxChar*. Для пропорционального шрифта *cxCaps* равна 150% от *cxChar*. Младший бит поля *tmPitchAndFamily* структуры TEXTMETRIC равен 1 для пропорционального шрифта и 0 для фиксированного. SYSMETS1 использует значение этого бита для расчета *cxCaps* из *cxChar* следующим образом:

```
cxCaps =(tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2;
```

SYSMETS1 полностью выполняет процедуру рисования окна во время обработки сообщения WM_PAINT. Как правило, оконная процедура путем вызова *BeginPaint*, получает, в первую очередь, описатель контекста устройства. В цикле *for* обрабатываются все элементы структуры *sysmetrics*, определенной в SYSMETS.H. Три колонки текста выводятся на экран тремя функциями *TextOut*. В каждом случае третий параметр *TextOut* — это выражение:

```
cyChar *(1 + i)
```

Этот параметр показывает в пикселях положение верхней границы строки символов относительно верхней границы рабочей области. Таким образом программа делает верхний отступ равным *cyChar*. Первая строка текста (если *i* равно 0) начинается на *cyChar* пикселей ниже верхней границы рабочей области.

Первая инструкция *TextOut* выводит на экран в первую колонку идентификаторы, написанные прописными буквами. Вторым параметром *TextOut* является *cxChar*. Он оставляет левый отступ между первым символом строки и левой границей рабочей области окна равным одному символу. Текст берется из поля *szLabel* структуры *sysmetrics*. Функция периода выполнения *strlen* языка C используется для получения длины строки, которая необходима в качестве последнего параметра для *TextOut*.

Вторая инструкция *TextOut* выводит на экран описание значений системных размеров. Эти описания хранятся в поле *szDesc* структуры *sysmetrics*. В этом случае второй параметр *TextOut* — это выражение:

```
cxChar + 22 * cxCaps
```

Максимальная длина идентификаторов, выводимых на экран в первой колонке прописными буквами, составляет 20 символов, поэтому вторая колонка должна начинаться в позиции, по крайней мере, на 20* *cxCaps* правее начала первой колонки текста.

Третий оператор *TextOut* выводит на экран численные значения, полученные от функции *GetTextMetrics*. Пропорциональный шрифт делает форматирование колонки, выравненных по правому краю чисел, слегка обманчивым. Каждая цифра от 0 до 9 имеют одну и ту же ширину, но эта ширина больше, чем ширина пробела. Числа могут быть шириной в одну или более цифр, поэтому начальное горизонтальное положение чисел может меняться от строки к строке.

Возможно, было бы проще, если бы мы выводили колонку выравненных по правому краю чисел, задав положение последней цифры, вместо первой. Это позволяет делать функция *SetTextAlign*. После вызова этой функции в программе SYSMETS1:

```
SetTextAlign(hdc, TA_RIGHT | TA_TOP);
```

координаты, переданные последующим функциям *TextOut*, будут задавать правый верхний угол строки текста вместо ее левого верхнего угла.

Второй параметр функции *TextOut* для вывода колонки чисел — это выражение:

```
cxChar + 22 * cxCaps + 40 * cxChar
```

Значение 40 * *cxChar* позволяет согласовать ширину второй колонки и ширину третьей колонки. Следующий за функцией *TextOut* другой вызов функции *SetTextAlign* возвращает все в исходное состояние на время до начала следующего цикла.

Не хватает места!

В программе SYSMETS1 проявляется одна очень неприятная проблема: если у вас нет огромного экрана и видеоадаптера с высоким разрешением, вы не сможете увидеть последних строк перечня измерений системы. Если вы сузите окно, то не сможете увидеть даже сами значения.

Программа SYSMETS1 ничего не знает о ширине рабочей области. Программа выводит текст, начиная от верхнего края окна и заставляет Windows отрезать все, что оказывается вне рамок рабочей области. Это нежелательно. Первым шагом в решении этой проблемы должно быть выяснение того, насколько много информации, выводимой программой, может быть помещено в рабочую область.

Размер рабочей области

Если вы уже работаете с существующими приложениями для Windows, то обнаружите, что размер окон может меняться в очень широких пределах. Большинство (если предположить отсутствие в окнах меню или полос прокрутки) окон можно развернуть, и рабочая область займет весь экран, за исключением панели заголовка программы. Полный размер этой развернутой рабочей области становится доступным, благодаря функции *GetSystemMetrics*, использующей параметры SM_CXFULLSCREEN и SM_CYFULLSCREEN. Для дисплея типа VGA возвращаемые значения равны 640 и 461 пикселей. Минимальный размер окна может быть совершенно незначительным, иногда почти невидимым, когда рабочая область фактически исчезает.

Одним из наиболее общих способов определения размера рабочей области окна является обработка сообщения WM_SIZE в оконной процедуре. Windows посылает в оконную процедуру сообщение WM_SIZE при любом изменении размеров окна. Переменная *lParam*, переданная в оконную процедуру, содержит ширину рабочей области в младшем слове и высоту в старшем слове. Код программы для обработки этого сообщения часто выглядит следующим образом:

```
static int cxClient, cyClient;
[другие инструкции программы]
case WM_SIZE:
cxClient = LOWORD(lParam);
cyClient = HIWORD(lParam);
return 0;
```

Макросы LOWORD и HIWORD определяются в заголовочных файлах Windows. Также как *cxChar* и *cyChar*, переменные *cxClient* и *cyClient* определяются внутри оконной процедуры в качестве статических, так как они используются позднее при обработке других сообщений.

В конечном итоге за сообщением WM_SIZE будет следовать сообщение WM_PAINT. Почему? Потому что при определении класса окна мы следующим образом задали стиль класса:

```
CS_HREDRAW | CS_VREDRAW
```

Такой стиль класса указывает Windows на необходимость перерисовки как при горизонтальных изменениях размеров, так и при вертикальных.

Вы можете рассчитать полное число строк текста, которые помещаются внутри рабочей области окна по формуле:

```
cyClient / cyChar
```

Результат может быть равен 0, если высота рабочей области слишком мала для вывода на экран целого символа. Аналогично, примерное число строчных символов, которые вы сможете горизонтально изобразить в рабочей области равно:

```
cxClient / cxChar
```

Если вы определяете *cxChar* и *cyChar* при обработке сообщения WM_CREATE, не беспокойтесь о возможном делении на 0 в этом выражении. Ваша оконная процедура получает сообщение WM_CREATE, когда *WinMain* вызывает *CreateWindow*. Первое сообщение WM_SIZE приходит несколько позднее, когда *WinMain* вызывает *ShowWindow*, благодаря этому *cxChar* и *cyChar* всегда являются положительными ненулевыми величинами.

Знание размера рабочей области окна является первым шагом в обеспечении возможности для пользователя двигать текст внутри рабочей области, если рабочая область недостаточно велика, чтобы вместить в себя что-либо целиком. Если вы хорошо знакомы с другими приложениями для Windows, имеющими аналогичные требования, вы, вероятно, знаете, что нам нужно: добавить такие великолепные средства, как полосы прокрутки.

Полосы прокрутки

Полосы прокрутки являются одними из самых лучших возможностей, который дает графический интерфейс и манипулятор мышь. Они просты в использовании и обеспечивают удобный просмотр информации. Вы можете пользоваться полосами прокрутки для вывода на экран текста, графики, электронных таблиц, записей баз данных, картинок — всего, что требует больше пространства, чем доступно в рабочей области окна.

Полосы прокрутки предназначены для просмотра информации как в вертикальном (движение вверх и вниз), так и в горизонтальном (движение вправо и влево) направлениях. Вы можете щелкнуть мышью на стрелке в любом конце полосы прокрутки или между стрелками. Бегунок ("scroll box" или "thumb") перемещается по длине полосы прокрутки, индицируя положение информации на экране относительно документа в целом. Вы также можете с помощью мыши переместить бегунок в конкретное положение. На рис. 3.7 показано рекомендуемое использование вертикальной полосы прокрутки для просмотра текста.

Программисты иногда сталкиваются с проблемой терминологии, относящейся к полосам прокрутки, поскольку их точка зрения отличается от пользовательской. Пользователь, который передвигает бегунок вниз, хочет увидеть нижнюю часть документа; однако, программа фактически перемещает документ вверх относительно окна. Документация Windows и идентификаторы ее заголовочных файлов основываются на точке зрения пользователя: Прокрутка вверх означает движение к началу документа; прокрутка вниз означает движение к концу.

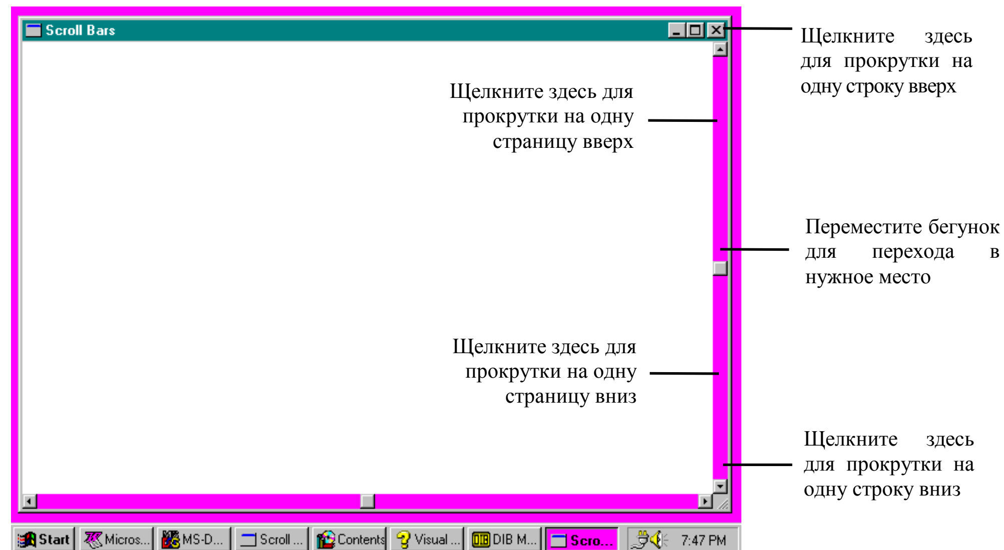


Рис. 3.7 Вертикальная полоса прокрутки

Вставить в ваше окно приложения вертикальную или горизонтальную полосу прокрутки очень просто. Все, что вам нужно сделать, это включить идентификатор WS_VSCROLL (вертикальная прокрутка) и WS_HSCROLLW (горизонтальная прокрутка) или оба сразу в описание стиля окна в инструкции *CreateWindow*. Эти полосы прокрутки всегда размещаются у правого края или в нижней части окна и занимают всю высоту или ширину рабочей области. Рабочая область не включает в себя пространство, занятое полосами прокрутки. Ширина вертикальной полосы прокрутки и высота горизонтальной постоянны для конкретного дисплейного драйвера. Если вам необходимы эти значения, вы можете получить их (как вы могли бы заметить), вызвав функцию *GetSystemMetrics*.

Windows обеспечивает всю логику работы мыши с полосами прокрутки. Однако, у полос прокрутки нет интерфейса клавиатуры. Если вы хотите дублировать клавишами управления курсором некоторые функции полос прокрутки, вы должны точно реализовать эту логику (как это делается, можно прочесть в главе 5, целиком посвященной клавиатуре).

Диапазон и положение полос прокрутки

Каждая полоса прокрутки имеет соответствующий "диапазон" (range) (два целых, отражающих минимальное и максимальное значение) и "положение" (position) (местоположение бегунка внутри диапазона). Когда бегунок находится в крайней верхней (или крайней левой) части полосы прокрутки, положение бегунка соответствует минимальному значению диапазона. Крайнее правое (или крайнее нижнее) положение бегунка на полосе прокрутки соответствует максимальному значению диапазона.

По умолчанию устанавливается следующий диапазон полосы прокрутки: 0 (сверху или слева) и 100 (снизу или справа), но диапазон легко изменить на какое-нибудь более подходящее для вашей программы значение:

```
SetScrollRange(hwnd, iBar, iMin, iMax, bRedraw);
```

Параметр *iBar* равен либо SB_VERT, либо SB_HORZ, *iMin* и *iMax* являются минимальной и максимальной границами диапазона, а *bRedraw* устанавливается в TRUE, если Вы хотите, чтобы Windows перерисовала полосы прокрутки на основе вновь заданного диапазона.

Положение бегунка всегда дискретно. Например, полоса прокрутки с диапазоном от 0 до 4 имеет пять положений бегунка, как показано на рис. 3.8. Для установки нового положения бегунка внутри диапазона полосы прокрутки можно использовать функцию *SetScrollPos*:

```
SetScrollPos(hwnd, iBar, iPos, bRedraw);
```

Параметр *iPos* — это новое положение бегунка, оно должно быть задано внутри диапазона от *iMin* до *iMax*. Для получения текущего диапазона и положения полосы прокрутки в Windows используются похожие функции (*GetScrollRange* и *GetScrollPos*).

Если в вашей программе используются полосы прокрутки, вы совместно с Windows берете на себя ответственность за поддержку полос прокрутки и обновление положения бегунка. Далее перечислены сферы ответственности Windows по поддержке полос прокрутки:

- Управляет логикой работы мыши с полосой прокрутки.
- Обеспечивает временную "инверсию цвета" при нажатии на кнопку мыши на полосе прокрутки.
- Перемещает бегунок в соответствие с тем, как внутри полосы прокрутки его перемещает пользователь.
- Отправляет сообщения полосы прокрутки в оконную процедуру для окна, содержащего полосу прокрутки.

Ниже представлены сферы ответственности вашей программы:

- Инициализация диапазона полосы прокрутки.
- Обработка сообщений полосы прокрутки.
- Обновление положения бегунка полосы прокрутки.

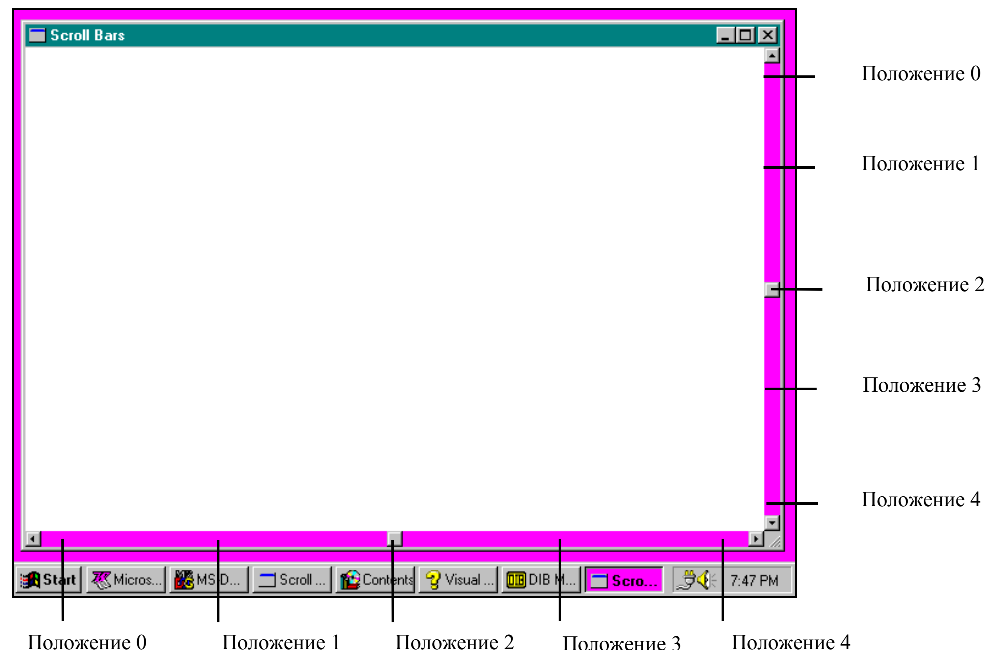


Рис. 3.8 Полосы прокрутки с пятью положениями бегунков

Сообщения полос прокрутки

Windows посылает оконной процедуре асинхронные сообщения WM_VSCROLL и WM_HSCROLL, когда на полосе прокрутки щелкают мышью или перетаскивается бегунок. Каждое действие мыши на полосе прокрутки вызывает появление по крайней мере двух сообщений, одного при нажатии кнопки мыши и другого, когда ее отпускают.

Младшее слово параметра *wParam*, которое объединяет сообщения WM_VSCROLL и WM_HSCROLL — это число, показывающее, что мышь осуществляет какие-то действия на полосе прокрутки. Его значения соответствуют определенным идентификаторам, которые начинаются с SB_, что означает "полоса прокрутки" (scroll bar). Хотя в некоторых из этих идентификаторов используются слова "UP" и "DOWN", они применяются и к горизонтальным и к вертикальным полосам прокрутки, как показано на рис. 3.9. Ваша оконная процедура может получить множество сообщений типа SB_LINEUP, SB_PAGEUP, SB_LINEDOWN или SB_PSGEDOWN, если кнопка мыши остается нажатой при перемещении по полосе прокрутки. Сообщение SB_ENDSCROLL показывает, что кнопка мыши отпущена. Как правило, сообщения SB_ENDSCROLL можно игнорировать.

Если младшее слово параметра *wParam* равно SB_THUMBTRACK или SB_THUMBPOSITION, то старшее слово *wParam* определяет текущее положение полосы прокрутки. Это положение находится между минимальным и максимальным значениями диапазона полосы прокрутки. Во всех других случаях при работе с полосами прокрутки старшее слово *wParam* следует игнорировать. Вы также можете игнорировать параметр *lParam*, который обычно используется для полос прокрутки, создаваемых в окнах диалога.

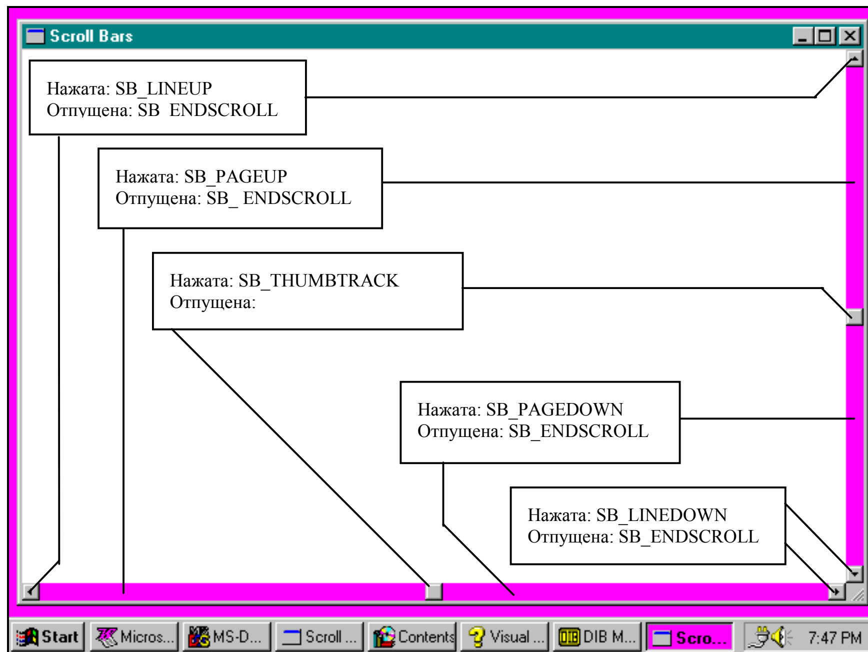


Рис. 3.9 Значения идентификаторов для параметра *wParam* сообщений полосы прокрутки

В документации по Windows указано, что младшее слово *wParam* может также быть равно SB_TOP или SB_BOTTOM. Оно показывает, что полоса прокрутки была переведена в свое максимальное или минимальное положение. Однако, вы никогда не получите эти значения для полосы прокрутки, созданной в окне вашего приложения.

Обработка сообщений SB_THUMBTRACK и SB_THUMBPOSITION весьма проблематична. Если вы устанавливаете большой диапазон полосы прокрутки, а пользователь быстро перемещает бегунок по полосе, то Windows отправит вашей оконной функции множество сообщений SB_THUMBTRACK. Ваша программа столкнется с проблемой обработки этих сообщений. По этой причине в большинстве приложений Windows эти сообщения игнорируются, а действия предпринимаются только при получении сообщения SB_THUMBPOSITION, которое означает, что бегунок оставлен в покое.

Однако, если у вас есть возможность быстро обновлять содержимое экрана, вы можете захотеть включить в программу обработку сообщений SB_THUMBTRACK. Но знайте, что те пользователи, которые обнаружат, что ваша программа мгновенно реагирует на перемещение бегунка по полосе прокрутки, несомненно будут пытаться двигать его как можно быстрее, чтобы понаблюдать, сможет ли программа отследить это движение — и они будут несказанно удовлетворены, если этого не произойдет.

Прокрутка в программе SYSMETS

Достаточно объяснений. Самое время использовать этот материал на практике. Давайте начнем с простого. Начнем мы с вертикальной прокрутки, поскольку требуется она гораздо чаще. Горизонтальная прокрутка может подождать. SYSMETS2 представлена на рис. 3.10.

Обновленный вызов функции *CreateWindow* добавляет вертикальную полосу прокрутки к окну, благодаря включению в описание стиля окна в *CreateWindow* идентификатора WS_VSCROLL:

```
WS_OVERLAPPEDWINDOW | WS_VSCROLL
```

SYSMETS2.MAK

```
#------------------------
# SYSMETS2.MAK make file
#------------------------

sysmets2.exe : sysmets2.obj
     $(LINKER) $(GUIFLAGS) -OUT:sysmets2.exe sysmets2.obj $(GUILIBS)

sysmets2.obj : sysmets2.c sysmets.h
     $(CC) $(CFLAGS) sysmets2.c
```

SYSMETS2.C

```
/*-----------------------------------------------------
   SYSMETS2.C -- System Metrics Display Program No. 2
                 (c) Charles Petzold, 1996
  -----------------------------------------------------*/

#include <windows.h>
#include <string.h>
#include "sysmets.h"

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow)
{
        static char szAppName[] = "SysMets2";
        HWND        hwnd;
        MSG         msg;
        WNDCLASSEX  wndclass;

        wndclass.cbSize        = sizeof(wndclass);
        wndclass.style         = CS_HREDRAW | CS_VREDRAW;
        wndclass.lpfnWndProc   = WndProc;
        wndclass.cbClsExtra    = 0;
        wndclass.cbWndExtra    = 0;
        wndclass.hInstance     = hInstance;
        wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
        wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
        wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
        wndclass.lpszMenuName  = NULL;
        wndclass.lpszClassName = szAppName;
        wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

        RegisterClassEx(&wndclass);

        hwnd = CreateWindow(
                        szAppName,
                        "Get System Metrics No. 2",
                        WS_OVERLAPPEDWINDOW | WS_VSCROLL,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        NULL, NULL, hInstance, NULL
        );

        ShowWindow(hwnd, iCmdShow);
        UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
        static int  cxChar, cxCaps, cyChar, cyClient, iVscrollPos;
        char        szBuffer[10];
        HDC         hdc;
        int         i, y;
        PAINTSTRUCT ps;
        TEXTMETRIC  tm;
```

```
switch(iMsg)
{
case WM_CREATE :
        hdc = GetDC(hwnd);

        GetTextMetrics(hdc, &tm);
        cxChar = tm.tmAveCharWidth;
        cxCaps =(tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2;
        cyChar = tm.tmHeight + tm.tmExternalLeading;

        ReleaseDC(hwnd, hdc);

        SetScrollRange(hwnd, SB_VERT, 0, NUMLINES, FALSE);
        SetScrollPos  (hwnd, SB_VERT, iVscrollPos, TRUE);
        return 0;

case WM_SIZE :
        cyClient = HIWORD(lParam);
        return 0;

case WM_VSCROLL :
        switch(LOWORD(wParam))
        {
        case SB_LINEUP :
                iVscrollPos -= 1;
                break;

        case SB_LINEDOWN :
                iVscrollPos += 1;
                break;

        case SB_PAGEUP :
                iVscrollPos -= cyClient / cyChar;
                break;

        case SB_PAGEDOWN :
                iVscrollPos += cyClient / cyChar;
                break;

        case SB_THUMBPOSITION :
                iVscrollPos = HIWORD(wParam);
                break;

        default :
                break;
        }
        iVscrollPos = max(0, min(iVscrollPos, NUMLINES));

        if (iVscrollPos != GetScrollPos(hwnd, SB_VERT))
        {
                SetScrollPos(hwnd, SB_VERT, iVscrollPos, TRUE);
                InvalidateRect(hwnd, NULL, TRUE);
        }
       return 0;

case WM_PAINT :
        hdc = BeginPaint(hwnd, &ps);

        for(i = 0; i < NUMLINES; i++)
        {
                y = cyChar *(1 - iVscrollPos + i);

                TextOut(
                        hdc, cxChar, y,
```

```
                        sysmetrics[i].szLabel,
                        strlen(sysmetrics[i].szLabel)
                );

                TextOut(
                        hdc, cxChar + 22 * cxCaps, y,
                        sysmetrics[i].szDesc,
                        strlen(sysmetrics[i].szDesc)
                );

                SetTextAlign(hdc, TA_RIGHT | TA_TOP);

                TextOut(
                        hdc, cxChar + 22 * cxCaps + 40 * cxChar, y,
                        szBuffer,
                        wsprintf(
                                szBuffer, "%5d",
                                GetSystemMetrics(sysmetrics[i].iIndex)
                        )
                );

                SetTextAlign(hdc, TA_LEFT | TA_TOP);
            }

            EndPaint(hwnd, &ps);
            return 0;

        case WM_DESTROY :
            PostQuitMessage(0);
            return 0;
    }

    return DefWindowProc(hwnd, iMsg, wParam, lParam);
}
```

Рис. 3.10 Программа SYSMETS2

К оконной процедуре *WndProc* добавляются две строки для установки диапазона и положения вертикальной полосы прокрутки во время обработки сообщения WM_CREATE:

```
SetScrollRange(hwnd, SB_VERT, 0, NUMLINES, FALSE);
SetScrollPass(hwnd, SB_VERT, iVscrollPos, TRUE);
```

Структура *sysmetrics* содержит NUMLINES строк текста, поэтому границы диапазона полосы прокрутки устанавливается от 0 до NUMLINES. Каждое положение полосы прокрутки соответствует строке текста в верхней части рабочей области. Если бегунок полосы прокрутки находится в положении 0, то в окне сверху остается пустая строка. При увеличении значения, определяющего положение полосы прокрутки, путем перемещения бегунка вниз, текст будет подниматься. Если положение бегунка полосы прокрутки находится в крайнем нижнем положении полосы, то последняя строка текста находится на самом верху рабочей области окна.

Для упрощения обработки сообщений WM_VSCROLL статическая переменная с именем *iVscrollPos* определяется внутри оконной процедуры *WndProc*. Эта переменная соответствует текущему положению бегунка полосы прокрутки. Что касается сообщений SB_LINEUP и SB_LINEDOWN, то все, что нужно сделать, это изменить положение прокрутки на 1. При получении сообщений SB_PAGEUP и SB_PAGEDOWN появляется возможность перемещать текст постранично (вернее "поэкранно"), или, что то же самое, изменять положение полосы прокрутки на величину, равную *cyClient* деленную на *cyChar*. Для SB_THUMBPOSITION новое положение бегунка определяется старшим словом *wParam*. Сообщения SB_ENDSCROLL и SB_THUMBTRACK игнорируются.

Затем параметр *iVscrollPos* устанавливается с использованием макросов *min* и *max*, чтобы гарантировать, что значение параметра будет находиться между минимальным и максимальным значениями диапазона. Если положение прокрутки изменилось, оно обновляется с помощью функции *SetScrollPos* и все окно делается недействительным путем вызова *InvalidateRect*.

При вызове функции *InvalidateRect* вырабатывается сообщение WM_PAINT. Когда исходная программа SYSMETS1 обрабатывала сообщение WM_PAINT, координата y для каждой строки рассчитывалась следующим образом:

```
cyChar *(1 + i)
```

В SYSMETS2 эта формула выглядит так:

```
cyChar *(1 - iVscrollPos +i)
```

Цикл по прежнему выводит на экран NUMLINES строк текста, но для значений параметра *iVscrollPos* от 2 и выше, цикл начинает выводить строки за пределами верхней границы рабочей области. Поскольку строки находятся за пределами рабочей области, Windows эти строки на экран не выводит.

Вам уже говорилось, что мы начнем с простой программы. Она достаточно неэкономична и неэффективна. В дальнейшем мы ее модифицируем, но сначала разберемся с тем, как обновить рабочую область после сообщения WM_VSCROLL.

Структурирование вашей программы для рисования

Оконная процедура в SYSMETS2 не перерисовывает рабочую область после обработки сообщения полосы прокрутки. Вместо этого она вызывает функцию *InvalidateRect* для того, чтобы сделать рабочую область недействительной. Это заставляет Windows поместить сообщение WM_PAINT в очередь сообщений.

Было бы хорошо организовать вашу Windows-программу таким образом, чтобы все действия по перерисовке рабочей зоны окна осуществлялись в ответ на сообщение WM_PAINT. Поскольку ваша программа должна быть готова перерисовать всю рабочую область окна в любое время по получении сообщения WM_PAINT, вам, вероятно, следует просто продублировать соответствующий код в других рисующих частях программы.

Вначале вы можете возмутиться по поводу этого высказывания, поскольку это так непохоже на нормальное программирование для PC. Нельзя отрицать, что иногда рисование в ответ на другие, отличные от WM_PAINT сообщения гораздо более удобно. (Программа KEYLOOK в главе 5 — именно такой пример). Но во многих случаях это просто не нужно, и, поскольку вы теперь специалист в области сбора всей необходимой для рисования информации в ответ на сообщение WM_PAINT, вы должны быть довольны результатами своей работы. Однако, вашей программе часто будет необходимо перерисовать только отдельную область экрана при обработке другого, отличного от WM_PAINT сообщения. В таком случае становится удобной функция *InvalidateRect*. Вы можете использовать ее, чтобы делать недействительными конкретные прямоугольные зоны рабочей области или рабочую область в целом.

Для некоторых приложений может оказаться недостаточным просто пометить некоторые зоны окна как недействительные для выработки сообщения WM_PAINT. После вызова функции *InvalidateRect*, Windows помещает сообщение WM_PAINT в очередь сообщений, и оконная процедура в конце концов его обрабатывает. Однако, Windows обращается с сообщениями WM_PAINT как с сообщениями, имеющими наименьший приоритет, поэтому, если в системе происходит множество каких-то событий, то эти события могут произойти до того, как оконная процедура получит сообщения WM_PAINT. Каждый из вас видел в программах пустые белые "дыры", появляющиеся на месте окон диалога.

Если вы хотите немедленно обновить недействительную область, вы можете после вызова функции *InvalidateRect* вызвать *UpdateWindow*:

```
UpdateWindow(hwnd);
```

Вызов *UpdateWindow* приводит к немедленному вызову оконной процедуры с сообщением WM_PAINT в случае, если существует какая-либо недействительная зона в рабочей области окна. (Если вся рабочая область действительна, вызова оконной процедуры не произойдет.) Такие сообщения WM_PAINT минуют очередь сообщений. Оконная процедура вызывается прямо из Windows. После того как оконная процедура завершает перерисовку, она заканчивает свою работу, и Windows возвращает управление в программу на следующую после вызова *UpdateWindow* инструкцию.

Заметьте, что *UpdateWindow* — это та же самая функция, которая использовалась в *WinMain* для выработки первого сообщения WM_PAINT. При создании окна вся рабочая область является недействительной. *UpdateWindow* заставляет оконную процедуру перерисовать ее.

Создание улучшенной прокрутки

Поскольку программа SYSMETS2 слишком неэффективна, как модель для повторения ее в других программах, давайте модифицируем ее. SYSMETS3 — наша окончательная версия программы SYSMETS этой главы — показана на рис. 3.11. В этой версии добавлена горизонтальная полоса прокрутки для прокрутки рабочей области влево и вправо, и перерисовка рабочей области организована более эффективно.

SYSMETS3.MAK

```
#------------------------
# SYSMETS3.MAK make file
```

```
#-----------------------

sysmets3.exe : sysmets3.obj
     $(LINKER) $(GUIFLAGS) -OUT:sysmets3.exe sysmets3.obj $(GUILIBS)

sysmets3.obj : sysmets3.c sysmets.h
     $(CC) $(CFLAGS) sysmets3.c
```

SYSMETS3.C

```
/*----------------------------------------------------
   SYSMETS3.C -- System Metrics Display Program No. 3
                 (c) Charles Petzold, 1996
  ----------------------------------------------------*/

#include <windows.h>
#include "sysmets.h"

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow)
{
        static char szAppName[] = "SysMets3";
        HWND        hwnd;
        MSG         msg;
        WNDCLASSEX  wndclass;

        wndclass.cbSize        = sizeof(wndclass);
        wndclass.style         = CS_HREDRAW | CS_VREDRAW;
        wndclass.lpfnWndProc   = WndProc;
        wndclass.cbClsExtra    = 0;
        wndclass.cbWndExtra    = 0;
        wndclass.hInstance     = hInstance;
        wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
        wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
        wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
        wndclass.lpszMenuName  = NULL;
        wndclass.lpszClassName = szAppName;
        wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

        RegisterClassEx(&wndclass);

        hwnd = CreateWindow(
                        szAppName,
                        "Get System Metrics No. 3",
                        WS_OVERLAPPEDWINDOW | WS_VSCROLL | WS_HSCROLL,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        NULL, NULL, hInstance, NULL
        );

        ShowWindow(hwnd, iCmdShow);
        UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
        static int  cxChar, cxCaps, cyChar, cxClient, cyClient, iMaxWidth,
```

```
             iVscrollPos, iVscrollMax, iHscrollPos, iHscrollMax;
char         szBuffer[10];
HDC          hdc;
int          i, x, y, iPaintBeg, iPaintEnd, iVscrollInc, iHscrollInc;
PAINTSTRUCT ps;
TEXTMETRIC  tm;

switch(iMsg)
{
case WM_CREATE :
        hdc = GetDC(hwnd);

        GetTextMetrics(hdc, &tm);
        cxChar = tm.tmAveCharWidth;
        cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2;
        cyChar = tm.tmHeight + tm.tmExternalLeading;

        ReleaseDC(hwnd, hdc);

        iMaxWidth = 40 * cxChar + 22 * cxCaps;
        return 0;

case WM_SIZE :
        cxClient = LOWORD(lParam);
        cyClient = HIWORD(lParam);

        iVscrollMax = max(0, NUMLINES + 2 - cyClient / cyChar);
        iVscrollPos = min(iVscrollPos, iVscrollMax);

        SetScrollRange(hwnd, SB_VERT, 0, iVscrollMax, FALSE);
        SetScrollPos(hwnd, SB_VERT, iVscrollPos, TRUE);

        iHscrollMax = max(0, 2 +(iMaxWidth - cxClient) / cxChar);
        iHscrollPos = min(iHscrollPos, iHscrollMax);

        SetScrollRange(hwnd, SB_HORZ, 0, iHscrollMax, FALSE);
        SetScrollPos(hwnd, SB_HORZ, iHscrollPos, TRUE);
        return 0;

case WM_VSCROLL :
        switch(LOWORD(wParam))
        {
        case SB_TOP :
                iVscrollInc = -iVscrollPos;
                break;

        case SB_BOTTOM :
                iVscrollInc = iVscrollMax - iVscrollPos;
                break;

        case SB_LINEUP :
                iVscrollInc = -1;
                break;

        case SB_LINEDOWN :
                iVscrollInc = 1;
                break;

        case SB_PAGEUP :
                iVscrollInc = min(-1, -cyClient / cyChar);
                break;

        case SB_PAGEDOWN :
                iVscrollInc = max(1, cyClient / cyChar);
```

```
                        break;

                case SB_THUMBTRACK :
                        iVscrollInc = HIWORD(wParam) - iVscrollPos;
                        break;

                default :
                        iVscrollInc = 0;
                }
                iVscrollInc = max(
                                -iVscrollPos,
                                min(iVscrollInc, iVscrollMax - iVscrollPos)
                );

                if (iVscrollInc != 0)
                {
                        iVscrollPos += iVscrollInc;
                        ScrollWindow(hwnd, 0, -cyChar * iVscrollInc, NULL, NULL);
                        SetScrollPos(hwnd, SB_VERT, iVscrollPos, TRUE);
                        UpdateWindow(hwnd);
                }
                return 0;

        case WM_HSCROLL :
                switch(LOWORD(wParam))
                {
                case SB_LINEUP :
                        iHscrollInc = -1;
                        break;

                case SB_LINEDOWN :
                        iHscrollInc = 1;
                        break;

                case SB_PAGEUP :
                        iHscrollInc = -8;
                        break;

                case SB_PAGEDOWN :
                        iHscrollInc = 8;
                        break;

                case SB_THUMBPOSITION :
                        iHscrollInc = HIWORD(wParam) - iHscrollPos;
                        break;

                default :
                        iHscrollInc = 0;
                }
                iHscrollInc = max(
                                -iHscrollPos,
                                min(iHscrollInc, iHscrollMax - iHscrollPos)
                );

                if (iHscrollInc != 0)
                {
                        iHscrollPos += iHscrollInc;
                        ScrollWindow(hwnd, -cxChar * iHscrollInc, 0, NULL, NULL);
                        SetScrollPos(hwnd, SB_HORZ, iHscrollPos, TRUE);
                }
                return 0;

        case WM_PAINT :
                hdc = BeginPaint(hwnd, &ps);
```

```
            iPaintBeg = max(0, iVscrollPos + ps.rcPaint.top / cyChar - 1);
            iPaintEnd = min(NUMLINES, iVscrollPos + ps.rcPaint.bottom / cyChar);

            for(i = iPaintBeg; i < iPaintEnd; i++)
            {
                    x = cxChar *(1 - iHscrollPos);
                    y = cyChar *(1 - iVscrollPos + i);

                    TextOut(
                            hdc, x, y,
                            sysmetrics[i].szLabel,
                            strlen(sysmetrics[i].szLabel)
                    );

                    TextOut(
                            hdc, x + 22 * cxCaps, y,
                            sysmetrics[i].szDesc,
                            strlen(sysmetrics[i].szDesc)
                    );

                    SetTextAlign(hdc, TA_RIGHT | TA_TOP);

                    TextOut(
                            hdc, x + 22 * cxCaps + 40 * cxChar, y,
                            szBuffer,
                            wsprintf(
                                    szBuffer, "%5d",
                                    GetSystemMetrics(sysmetrics[i].iIndex)
                            )
                    );

                    SetTextAlign(hdc, TA_LEFT | TA_TOP);
            }

            EndPaint(hwnd, &ps);
            return 0;

        case WM_DESTROY :
            PostQuitMessage(0);
            return 0;
        }

        return DefWindowProc(hwnd, iMsg, wParam, lParam);
}
```

Рис. 3.11 Программа SYSMETS3

Далее рассматриваются улучшения в программе SYSMETS3 и то, как они реализованы в программе:

- Вы не можете прокручивать экран так, чтобы последняя строка оказалась на самом верху рабочей зоны окна. Процесс прокрутки продолжается только до появления последней строки текста в нижней части рабочей области. Для этого необходимо, чтобы программа рассчитала новый диапазон полосы прокрутки (и, возможно, новое положение бегунка) при обработке сообщения WM_SIZE. Логика WM_SIZE вычисляет диапазон полосы прокрутки на основе числа строк текста и размера рабочей области. Такой подход приводит к уменьшению диапазона — необходимо только получить возможность вновь увидеть текст, оказавшийся вне рабочей области.

  Это решение дает интересный результат. Предположим, что рабочая область окна достаточно велика, чтобы вывести на экран весь текст, включая верхний и нижний отступы. В таком случае, и минимальное и максимальное положение диапазона полосы прокрутки будут равны 0. Что с этой информацией будет делать Windows? Она удалит полосу прокрутки из окна! Она больше не нужна. Аналогично получается, если рабочая область достаточно широка для вывода текста со строкой в 60 знаков, тогда горизонтальная полоса прокрутки не появляется на экране.

- Для каждого действия с полосой прокрутки сначала рассчитывается приращение ее текущей позиции при обработке сообщений WM_VSCROLL и WM_HSCROLL. Это значение затем используется для прокрутки имеющегося в окне содержимого с помощью вызова функции *ScrollWindow*. Эта функция имеет следующий формат:

  ```
  ScrollWindow(hwnd, xInc, yInc, pRect, pClipRect);
  ```

  Значения *xInc* и *yInc* задают величину прокрутки в пикселях. В SYSMETS3 значения *pRect* и *pClipRect* устанавливаются в NULL для указания, что необходимо прокручивать всю рабочую область. Windows делает недействительным прямоугольную зону рабочей области, открываемую операцией прокрутки. Это приводит к выдаче сообщения WM_PAINT. *InvalidateRect* больше не нужна. (Отметьте, что функция *ScrollWindow* не является процедурой GDI, поскольку ей не нужен описатель контекста устройства. Это одна из немногих функций Windows, которая меняет вид рабочей области окна, не являясь функциями GDI.)

- Обработчик сообщения WM_PAINT теперь определяет, какие строки находятся внутри недействительного прямоугольника и выводит только эти строки. Он делает это путем анализа координат верхней и нижней границ недействительного прямоугольника, хранящихся в структуре PAINTSTRUCT. Программа рисует только те строки текста, которые находятся внутри недействительного прямоугольника. Исходный текст программы более сложен, но она работает гораздо быстрее.

- Поскольку сообщения WM_PAINT стали обрабатываться быстрее, теперь, очевидно, есть смысл обрабатывать в SYSMETS3 действия SB_THUMBTRACK для сообщений WM_VSCROLL. Ранее программа игнорировала сообщения SB_THUMBTRACK (которые посылаются, когда пользователь перемещает бегунок полосы прокрутки), а реагировала только на сообщения SB_THUMBPOSITION, которые посылаются, когда пользователь прекращает перемещение бегунка. Сообщение WM_VSCROLL также приводит к вызову функции *UpdateWindow* для немедленного обновления рабочей области окна. Когда Вы перемещаете бегунок по вертикальной полосе прокрутки, SYSMETS3 непрерывно прокручивает и обновляет рабочую область. Вам самим предлагается решить, насколько быстро работает SYSMETS3 (и Windows), и насколько оправданны внесенные изменения.

Мне не нравится пользоваться мышью

В первое время существования Windows довольно многие пользователи пытались не пользоваться мышью. На самом деле в самой Windows (и во многих программах для Windows) мышь не требовалась. Хотя сегодня, в основном, персональные компьютеры без мыши ушли в прошлое вместе с монохромными дисплеями и матричными принтерами, вам по-прежнему рекомендуется писать программы, где бы действия мыши дублировались с помощью клавиатуры. Это особенно очевидно для таких фундаментальных средств, как полосы прокрутки, тем более, что в наших клавиатурах имеется целый набор клавиш, управляющих движением курсора, которые должны выполнять те же операции.

В главе 5 вы узнаете, как использовать клавиатуру и как добавить в вашу программу интерфейс клавиатуры. Вы также увидите, что SYSMETS3 обрабатывает сообщения WM_VSCROLL, когда младшее слово в *wParam* равно SB_TOP и SB_BOTTOM. Уже упоминалось, что оконная процедура не получает эти сообщения от полос прокрутки, поэтому для данного случая эти коды излишни. Когда мы в главе 5 вернемся к этой программе, вы увидите смысл во включении в программу этих действий.

Глава 4 Главное о графике

4

Графический интерфейс устройства (GDI) — подсистема Windows 95, отвечающая за отображение графики (включая текст) на видеотерминалах и принтерах. Как можно догадаться, GDI — очень важная компонента Windows. Не только приложения, разрабатываемые вами, активно используют GDI для отображения информации, но и Windows сама очень активно использует его для отображения элементов пользовательского интерфейса, таких как меню, полосы прокрутки, значки и курсоры мыши.

Вероятно, программисты, работающие с MS DOS и ранними версиями Windows по принципу "делаю, что хочу", часто пользовались возможностью работать, минуя GDI, и записывать информацию непосредственно в видеопамять. Пожалуйста, никогда не помышляйте об этом. Это может доставить вам много головной боли, привести к конфликтам с другими программами Windows 95, сделать ваши программы несовместимыми с будущими версиями операционной системы.

В отличие от некоторых новых особенностей Windows 95, GDI практически не менялся с самого начала. Внутренне Windows 1.0, в основном, состояла из трех динамически подключаемых библиотек, KERNEL (ядро — обработка задач, управление памятью, файловый ввод/вывод), USER (интерфейс пользователя) и GDI. В более поздние версии Windows были включены дополнительные функциональные возможности GDI, сохраняя в основном совместимость с существующими программами, но основа GDI осталась без изменений.

Конечно, полное описание GDI потребовало бы отдельной книги. Но это не наша задача. В этой главе вам будут даны достаточные знания о GDI для понимания программ из этой книги, использующих графику, и, некоторая другая информация, которая может вам оказаться полезной.

Концепция GDI

Графика в 32-битной Windows реализуется, в основном, функциями, экспортируемыми из GDI32.DLL, динамически подключаемой библиотеки, которая часто использует 16-битную динамически подключаемую библиотеку GDI.EXE. (Динамически подключаемые библиотеки в ранних версиях Windows чаще имели расширение EXE, а не DLL). Эти модули обращаются к различным функциям драйверов отображения — .DRV файлу для видеомониторов и, возможно, к одному или нескольким .DRV файлам драйверов принтеров или плоттеров. Видеодрайвер осуществляет доступ к аппаратуре видеомонитора или преобразует команды GDI в коды или команды, воспринимаемые различными принтерами. Разные видеоадаптеры и принтеры требуют различных файлов драйверов.

Поскольку к IBM PC совместимым компьютерам может быть подключено множество различных устройств отображения, одной из основных задач GDI является поддержка аппаратно-независимой графики. Программы для Windows должны нормально работать на любых графических устройствах отображения, которые поддерживаются Windows. GDI выполняет эту задачу, предоставляя возможность отделить ваши программы от конкретных характеристик различных устройств отображения.

Все графические устройства отображения делятся на две больших группы: растровые устройства и векторные устройства. Большинство устройств, подключаемых к PC — растровые устройства, т. е. они представляют графические образы как шаблон точек. Эта группа включает видеоадаптеры, матричные принтеры и лазерные принтеры. Группа векторных устройств, отображающих графические образы с использованием линий, в основном, состоит из плоттеров.

Большинство традиционных графических программ работает исключительно с векторами. Это значит, что программа, использующая один из таких графических языков, абстрагируется от особенностей оборудования. Устройство отображения оперирует пикселями для создания графических образов, тогда как программа при связи с интерфейсом не использует понятие пикселя. Несмотря на то, что Windows GDI — это высокоуровневая

векторная система рисования, она также может применяться и для относительно низкоуровневых манипуляций с пикселями.

С этой точки зрения, Windows GDI это такой же традиционный графический язык, как C — язык программирования. Язык C — хорошо известен своей высокой степенью переносимости относительно разных операционных систем и сред. C также хорошо известен тем, что дает программисту возможность выполнять низкоуровневые системные функции, что часто недоступно в других языках программирования высокого уровня. Также как C иногда называют "ассемблером высокого уровня", так можно считать, что GDI — это высокоуровневый интерфейс для аппаратных средств графики.

Как уже было сказано выше, по умолчанию Windows использует систему координат, основанную на пикселях. Большинство традиционных графических языков используют "виртуальные" системы координат, с границами 0 и 32767 для горизонтальной и вертикальной осей. Хотя некоторые графические языки не дают вам возможности использовать пиксельную систему координат, Windows GDI позволяет применять обе системы (также как дополнительные координатные системы на базе физических единиц измерения). Вы можете работать с виртуальной системой координат и абстрагировать вашу программу от аппаратуры или использовать систему координат устройства и приблизиться вплотную к аппаратуре.

Некоторые программисты думают, что как только начинается работа в терминах пикселей, нарушается аппаратурная независимость. В главе 3 мы уже видели, что это не всегда верно. Хитрость состоит в использовании пикселей в аппаратно-независимых образах. Это требует, чтобы язык графического интерфейса давал программе возможность определить аппаратные характеристики устройства и выполнить соответствующее согласование. Например, в программе SYSMETS мы использовали размер символа стандартного системного шрифта в пикселях для расчета пробелов в тексте на экране. Этот подход позволил программе работать на различных дисплейных адаптерах с разной разрешающей способностью, размерами текста и коэффициентом искажения. В этой главе будут показаны другие методы определения размеров дисплея.

Когда-то работа под Windows с монохромным дисплеем была нормой. Даже еще совсем недавно дисплеи переносных компьютеров были ограничены оттенками серого. Поэтому GDI был разработан так, что можно писать программу, не беспокоясь по поводу цветов — Windows может преобразовать цвета в оттенки серого. И сегодня видеомониторы, используемые с Windows 95, имеют различные цветовые возможности (16 цветов, 256 цветов, "полноцветные"), а большинство пользователей работают с монохромными принтерами. Можно применять эти устройства вслепую, но ваша программа вполне способна определить, сколько цветов доступно на конкретном дисплее, а затем наилучшим образом воспользоваться устройством.

Есть вероятность, что в то самое время когда вы пишете программы на C и мастерски решаете проблемы переносимости с компьютера на компьютер, в ваши программы для Windows, естественно непреднамеренно, будут включены аппаратно-зависимые фрагменты. Это часть цены за то, что вы не полностью изолированы от аппаратуры. Мы рассмотрим некоторые из этих аппаратно-зависимых аспектов в данной главе.

Также следует упомянуть об ограничениях Windows GDI. GDI не в состоянии (в настоящее время) сделать все, что вы можете пожелать. Несмотря на то, что существует возможность перемещения графических объектов по экрану, GDI, в основном, статическая система отображения с ограниченной поддержкой анимации. В реализации Windows 95 GDI не обеспечивает трехмерных представлений или возможности вращения объектов. Например, при рисовании эллипса, его оси должны быть параллельны горизонтальной и вертикальной осям. Несмотря на то, что некоторые графические языки используют числа с плавающей точкой для представления виртуальных координат, Windows 95 из соображений производительности всегда использует 16-разрядные знаковые целые. Это особенность Windows 95. Windows NT поддерживает 32-разрядные координаты.

Структура GDI

С точки зрения программиста GDI состоит из нескольких сотен функций и нескольких связанных с ними типов данных, макросов и структур. Но прежде, чем рассматривать некоторые из этих функций подробно, следует остановиться на общей структуре GDI .

Типы функций

В основном, функции GDI могут быть разбиты на несколько крупных групп. Это группы не имеют четких границ и частично перекрываются. Все они, тем не менее, перечислены ниже:

- *Функции, которые получают (или создают) и освобождают (или уничтожают) контекст устройства.* Как уже указывалось в главе 3, вам для рисования необходим описатель контекста устройства. Функции *GetDC* и *ReleaseDC* позволяют вам сделать это внутри обработчиков сообщений, отличных от WM_PAINT. Функции *BeginPaint* и *EndPaint* (хотя технически — это часть подсистемы USER в Windows) используются в теле обработчика сообщения WM_PAINT для рисования. Некоторые другие функции, относящиеся к работе с контекстом устройства, мы рассмотрим ниже.

- *Функции, которые получают информацию о контексте устройства.* Вспомним, в программе SYSMETS в главе 3 мы использовали функцию *GetTextMetrics* для получения информации о размерах выбранного в контексте устройства шрифта. Далее в этой главе мы рассмотрим программу DEVCAPS1, с помощью которой можно получить самую общую информацию о контексте устройства.

- *Функции рисования.* Очевидно, из всех предварительно рассмотренных функций — это одна из самых важных. В главе 3 мы использовали функцию *TextOut* для отображения текста в рабочей области окна. Как мы увидим далее, другие функции GDI позволяют рисовать линии, залитые области, растровые образы.

- *Функции, которые устанавливают и получают атрибуты контекста устройства.* Атрибут контекста устройства определяет различные особенности работы функции рисования. Например, вы используете функцию *SetTextColor* для задания любого текста, выводимого с использованием функции *TextOut* (или любой другой функции вывода текста). В программах SYSMETS в главе 3 мы использовали функцию *SetTextAlign* для того, чтобы сообщить GDI, что начальное положение текстовой строки при вызове функции *TextOut* должно быть справа, по умолчанию — левое начальное положение. Все атрибуты контекста устройства имеют значение по умолчанию, которое устанавливается, при получении контекста устройства. Для всех функций *Set* есть функции *Get*, позволяющие получить текущее значение атрибута контекста устройства.

- *Функции, которые работают с объектами GDI.* Именно эти функции вносят в GDI некоторый беспорядок. Сначала пример: по умолчанию любые линии, которые вы рисуете, используя GDI, — сплошные и стандартной ширины. Вы хотите изобразить линии более широкими или сделать их штрихпунктирными. Ширина линии и стиль линии не являются атрибутами контекста устройства. Это характеристики "логического карандаша". Вы можете создать логический карандаш, указав данные характеристики в функциях *CreatePen*, *CreatePenIndirect*, *ExtCreatePen*. Эти функции возвращают описатель логического карандаша. (Хотя считается, что эти функции являются частью GDI, в отличие от большинства функций GDI они не требуют описателя контекста устройства.) Для использования логического карандаша вы "выбираете" описатель в контекст устройства. С этого момента все рисуемые линии будут отображаться с использованием этого карандаша. Затем вы отменяете выбор объекта "карандаш" и уничтожаете его. Кроме карандашей, вы также используете объекты GDI для создания кистей, которыми зарисовываются замкнутые области для создания шрифтов, растровых образов и других объектов GDI, о которых будет рассказано в этой главе.

Примитивы GDI

Типы графических объектов, выводимых на экран или принтер, которые могут быть разделены на несколько категорий, часто называют "примитивами". Это:

- *Прямые (отрезки) и кривые.* Прямые — основа любой векторной графической системы. GDI поддерживает прямые линии, прямоугольники, эллипсы (включая окружности), дуги, являющиеся частью кривой эллипса, и сплайны Безье. Все они будут рассмотрены в этой главе. Более сложные кривые могут быть изображены как ломаные линии, которые состоят из очень коротких прямых, определяющих кривые. Линии рисуются с использованием карандаша, выбранного в контексте устройства.

- *Закрашенные области.* Если набор прямых и кривых линий ограничивает со всех сторон некоторую область, то она может быть закрашена с использованием объекта GDI "кисть", выбранного в контексте устройства. Эта кисть может быть сплошной, штриховой (состоящей из горизонтальных, вертикальных или диагональных штрихов) или шаблонной, заполняющей область горизонтально и вертикально.

- *Битовые шаблоны (растровые шаблоны, растровые образы).* Битовые шаблоны — это двумерный массив битов, соответствующий пикселям устройства отображения. Это базовый инструмент в растровой графике. Битовые образы используются, в основном, для отображения сложных (часто из реального мира) изображений на дисплее или принтере. Битовые образы также используются для отображения маленьких картинок, таких как значки, курсоры мыши, кнопки панели инструментов программ, которые нужно быстро нарисовать. GDI поддерживает два типа битовых образов или шаблонов — старые, но все еще используемые, аппаратно-зависимые, являющиеся объектами GDI, и новые (начиная с версии Windows 3.0) аппаратно-независимые (Device Independent Bitmap, DIB), которые могут быть сохранены в файлах на диске.

- *Текст.* Текст отличается от других математических объектов компьютерной графики. Типов текста бесконечно много. Это известно из многолетней истории типографского дела, которое многие считают искусством. Поэтому поддержка текста часто наиболее сложная часть в системах компьютерной графики, и, вместе с тем, наиболее важная. Структуры данных, используемые для описания объектов GDI — шрифтов, а также для получения информации о них — самые большие среди других структур данных в Windows. Начиная с версии Windows 3.1, GDI поддерживает шрифты TrueType, основанные на закрашенных контурах, которыми могут манипулировать другие функции GDI. Windows 95 из

соображений совместимости и экономии памяти по-прежнему поддерживает старые шрифты, основанные на битовых массивах (такие как системный шрифт по умолчанию).

Другие аспекты

Другие аспекты GDI не так легко классифицируются. Это:

- *Режимы масштабирования и преобразования.* Хотя, по умолчанию, вывод задается в пикселях, существуют и другие возможности. Режимы масштабирования GDI позволяют вам рисовать, задавая размеры в дюймах (иногда, в долях дюйма), в миллиметрах, или других удобных вам единицах измерения. (Windows NT также поддерживает привычное "преобразование пространства", задаваемое матрицей 3×3. Это дает возможность нелинейно менять размеры и вращать графические объекты. В Windows 95 это преобразование не поддерживается.)
- *Метафайлы.* Метафайл — это набор вызовов команд GDI, сохраненный в двоичном виде. Метафайлы, в основном, используются для передачи изображений векторной графики через буфер обмена (clipboard).
- *Регионы.* Регион — это сложная область, состоящая из любых фигур, и обычно задаваемая как булева комбинация простых регионов. Регионы, как правило, хранятся внутри GDI как ряды скан-линий, независимо от любой комбинации отрезков, которые могут быть использованы для задания регионов.
- *Пути.* Путь — это набор отрезков и кривых, хранящихся внутри GDI. Они могут использоваться для рисования, закрашивания и при отсечении. Пути могут быть преобразованы в регионы.
- *Отсечение.* Рисование может быть ограничено некоторой областью рабочего пространства окна. Это и называется отсечением, область отсечения может быть прямоугольной или любой другой, какую вы можете описать математически как набор коротких отрезков. Отсечение, как правило, задается регионом или путем.
- *Палитры.* Использование привычных палитр обычно ограничено способностью дисплея показывать не более 256 цветов. Windows резервирует только 20 из этих цветов для использования системой. Вы можете изменять другие 236 цветов для точного отображения красок предметов реального мира как битовые образы.
- *Печать.* Несмотря на то, что эта глава посвящена отображению на экране дисплея, все, чему вы научитесь здесь, относится и к принтерам. (Смотри главу 15, где рассматривается печать.)

Контекст устройства

Перед тем, как начать рисовать, рассмотрим контекст устройства более подробно, чем в главе 3.

Если вы хотите рисовать на устройстве графического вывода (экране дисплея или принтере), сначала надо получить описатель контекста устройства (device context, DC). Передавая этот описатель, Windows тем самым дает вам право на использование самого устройства. Затем вы включаете этот описатель как параметр в функции GDI для того, чтобы сообщить Windows, на каком устройстве вы собираетесь рисовать.

Контекст устройства содержит много текущих атрибутов, определяющих поведение функций GDI при работе с устройством. Эти атрибуты позволяют включать в вызовы функций GDI только начальные координаты или размеры, и ничего больше из того, что требуется для отображения объекта на устройстве. Например, когда вы вызываете функцию *TextOut*, в ее параметрах вам надо указать только описатель контекста устройства, начальные координаты, сам выводимый текст и его длину. Вам не нужно указывать шрифт, цвет текста, цвет фона и межсимвольное расстояние, потому что эти атрибуты являются частью контекста устройства. Когда вы хотите изменить один из этих атрибутов, вы вызываете функцию, изменяющую значение атрибута в контексте устройства. Последующие вызовы функции *TextOut* будут использовать измененные значения атрибутов.

Получение описателя контекста устройства

Windows предоставляет несколько методов для получения описателя контекста устройства. Если вы получаете описатель контекста устройства в теле обработчика сообщения, вы должны освободить его (удалить, вернуть системе) перед выходом из оконной процедуры. После того, как вы освободите описатель контекста устройства, его значение теряет смысл.

Наиболее общий метод получения контекста устройства и его освобождения состоит в использовании функций *BeginPaint* и *EndPaint* при обработке сообщения WM_PAINT:

```
hdc = BeginPaint(hwnd, &ps);
[другие строки программы]
EndPaint(hwnd, &ps);
```

Переменная *ps* — это структура типа PAINTSTRUCT. Поле *hdc* этой структуры — это описатель контекста устройства, который возвращается функцией *BeginPaint*. Структура PAINTSTRUCT содержит также структуру типа RECT с именем *rcPaint* (прямоугольник), определяющую прямоугольную область, содержащую недействительный (требующий перерисовки) регион клиентской области окна. Получив описатель контекста устройства от функции *BeginPaint*, вы можете рисовать только в пределах этого региона. Функция *BeginPaint* делает этот регион действительным.

Программы для Windows могут также получать описатель контекста устройства в теле обработчика сообщения, отличного от WM_PAINT:

```
hdc = GetDC(hwnd);
[другие строки программы]
ReleaseDC(hwnd, hdc);
```

Полученный контекст устройства с описателем *hwnd* относится к клиентской (рабочей) области окна. Основная разница между использованием этих функций и комбинации функций *BeginPaint* и *EndPaint* состоит в том, что вы можете рисовать в пределах всей рабочей области окна, используя описатель контекста устройства, возвращенный функцией *GetDC*. Кроме того, функции *GetDC* и *ReleaseDC* не делают действительным (не требующим перерисовки) ни один недействительный регион клиентской области окна.

Программы для Windows могут также получать описатель контекста устройства, относящийся ко всему окну программы, а не только к его клиентской области:

```
hdc = GetWindowDC(hwnd);
[другие строки программы]
ReleaseDC(hwnd. hdc);
```

Этот контекст устройства включает заголовок окна, меню, полосы прокрутки и рамку окна в дополнение к клиентской области. Функция *GetWindowDC* редко используется в приложениях. Если вы хотите поэкспериментировать с ней, то вам следует обработать сообщение WM_NCPAINT ("nonclient paint", рисование неклиентской области), которое генерируется Windows для перерисовки неклиентской области окна.

Функции *BeginPaint*, *GetDC* и *GetWindowDC* получают контекст устройства, связанный с конкретным окном на экране. Более общая функция для получения описателя контекста устройства — это функция *CreateDC*:

```
hdc = CreateDC(pszDriver, pszDevice, pszOutput, pData);
[другие строки программы]
DeleteDC(hdc);
```

Например, вы можете получить описатель контекста устройства всего дисплея так:

```
hdc = CreateDC("DISPLAY", NULL, NULL, NULL);
```

Запись вне вашего окна обычно не принята, но это удобно для некоторых редко используемых приложений. (Хотя это и не документировано, вы можете получить описатель контекста устройства для экрана дисплея посредством вызова функции *GetDC* с параметром NULL.) В главе 15 мы будем использовать эту функцию для получения описателя контекста устройства принтера.

Иногда вам нужно только получить некоторую информацию о контексте устройства, и не надо ничего рисовать. В этих случаях вы можете получить описатель так называемого "информационного контекста" (information context), используя функцию *CreateIC*. Параметры этой функции такие же, как у функции *CreateDC*, например:

```
hdcInfo = CreateIC("DISPLAY", NULL, NULL, NULL);
[другие строки программы]
DeleteDC(hdcInfo);
```

Вы не можете осуществлять вывод на устройство, используя информационный контекст.

При работе с битовыми образами иногда может быть полезно получить "контекст памяти" (memory device context):

```
hdcMem = CreateCompatibleDC(hdc);
[другие строки программы]
DeleteDC(hdcMem);
```

Это достаточно общая концепция. Главное, что вам надо сделать, это выбрать битовый образ в контекст памяти, а затем вызвать функцию GDI для рисования битового образа. Мы обсудим это позднее в данной главе и используем рассмотренную методику в программе GRAFMENU из главы 10.

Как уже упоминалось раньше, метафайл — это набор вызовов GDI в двоичном виде. Вы можете создать метафайл, получая контекст метафайла:

```
hdcMeta = CreateMetaFile(pszFilename);
[другие строки программы]
hmf = CloseMetaFile(hdcMeta);
```

Пока контекст метафайла действителен вызов GDI, который вы осуществляете, используя *hdcMeta*, не вызывает вывода на устройство, а записывается в метафайл. Когда вы вызываете *CloseMetaFile*, описатель контекста становится недействительным. Функция возвращает описатель метафайла (*hmf*).

Получение информации из контекста устройства

Контекст устройства обычно описывает такие физические устройства как видеотерминалы или принтеры. Часто вам необходимо получить информацию об одном из этих устройств, такую как размер экрана (в терминах пикселей и физических единицах измерения) и его цветовые возможности. Вы можете получить эту информацию посредством вызова функции *GetDeviceCaps*:

```
iValue = GetDeviceCaps(hdc, iIndex);
```

Параметр *iIndex* — один из 28 идентификаторов, определенных в заголовочном файле Windows. Например, значение *iIndex* равное HORZRES заставляет функцию *GetDeviceCaps* вернуть ширину устройства в пикселях; значение VERTRES — высоту устройства в пикселях. Если *hdc* является описателем контекста устройства дисплея, то эту же информацию вы можете получить от функции *GetSystemMetrics*. Если *hdc* является описателем контекста устройства принтера, то тогда уже функция *GetDeviceCaps* возвращает высоту и ширину рабочей области принтера в пикселях.

Вы можете также использовать функцию *GetDeviceCaps* для определения возможностей устройства по обработке различных типов графики. Это неважно для видеомониторов, но становится очень важным при работе с принтерами. Например, большинство плоттеров не способны отображать битовые образы, и *GetDeviceCaps* сообщит вам об этом.

Программа DEVCAPS1

Программа DEVCAPS1, приведенная на рис. 4.1, частично отображает информацию, доступную посредством вызова *GetDeviceCaps* с использованием контекста устройства для дисплея. (Вторая, расширенная версия программы, DEVCAPS2, будет приведена в главе 15 для получения информации о принтере.)

DEVCAPS1.MAK

```
#------------------------
# DEVCAPS1.MAK make file
#------------------------

devcaps1.exe : devcaps1.obj
     $(LINKER) $(GUIFLAGS) -OUT:devcaps1.exe devcaps1.obj $(GUILIBS)

devcaps1.obj : devcaps1.c
     $(CC) $(CFLAGS) devcaps1.c
```

DEVCAPS1.C

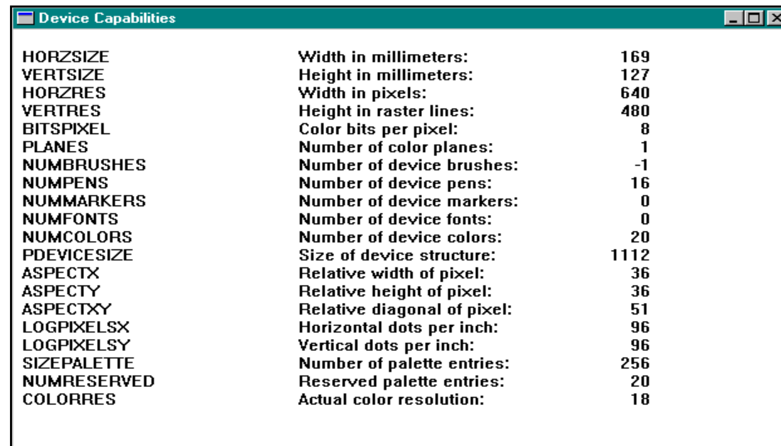
```
/*---------------------------------------------------------
   DEVCAPS1.C -- Device Capabilities Display Program No. 1
                 (c) Charles Petzold, 1996
  ---------------------------------------------------------*/

#include <windows.h>
#include <string.h>

#define NUMLINES((int)(sizeof devcaps / sizeof devcaps [0]))

struct
{
        int  iIndex;
        char *szLabel;
        char *szDesc;
} devcaps [] =
{
        HORZSIZE,      "HORZSIZE",     "Width in millimeters:",
        VERTSIZE,      "VERTSIZE",     "Height in millimeters:",
        HORZRES,       "HORZRES",      "Width in pixels:",
        VERTRES,       "VERTRES",      "Height in raster lines:",
```

```
        BITSPIXEL,     "BITSPIXEL",    "Color bits per pixel:",
        PLANES,        "PLANES",       "Number of color planes:",
        NUMBRUSHES,    "NUMBRUSHES",   "Number of device brushes:",
        NUMPENS,       "NUMPENS",      "Number of device pens:",
        NUMMARKERS,    "NUMMARKERS",   "Number of device markers:",
        NUMFONTS,      "NUMFONTS",     "Number of device fonts:",
        NUMCOLORS,     "NUMCOLORS",    "Number of device colors:",
        PDEVICESIZE,   "PDEVICESIZE",  "Size of device structure:",
        ASPECTX,       "ASPECTX",      "Relative width of pixel:",
        ASPECTY,       "ASPECTY",      "Relative height of pixel:",
        ASPECTXY,      "ASPECTXY",     "Relative diagonal of pixel:",
        LOGPIXELSX,    "LOGPIXELSX",   "Horizontal dots per inch:",
        LOGPIXELSY,    "LOGPIXELSY",   "Vertical dots per inch:",
        SIZEPALETTE,   "SIZEPALETTE",  "Number of palette entries:",
        NUMRESERVED,   "NUMRESERVED",  "Reserved palette entries:",
        COLORRES,      "COLORRES",     "Actual color resolution:"
};

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow)
{
        static char szAppName[] = "DevCaps1";
        HWND        hwnd;
        MSG         msg;
        WNDCLASSEX  wndclass;

        wndclass.cbSize        = sizeof(wndclass);
        wndclass.style         = CS_HREDRAW | CS_VREDRAW;
        wndclass.lpfnWndProc   = WndProc;
        wndclass.cbClsExtra    = 0;
        wndclass.cbWndExtra    = 0;
        wndclass.hInstance     = hInstance;
        wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
        wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
        wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
        wndclass.lpszMenuName  = NULL;
        wndclass.lpszClassName = szAppName;
        wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

        RegisterClassEx(&wndclass);

        hwnd = CreateWindow(
                        szAppName,
                        "Device Capabilities",
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        NULL, NULL, hInstance, NULL
        );

        ShowWindow(hwnd, iCmdShow);
        UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
```

```
        static int  cxChar, cxCaps, cyChar;
        char        szBuffer[10];
        HDC         hdc;
        int         i;
        PAINTSTRUCT ps;
        TEXTMETRIC  tm;

        switch(iMsg)
        {
        case WM_CREATE:
                hdc = GetDC(hwnd);

                GetTextMetrics(hdc, &tm);
                cxChar = tm.tmAveCharWidth;
                cxCaps =(tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2;
                cyChar = tm.tmHeight + tm.tmExternalLeading;

                ReleaseDC(hwnd, hdc);
                return 0;

        case WM_PAINT:
                hdc = BeginPaint(hwnd, &ps);

                for(i = 0; i < NUMLINES; i++)
                {
                        TextOut(
                                hdc, cxChar, cyChar *(1 + i),
                                devcaps[i].szLabel,
                                strlen(devcaps[i].szLabel)
                        );

                        TextOut(
                                hdc, cxChar + 22 * cxCaps, cyChar *(1 + i),
                                devcaps[i].szDesc,
                                strlen(devcaps[i].szDesc)
                        );

                        SetTextAlign(hdc, TA_RIGHT | TA_TOP);

                        TextOut(
                                hdc, cxChar + 22 * cxCaps + 40 * cxChar,
                                cyChar *(1 + i), szBuffer,
                                wsprintf(
                                        szBuffer, "%5d",
                                        GetDeviceCaps(hdc, devcaps[i].iIndex)
                                )
                        );

                        SetTextAlign(hdc, TA_LEFT | TA_TOP);
                }

                EndPaint(hwnd, &ps);
                return 0;

        case WM_DESTROY:
                PostQuitMessage(0);
                return 0;
        }

        return DefWindowProc(hwnd, iMsg, wParam, lParam);
}
```

Рис. 4.1 Программа DEVCAPS1

Как вы можете видеть, эта программа похожа на программу SYSMETS1, приведенную в главе 3. Чтобы уменьшить размер программы, в нее не включены полосы прокрутки, и поэтому информация уместилась на одном экране. Результаты работы программы для 256-цветного VGA монитора приведены на рис. 4.2.



HORZSIZE	Width in millimeters:	169
VERTSIZE	Height in millimeters:	127
HORZRES	Width in pixels:	640
VERTRES	Height in raster lines:	480
BITSPIXEL	Color bits per pixel:	8
PLANES	Number of color planes:	1
NUMBRUSHES	Number of device brushes:	-1
NUMPENS	Number of device pens:	16
NUMMARKERS	Number of device markers:	0
NUMFONTS	Number of device fonts:	0
NUMCOLORS	Number of device colors:	20
PDEVICESIZE	Size of device structure:	1112
ASPECTX	Relative width of pixel:	36
ASPECTY	Relative height of pixel:	36
ASPECTXY	Relative diagonal of pixel:	51
LOGPIXELSX	Horizontal dots per inch:	96
LOGPIXELSY	Vertical dots per inch:	96
SIZEPALETTE	Number of palette entries:	256
NUMRESERVED	Reserved palette entries:	20
COLORRES	Actual color resolution:	18

Рис. 4.2 Вывод программы DEVCAPS1 для 256-цветного VGA

Размер устройства

Наиболее важная информация, которую может получить ваша Windows-программа об устройстве отображения от функции *GetDeviceCaps*, это размеры области отображения (в миллиметрах и пикселях) и коэффициент растяжения пикселя. Эти данные могут помочь в масштабировании изображений перед их отображением.

Значения HORZSIZE и VERTSIZE — это ширина и высота области отображения в миллиметрах. Конечно, драйвер Windows в действительности не знает точных размеров дисплея, подключенного к вашему видеоадаптеру. Эти размеры основаны на базе размеров стандартного дисплея для данного видеоадаптера.

Значения HORZRES и VERTRES — это ширина и высота области отображения в пикселях. Для контекста устройства дисплея эти данные равны значениям, возвращаемым функцией *GetSystemMetrics*. Используя эти значения совместно с HORZSIZE и VERTSIZE, вы можете получить информацию о разрешении вашего устройства в пикселях на миллиметр. Если вы знаете, что в одном дюйме 25.4 миллиметра, то вы можете получить разрешение в точках на дюйм.

Величины ASPECTX, ASPECTY и ASPECTXY — это относительные ширина, высота и диагональный размер каждого пикселя, округленные до ближайшего целого. ASPECTXY равно корню квадратному из суммы квадратов ASPECTX и ASPECTY (по теореме Пифагора).

Величины LOGPIXELSX, LOGPIXELSY — это число пикселей в одном горизонтальном и вертикальном логическом дюйме. Для дисплея логический дюйм не равен физическому дюйму (25.4 мм), как вы могли легко убедиться, выполнив несложные расчеты с использованием значений HORZSIZE, VERTSIZE, HORZRES и VERTRES. Величины LOGPIXELSX и LOGPIXELSY требуют небольшого разъяснения. Вы могли не раз видеть, что текстовые процессоры, работающие под Windows, отображают линейку, которая не совсем правильна: Если вы измерите линейку на VGA мониторе, вы обнаружите, что на ней интервал, равный 1 дюйму, на самом деле чуть больше 1.5 дюйма. Текстовые процессоры используют значения LOGPIXELSX и LOGPIXELSY для отображения линейки. Если программа работает с реальными физическими единицами измерения, то обычный 10-точечный (10 point) или 12-точечный текст будет таким мелким, что его станет трудно читать. Логические единицы измерения позволяют адекватно представить текст на экране дисплея. Когда начнется работа с текстом, вновь придется решать эту проблему. Она касается только видеомониторов; для принтеров все единицы измерения, возвращаемые функцией *GetDeviceCaps* являются реальными.

О цветах

Цветные дисплеи требуют более одного бита для хранения информации о пикселе. Больше битов — больше цветов. Число уникальных цветов равно 2 в степени, равной числу битов. 16-цветные видеоадаптеры требуют 4 бита на пиксель. Эти биты организованы в цветовые плоскости — красная плоскость, зеленая плоскость, голубая плоскость и плоскость интенсивности. Адаптеры с 8, 16 или 24 битами на пиксель имеют одну цветовую плоскость, в которой набор смежных битов представляет цвет каждого пикселя.

Функция *GetDeviceCaps* дает вам возможность распознать организацию памяти видеоадаптера и число цветов, которые он может отобразить. Такой вызов данной функции возвращает число цветовых плоскостей:

```
iPlanes = GetDeviceCaps(hdc, PLANES);
```

Следующий вызов возвращает число битов цвета для представления одного пикселя:

```
iBitsPixel = GetDeviceCaps(hdc, BITSPIXEL);
```

Большинство графических устройств, способных отображать цвета, используют или множество цветных плоскостей или множество битов на пиксель, но не и то и другое сразу. Другими словами, один из указанных вызовов будет возвращать значение, равное 1. Число цветов, которые могут быть отображены видеоадаптером, вычисляется по формуле:

```
iColors = 1 <<(iPlanes * iBitsPixel);
```

Это значение не всегда совпадает с числом цветов, которое можно получить с помощью параметра NUMCOLORS:

```
iColors = GetDeviceCaps(hdc, NUMCOLORS);
```

Например, эти два значения будут различными для большинства плоттеров. Для плоттера значения и PLANES и BITSPIXEL будут равны 1, а величина NUMCOLORS будет зависеть от числа цветных перьев, которые имеются в плоттере. Для монохромных устройств функция *GetDeviceCaps* возвращает значение, равное 2, при вызове с параметром NUMCOLORS.

Более важно то, что эти два значения могут также отличаться для 256-цветных видеоадаптеров, поддерживающих загружаемые цветовые палитры. Функция *GetDeviceCaps* с параметром NUMCOLORS возвращает число цветов, зарезервированных Windows, которое равно 20. Оставшиеся 236 цветов могут быть заданы Windows-программой, использующей управление палитрами.

Windows использует беззнаковое 32-разрядное длинное целое для представления цвета. Тип данных для цвета называется COLORREF. Младшие три байта задают красную, зеленую и голубую составляющие, величина которых находится в интервале от 0 до 255, как показано на рис. 4.3. Таким образом, палитра может иметь 2^{24} (примерно 16 миллионов) цветов.

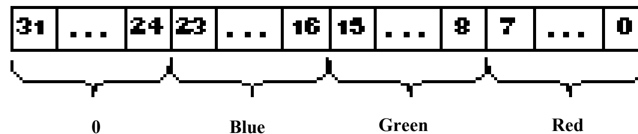


Рис. 4.3 32-разрядное представление цвета

Это беззнаковое длинное целое часто называют "RGB-цвет". Заголовочный файл Windows содержит несколько макросов для работы со значениями RGB. Макрос RGB получает три аргумента, представляющих красную, зеленую и голубую составляющие, и конструирует из них беззнаковое длинное целое:

```
#define RGB(r, g, b) ((COLORREF)(((BYTE)(r) | ((WORD)(g) << 8)) | ((DWORD)(BYTE)(b)) << 16)))
```

Таким образом, величина:

```
RGB(255, 0, 255)
```

равна 0x00FF00FF, значению RGB для пурпурного цвета (magenta). Когда все три аргумента равны 0, цвет — черный; когда все три равны 255 — белый. Макросы *GetRValue*, *GetGValue* и *GetBValue* извлекают беззнаковые символьные значения соответствующих цветов из значения RGB-цвета. Эти макросы иногда полезны, если вы используете функции Windows, возвращающие значение RGB в вашу программу.

Число цветов, возвращаемое функцией *GetDeviceCaps*, это число чистых цветов, которые может отобразить устройство. В дополнение к чистым цветам Windows может использовать полутона, представляющие собой пиксельный шаблон из пикселей разных цветов. Не все уникальные комбинации байтов красного, зеленого и голубого цветов формируют разные полутоновые шаблоны. Например, на 16-цветном VGA значения красного, зеленого, голубого должны быть возведены в 4-ю степень для получения различных полутонов. Таким образом, для этих адаптеров вы имеете 2^{18} или 262 144 полутона.

Вы можете определить ближайший чистый цвет для любого цвета, используя функцию *GetNearestColor*:

```
rgbPureColor = GetNearestColor(hdc, rgbColor);
```

Атрибуты контекста устройства

Как уже говорилось выше, Windows использует контекст устройства для хранения атрибутов, определяющих поведение функций GDI при выводе. Например, когда вы выводите текст, используя функцию *TextOut*, вам не надо задавать цвет текста или шрифт. Windows использует контекст устройства для получения этой информации.

Когда программа запрашивает описатель контекста устройства, Windows создает контекст устройства со значениями всех атрибутов по умолчанию. Атрибуты контекста устройства приведены в следующей таблице. Программа может изменить или получить любой из этих атрибутов.

Атрибут контекста устройства	Значение по умолчанию	Функции для изменения	Функции для получения
Режим отображения (Mapping mode)	MM_TEXT	*SetMapMode*	*GetMapMode*
Начало координат окна (Window origin)	(0,0)	*SetWindowOrgEx* *OffsetWindowOrgEx*	*GetWindowOrgEx*
Начало координат области вывода (Viewport origin)	(0,0)	*SetViewportOrgEx* *OffsetViewportOrgEx*	*GetViewportOrgEx*
Протяженность окна (Window extent)	(1,1)	*SetWindowExtEx* *SetMapMode* *ScaleWindowExtEx*	*GetWindowExtEx*
Протяженность области вывода (Viewport extent)	(1,1)	*SetViewportExtEx* *SetMapMode* *ScaleViewportExtEx*	*GetViewportExtEx*
Перо (Pen)	BLACK_PEN	*SelectObject*	*SelectObject*
Кисть (Brush)	WHITE_BRUSH	*SelectObject*	*SelectObject*
Шрифт (Font)	SYSTEM_FONT	*SelectObject*	*SelectObject*
Битовый образ (Bitmap)	Нет	*SelectObject*	*SelectObject*
Текущая позиция пера (Current pen position)	(0,0)	*MoveToEx* *LineTo* *PolylineTo* *PolyBezierTo*	*GetCurrent-PositionEx*
Режим фона (Background mode)	OPAQUE	*SetBkMode*	*GetBkMode*
Цвет фона (Background color)	Белый	*SetBkColor*	*GetBkColor*
Цвет текста (Text color)	Черный	*SetTextColor*	*GetTextColor*
Режим рисования (Drawing mode)	R2_COPYPEN	*SetROP2*	*GetROP2*
Режим растяжения (Stretching mode)	BLACKONWHITE	*SetStretchBltMode*	*GetStretchBlt-Mode*
Режим закрашивания многоугольников (Polygon filling mode)	ALTERNATE	*SetPolyFillMode*	*GetPolyFillMode*
Межсимвольный интервал (Intercharacter spacing)	0	*SetTextCharacter-Extra*	*GetTextCharac-terExtra*
Начало координат кисти (Brush origin)	(0,0) в экранных координатах	*SetBrushOrgEx*	*GetBrushOrgEx*
Область отсечения (Clipping region)	Нет	*SelectObject* *SelectClipRgn* *IntersectClipRgn* *OffsetClipRgn* *ExcludeClipRgn* *SelectClipPath*	*GetClipBox*

Сохранение контекста устройства

В этой главе вы столкнетесь с различными функциями, изменяющими атрибуты контекста устройства. Обычно Windows создает новый контекст устройства со значениями атрибутов по умолчанию, когда вы вызываете функции *GetDC* или *BeginPaint*. Все изменения атрибутов теряются, когда контекст устройства освобождается посредством вызова функций *ReleaseDC* или *EndPaint*. Если вашей программе необходимы значения атрибутов контекста устройства, отличные от значений по умолчанию, вам необходимо инициализировать контекст устройства каждый раз, когда вы получаете его описатель:

```
case WM_PAINT:
       hdc = BeginPaint(hwnd, &ps);
       [инициализация атрибутов контекста устройства]
       [рисование в клиентской области окна]
       EndPaint(hwnd, &ps);
       return 0;
```

Хотя этот подход вполне приемлем, вы можете предпочесть, чтобы изменения атрибутов контекста устройства, сделанные вами, сохранялись, когда вы освобождаете контекст устройства, и использовались в следующий раз при вызове *GetDC* или *BeginPaint*. Можно этого добиться, включив флаг CS_OWNDC при регистрации класса окна:

```
wndclass.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
```

Теперь каждое окно, которое вы создадите на базе этого класса окон, будет иметь свой собственный контекст устройства до тех пор, пока окно не будет уничтожено. Когда вы используете стиль CS_OWNDC, вам нужно только один раз проинициализировать атрибуты контекста устройства, скорее всего, при обработке сообщения WM_CREATE:

```
case WM_CREATE:
	hdc = GetDC(hwnd);
	[инициализация атрибутов контекста устройства]
	ReleaseDC(hwnd, hdc);
```

Атрибуты сохраняют значения до тех пор, пока вы их не измените.

Стиль CS_OWNDC влияет только на контексты устройств, полученные от функций *GetDC* и *BeginPaint*, и не влияет на полученные от других функций (таких как *GetWindowDC*). Использование стиля CS_OWNDC имеет цену: Windows требует примерно 800 байтов для хранения контекста устройства для каждого окна, созданного с помощью этого стиля. Даже если вы используете CS_OWNDC, вы должны освободить контекст устройства перед выходом из оконной процедуры.

Вы можете также использовать стиль CS_CLASSDC:

```
wndclass.style = CS_HREDRAW | CS_VREDRAW | CS_CLASSDC;
```

Использование этого стиля приводит к тому, что все окна такого базового класса разделяют один контекст устройства. Использование контекста устройства типа CS_CLASSDC сложнее, чем использование типа CS_OWNDC, потому что любые изменения атрибутов контекста устройства CS_CLASSDC, будут влиять на все окна, созданные на базе одного и того же базового класса. Это может привести к странным эффектам.

В некоторых случаях вам может потребоваться изменить некоторые атрибуты контекста устройства, нарисовать что-нибудь, используя измененные атрибуты, затем вернуться к оригинальному (предыдущему) состоянию контекста устройства. Для упрощения этого процесса вы сохраняете состояние контекста устройства, вызывая:

```
iSavedID = SaveDC(hdc);
```

Теперь вы изменяете атрибуты. Когда вы захотите вернуться к контексту устройства, существовавшему перед вызовом *SaveDC*, вы используете функцию:

```
RestoreDC(hdc, iSavedID);
```

Вы можете вызывать *SaveDC* любое число раз до вызова *RestoreDC*. Если вы хотите установить контекст устройства, существовавший перед последним вызовом функции *SaveDC*, вы вызываете:

```
RestoreDC(hdc, -1);
```

Рисование отрезков

Теоретически, все, что необходимо драйверу устройства для рисования, это функция *SetPixel* (и, в некоторых случаях, функция *GetPixel*). Все остальное можно осуществить с помощью высокоуровневых функций, реализуемых или модулем GDI или даже кодом вашей программы. Рисование отрезка, к примеру, просто требует неоднократных вызовов функции "рисование пикселя" с соответствующим изменением координат x и y.

Если вас не волнует время ожидания результата, вы можете выполнить почти любой рисунок с помощью только процедур рисования и чтения пикселя. Значительно более эффективным в графических системах является реализация функций рисования отрезков и других сложных графических операций на уровне драйвера устройства, который содержит код, оптимизированный для выполнения этих операций. По мере того, как технология видеоадаптеров становится все более изощренной, платы адаптеров будут содержать графические сопроцессоры, которые позволят рисовать объекты на аппаратном уровне.

Windows GDI, тем не менее, содержит функции *SetPixel* и *GetPixel*. Хотя в программе CONNECT из главы 7 используется функция *SetPixel*, в реальной жизни при программировании графики эти функции используются редко. Для большинства задач наиболее низкоуровневым векторным графическим примитивом является линия.

Windows способна отображать прямые линии (отрезки), эллиптические кривые и сплайны Безье. В Windows 95 поддерживаются семь функций для рисования линий. Это функции *LineTo* (отрезки прямых), *Polyline* и *PolylineTo* (ряды смежных отрезков прямой, ломаные), *PolyPolyline* (множественные ломаные), *Arc* (дуги эллипса), *PolyBezier* и *PolyBezierTo*. (Windows NT поддерживает еще три функции рисования линий — *ArcTo*, *AngleArc* и *PolyDraw*. Эти функции не поддерживаются в Windows 95.) Пять атрибутов контекста устройства влияют на представление линий, созданных с использованием этих функций: текущая позиция пера (только для функций *LineTo*, *PolylineTo* и *PolyBezierTo*), перо, режим фона (для несплошных перьев), цвет фона (для режима фона OPAQUE) и режим рисования.

По причинам, которые мы обсудим ниже, в данном разделе будут также рассмотрены функции *Rectangle, Ellipse, RoundRect, Chord* и *Pie*, хотя эти функции закрашивают замкнутую область и рисуют линии.

Функция *LineTo* — одна из немногих функций GDI, которые содержат не все размеры отображаемого объекта. Вместо этого *LineTo* рисует отрезок прямой из точки, называемой текущим положением пера и определенной в контексте устройства, до точки, заданной при вызове функции. Эта точка не включается в отрезок. Текущая позиция пера — это просто начальная точка для некоторых других функций GDI. В контексте устройства текущее положение пера по умолчанию устанавливается в точку (0,0). Если вы вызываете функцию *LineTo* без предварительной установки текущей позиции, она рисует отрезок, начинающийся в левом верхнем углу рабочей области окна.

Для рисования отрезка из точки с координатами (*xStart, yStart*) в точку с координатами (*xEnd*, *yEnd*) вы должны сначала использовать функцию *MoveToEx* для установки текущего положения пера в точку с координатами (*xStart, yStart*):

```
MoveToEx(hdc, xStart, yStart, &pt);
```

где *pt* — структура типа POINT, определенная в заголовочном файле Windows как:

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

MoveToEx ничего не рисует. Она просто изменяет текущее положение пера. Предыдущее текущее положение заносится в структуру POINT. Вы можете использовать *LineTo* для рисования отрезка:

```
LineTo(hdc, xEnd, yEnd);
```

Эта функция рисует отрезок до точки (*xEnd, yEnd*), не включая ее в отрезок. Для последующих вызовов *LineTo* текущее положение пера устанавливается в точку (*xEnd, yEnd*).

Краткое историческое замечание: в 16-битовых версиях Windows функция, изменяющая текущее положение пера, называлась *MoveTo* и имела три параметра — описатель контекста устройства и координаты по *x* и *y*. Функция возвращала предыдущее текущее положение пера, упакованное как два 16-разрядных значения в одном 32-разрядном беззнаковом длинном целом. Теперь, в 32-битных версиях Windows (включая Windows NT и Windows 95) координаты представлены 32-разрядными величинами. Поскольку 32-битные версии языка С не имеют 64-битного типа данных, потребовалось заменить функцию *MoveTo* на *MoveToEx*. Это изменение необходимо еще и потому, что возвращаемое из *MoveTo* значение почти никогда не использовалось в программировании реальных задач.

Теперь хорошие новости: если вам не нужно предыдущее текущее положение пера — что часто встречается на практике — вы можете просто установить в NULL последний параметр функции *MoveToEx*. Фактически, чтобы преобразовать ваш 16-битный код для Windows 95, вы можете определить такой макрос:

```
#define MoveTo(hdc, x, y) MoveToEx(hdc, x, y, NULL)
```

Этот макрос будет использоваться во многих программах в следующих главах данной книги.

А сейчас плохие новости: несмотря на то, что координаты в Windows 95 описываются 32-битными значениями, используются только младшие 16 бит. Значения координат возможны только в интервале от —32 768 до 32 767.

Вы можете узнать текущее положение пера посредством вызова:

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

Следующий фрагмент программы рисует сетку в рабочей области окна с интервалом в 100 пикселей, начиная от левого верхнего угла. Переменная *hwnd* представляет собой описатель окна, *hdc* — описатель контекста устройства, а *x* и *y* — целые:

```
GetClientRect(hwnd, &rect);
for (x = 0; x < rect.right; x += 100)
{
       MoveToEx(hdc, x, 0, NULL);
       LineTo (hdc, x, rect.bottom);
}
for (y = 0; y < rect.bottom; y += 100)
{
       MoveToEx(hdc, 0, y, NULL);
       LineTo (hdc, rect.right, y);
}
```

Хотя может показаться странным, что для рисования одного отрезка надо использовать две функции, атрибут текущего положения пригодится, когда вы захотите нарисовать ряд связанных отрезков. Например, вы можете определить массив из 5 точек (10 значений), описывающих контур прямоугольника:

```
POINT pt [5] = { 100, 100, 200, 100, 200, 200, 100, 200, 100, 100 };
```

Обратите внимание, что последняя и первая точки совпадают. Теперь вам нужно только вызвать *MoveToEx* для первой точки и *LineTo* для остальных:

```
MoveToEx(hdc, pt[0].x, pt[0].y, NULL);

for (i = 1; i < 5; i++)
        LineTo(hdc, pt[i].x, pt[i].y);
```

Поскольку функция *LineTo* рисует из текущей точки до конечной (не включая ее), ни одна точка не будет прорисована дважды. Повторное рисование точки на экране не создает никаких проблем, оно может плохо выглядеть на плоттере или в некоторых других режимах рисования, которые будут рассмотрены ниже.

Когда имеется массив точек, которые надо соединить отрезками, можно нарисовать их более простым способом, используя функцию *Polyline*. Этот оператор рисует такой же прямоугольник, как и код, приведенный выше:

```
Polyline(hdc, pt, 5);
```

Последний параметр — число точек. Мы могли бы также представить это число как *(sizeof(pt) / sizeof(*POINT*))*. Результат применения *Polyline* такой же, как и при использовании начального вызова функции *MoveToEx* с последующими многократными вызовами функции *LineTo*. Тем не менее, *Polyline* не учитывает и не изменяет текущее положение пера. *PolylineTo* немного отличается. Эта функция использует текущее положение для начальной точки и устанавливает текущее положение в конец последнего нарисованного отрезка. Приведенный ниже код рисует тот же прямоугольник, как и в предыдущих примерах:

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

Хотя вы можете использовать функции *Polyline* и *PolylineTo* для рисования нескольких отрезков или ломаных, эти функции более применимы при рисовании сложных кривых, состоящих из сотен и тысяч отрезков. Например, предположим вы хотите изобразить синусоидальную волну. Программа SINEWAVE, приведенная на рис. 4.4 иллюстрирует то, как это делается.

SINEWAVE.MAK

```
#-----------------------
# SINEWAVE.MAK make file
#-----------------------

sinewave.exe : sinewave.obj
     $(LINKER) $(GUIFLAGS) -OUT:sinewave.exe sinewave.obj $(GUILIBS)

sinewave.obj : sinewave.c
     $(CC) $(CFLAGS) sinewave.c
```

SINEWAVE.C

```
/*----------------------------------------
   SINEWAVE.C -- Sine Wave Using Polyline
                 (c) Charles Petzold, 1996
  ----------------------------------------*/

#include <windows.h>
#include <math.h>

#define NUM    1000
#define TWOPI (2 * 3.14159)

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "SineWave";
     HWND        hwnd;
     MSG         msg;
```

```
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Sine Wave Using Polyline",
                     WS_OVERLAPPEDWINDOW,
                     CW_USEDEFAULT, CW_USEDEFAULT,
                     CW_USEDEFAULT, CW_USEDEFAULT,
                     NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int  cxClient, cyClient;
     HDC         hdc;
     int         i;
     PAINTSTRUCT ps;
     POINT       pt [NUM];

     switch(iMsg)
          {
          case WM_SIZE:
               cxClient = LOWORD(lParam);
               cyClient = HIWORD(lParam);
               return 0;

          case WM_PAINT:
               hdc = BeginPaint(hwnd, &ps);

               MoveToEx(hdc, 0,        cyClient / 2, NULL);
               LineTo  (hdc, cxClient, cyClient / 2);

               for(i = 0; i < NUM; i++)
                    {
                    pt[i].x = i * cxClient / NUM;
                    pt[i].y =(int)(cyClient / 2 *
                                   (1 - sin(TWOPI * i / NUM)));
                    }

               Polyline(hdc, pt, NUM);
```

```
            return 0;

        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
        }

    return DefWindowProc(hwnd, iMsg, wParam, lParam);
    }
```

Рис. 4.4 Программа SINEWAVE

Эта программа содержит массив из 1000 структур POINT. В цикле от 0 до 999 член *x* структуры растет от 0 до *cxClient*. В каждом цикле член *y* структуры определяет значение синуса и масштабируется до размеров клиентской области окна. Вся кривая целиком отображается с использованием одного вызова функции *Polyline*. Поскольку функция *Polyline* реализована на уровне драйвера устройства, это работает значительно быстрее, чем 1000-кратные вызовы функции *LineTo*. Результаты работы программы приведены на рис. 4.5.

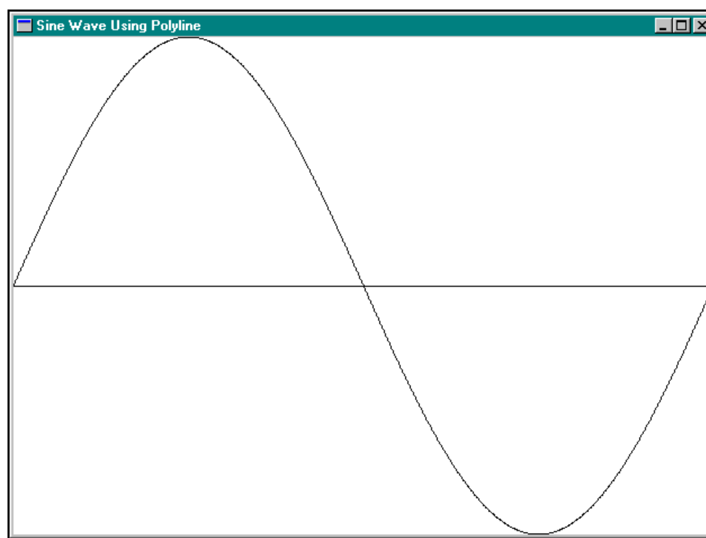


Рис. 4.5 Вывод программы SINEWAVE

Ограничивающий прямоугольник

Теперь рассмотрим функцию *Arc*, которая рисует эллиптическую кривую. Рассмотрение функции *Arc* не имеет смысла без предварительного рассмотрения функции *Ellipse*, рассмотрение функции *Ellipse* не имеет смысла без рассмотрения функции *Rectangle*. Если рассматривать функции *Ellipse* и *Rectangle*, то следует разобраться также с функциями *RoundRect*, *Chord* и *Pie*.

Проблема в том, что функции *Rectangle, Ellipse, RoundRect, Chord* и *Pie* предназначены не только для рисования линий. Да, эти функции рисуют линии, но они также закрашивают ограниченную этими линиями область, используя текущую кисть. По умолчанию эта кисть сплошная и белая, поэтому, начиная экспериментировать с этими функциями, вы могли не заметить, что они делают еще что-то, кроме рисования линий. Строго говоря, эти функции относятся к следующему разделу "Рисование закрашенных областей", но тем не менее рассмотрим их здесь.

Все функции, которые были указаны выше, схожи в том, что все они строятся с использованием "ограничивающего прямоугольника" (bounding box). Вы определяете координаты прямоугольника, ограничивающего объект, и Windows рисует объект, используя это прямоугольник.

Простейшей из этих функций является функция рисования прямоугольника:

```
Rectangle(hdc, xLeft, yTop, xRight, yBottom);
```

Точка с координатами (*xLeft, yTop*) — это левый верхний угол прямоугольника, а точка (*xRight, yBottom*) — правый нижний угол. Фигура, нарисованная с использованием функции *Rectangle*, приведена на рис. 4.6. Стороны прямоугольника всегда параллельны горизонтальной и вертикальной сторонам экрана.

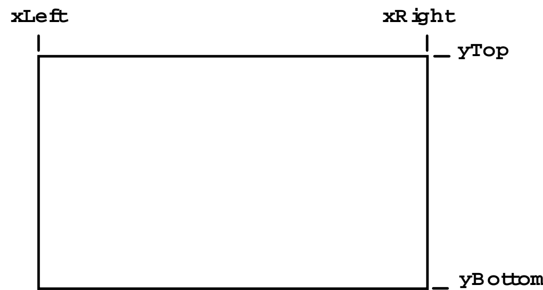


Рис. 4.6 Фигура, нарисованная с использованием функции *Rectangle*

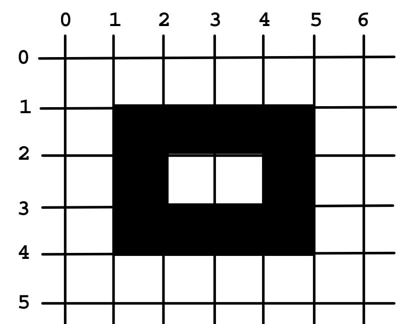
Программисты, ранее работавшие с графикой, привыкли к проблеме одного пикселя. Некоторые графические системы изображают фигуры, включающие правую и нижнюю координаты, а некоторые — рисуют фигуры до правой и нижней координаты, не включая их.

Windows использует последний подход. Есть простой путь для иллюстрации сказанного.

Предположим, вызывается функция:

```
Rectangle(hdc, 1, 1, 5, 4);
```

Как уже говорилось, Windows использует для рисования ограничивающий прямоугольник. Вы можете представить себе дисплей как сетку, каждая ячейка которой представляет один пиксель.



Воображаемый ограничивающий прямоугольник рисуется по сетке, а выводимый прямоугольник рисуется с использованием ограничивающий. Это выглядит так:

Ширина полос, отделяющих прямоугольник от верхней и левой границ рабочей области окна равна 1 пикселю. Windows использует текущую кисть для зарисовки 2-х пикселей внутри прямоугольника.

Вы уже знаете, как нарисовать прямоугольник, теперь надо выяснить, как нарисовать эллипс, поскольку для этого необходимы те же параметры:

```
Ellipse(hdc, xLeft, yTop, xRight, yBottom);
```

Фигура, отображаемая функцией *Ellipse* (вместе с ограничивающим прямоугольником) приведена на рис. 4.7.

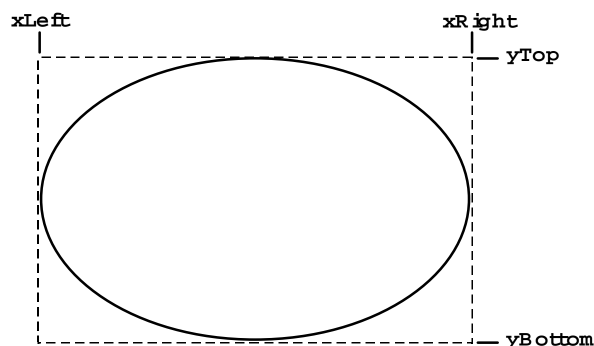


Рис. 4.7 Фигура, нарисованная с использованием функции *Ellipse*

Функция для рисования прямоугольника с скругленными углами применяет тот же ограничивающий прямоугольник, что и функции *Rectangle* и *Ellipse*, но с двумя дополнительными параметрами:

```
RoundRect(hdc, xLeft, yTop, xRight, yBottom, xCornerEllipse, yCornerEllipse);
```

Фигура, отображаемая этой функцией приведена на рис. 4.8.

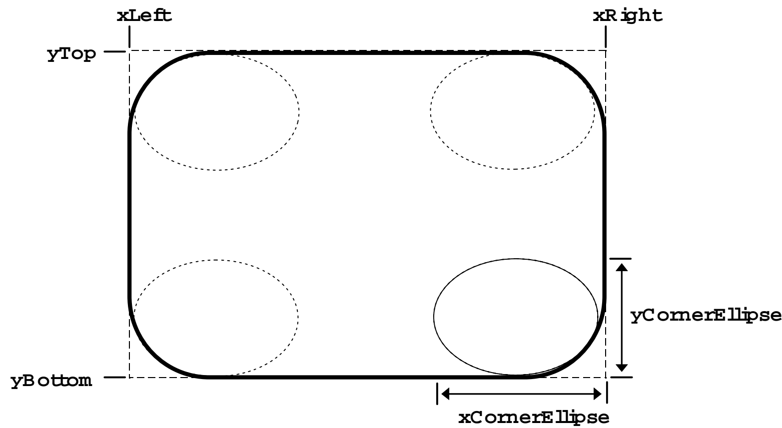


Рис. 4.8 Фигура, нарисованная с использованием функции *RoundRect*

Windows использует маленький эллипс для рисования скругленных углов. Ширина этого эллипса равна *xCornerEllipse*, а высота равна *yCornerEllipse*. Представьте себе, что Windows делит этот маленький эллипс на четыре квадранта по одному на каждый из четырех углов. Округлость углов более заметна при больших значениях *xCornerEllipse* и *yCornerEllipse*. Если значение *xCornerEllipse* равно разности между *xLeft* и *xRight*, а *yCornerEllipse* — разности между *yTop* и *yBottom*, то функция *RoundRect* будет отображать эллипс.

Скругленные углы, показанные на рис. 4.8, были нарисованы с использованием размеров углового эллипса, вычисленных по формулам:

```
xCornerEllipse =(xRight — xLeft) / 4;
yCornerEllipse =(yBottom — yTop) / 4;
```

Это простое приближение, но результаты, скорее всего, будут выглядеть не совсем правильно, потому что округлость углов более заметна при больших размерах прямоугольника. Для решения этой проблемы, вы, вероятно, захотите сделать равными реальные размеры *xCornerEllipse* и *yCornerEllipse*.

В функции *Arc*, *Chord* и *Pie* передаются одинаковые параметры:

```
Arc(hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd);
Chord(hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd);
Pie(hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd);
```

Линия, рисуемая функцией *Arc* приведена на рис. 4.9; фигуры, отображаемые функциями *Chord* и *Pie*, приведены на рис. 4.10 и 4.11. Windows использует воображаемую линию для соединения точки (*xStart, yStart*) с центром эллипса. В точке, где эта линия пересекается с ограничивающим прямоугольником, Windows начинает рисовать дугу эллипса в направлении против часовой стрелки. Windows также использует воображаемую линию для соединения точки (*xEnd, yEnd*) с центром эллипса. В точке, где эта линия пересекается с ограничивающим прямоугольником, Windows завершает рисование дуги.

В случае функции *Arc* действия Windows на этом заканчиваются, поскольку дуга — эллиптическая кривая, не ограничивающая замкнутую область. В случае функции *Chord* Windows соединяет конечные точки дуги. В случае функции *Pie* Windows соединяет начальную и конечную точки дуги с центром эллипса. Внутренняя область фигур, образуемых функциями *Chord* и *Pie*, закрашивается текущей кистью.

Вас может удивить использование начальной и конечной позиций в функциях *Arc*, *Chord* и *Pie*. Почему бы просто не указать начальную и конечную точки на кривой эллипса? Хорошо, вы можете сделать так, но вам придется численно описать, что это за точки. Метод, применяемый в Windows, работает, не требуя этих уточнений.

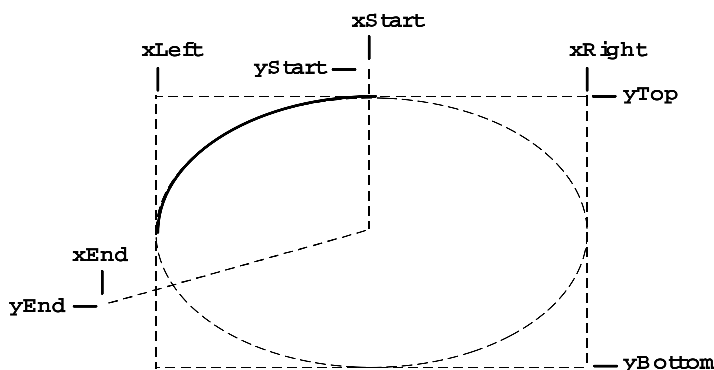


Рис. 4.9 Фигура, нарисованная с использованием функции *Arc*

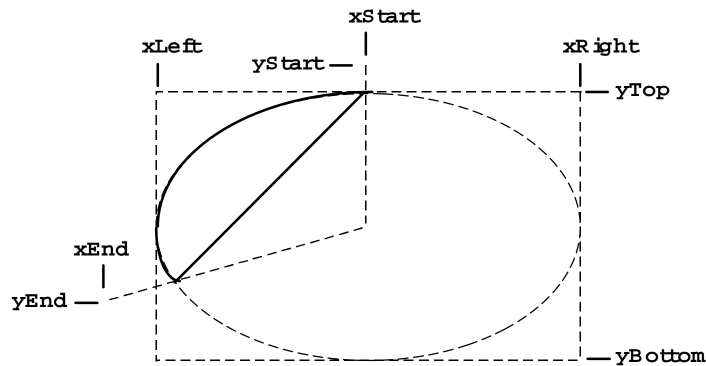


Рис. 4.10 Фигура, нарисованная с использованием функции *Chord*

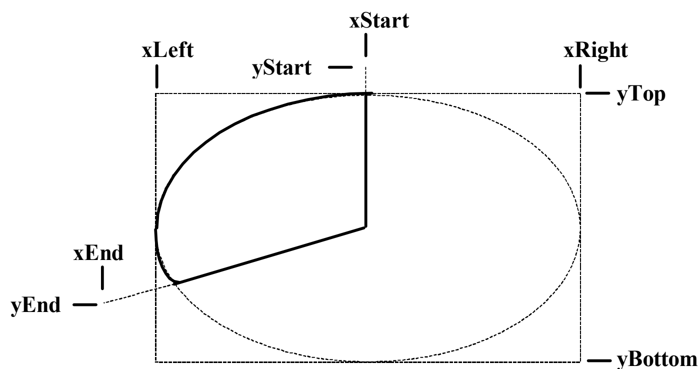


Рис. 4.11 Фигура, нарисованная с использованием функции *Pie*

Программа LINEDEMO, приведенная на рис. 4.12, рисует прямоугольник, эллипс, прямоугольник с скругленными углами и два отрезка, но в другом порядке. Эта программа показывает, что функции, определяющие области, закрашивают их. Поэтому отрезки не видны там, где нарисован эллипс. Результаты вывода программы приведены на рис. 4.13.

LINEDEMO.MAK

```
#------------------------
# LINEDEMO.MAK make file
#------------------------

linedemo.exe : linedemo.obj
     $(LINKER) $(GUIFLAGS) -OUT:linedemo.exe linedemo.obj $(GUILIBS)

linedemo.obj : linedemo.c
     $(CC) $(CFLAGS) linedemo.c
```

LINEDEMO.C

```
/*--------------------------------------------------
   LINEDEMO.C -- Line-Drawing Demonstration Program
                 (c) Charles Petzold, 1996
  --------------------------------------------------*/

#include <windows.h>

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "LineDemo";
```

```
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);
     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Line Demonstration",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int  cxClient, cyClient;
     HDC         hdc;
     PAINTSTRUCT ps;

     switch(iMsg)
          {
          case WM_SIZE:
               cxClient = LOWORD(lParam);
               cyClient = HIWORD(lParam);
               return 0;

          case WM_PAINT:
               hdc = BeginPaint(hwnd, &ps);

               Rectangle(hdc,     cxClient / 8,     cyClient / 8,
                                7 * cxClient / 8, 7 * cyClient / 8);

               MoveToEx (hdc,        0,        0, NULL);
               LineTo   (hdc, cxClient, cyClient);

               MoveToEx (hdc,        0, cyClient, NULL);
               LineTo   (hdc, cxClient,        0);

               Ellipse  (hdc,     cxClient / 8,     cyClient / 8,
                                7 * cxClient / 8, 7 * cyClient / 8);

               RoundRect(hdc,     cxClient / 4,     cyClient / 4,
```

```
                         3 * cxClient / 4, 3 * cyClient / 4,
                             cxClient / 4,     cyClient / 4);
               EndPaint(hwnd, &ps);
               return 0;

          case WM_DESTROY:
               PostQuitMessage(0);
               return 0;
          }

     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

Рис. 4.12 Программа LINEDEMO

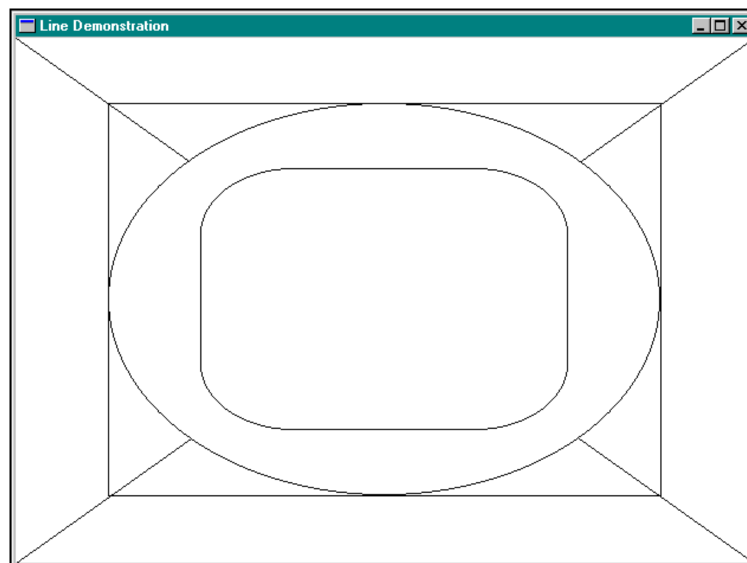


Рис. 4.13 Окно программы LINEDEMO

Сплайны Безье

Слово "сплайн" раньше относилось к гибкому куску дерева, резины или металла, который использовали для рисования кривых на листе бумаги. Например, если вы имели несколько отстоящих друг от друга точек, и вы хотели соединить их с помощью кривой (для интерполяции или экстраполяции), вы должны были, во-первых, отметить эти точки на чертежной бумаге, затем "привязать" сплайн к точкам и карандашом нарисовать кривую по сплайну так, как он был изогнут вокруг точек. (Пожалуйста, не смейтесь. Кажется, что так могло быть только в 19 веке, но хорошо известно, что механические сплайны использовались страховыми служащими, рассчитывавшими вероятность страхового случая, еще 15 лет назад.)

В наше время, конечно, сплайны — это математические выражения. Они имеют самые разные применения. Сплайны Безье — одни из самых популярных в программировании компьютерной графики. Это совсем недавнее усовершенствование в арсенале графических средств, доступных на уровне операционной системы, и оно пришло с неожиданной стороны. В шестидесятых годах автомобильная компания Renault переходила от ручного проектирования кузовов автомобилей (что требовало много глины) к компьютерному. Требовался математический аппарат, и Пьер Безье предложил набор формул, оказавшихся очень полезными в этой работе.

С тех пор двумерная форма сплайна Безье показала себя как самая удобная кривая (после прямых линий и эллипсов) в компьютерной графике. Например, в языке PostScript сплайны Безье используются для всех кривых — эллиптические линии аппроксимируются из сплайнов Безье. Кривые Безье также используются для описания контуров символов различных шрифтов языка PostScript. (TrueType используют более простые и быстрые формы сплайнов.)

Простой двумерный сплайн Безье определяется четырьмя точками — двумя конечными и двумя контрольными. Концы кривой привязаны к двум конечным точкам. Контрольные точки выступают в роли магнитов для оттягивания кривой от прямой, соединяющей две крайние точки. Это лучше всего иллюстрируется интерактивной программой BEZIER, приведенной на рис. 4.14.

BEZIER.MAK

```
#----------------------
# BEZIER.MAK make file
#----------------------

bezier.exe : bezier.obj
    $(LINKER) $(GUIFLAGS) -OUT:bezier.exe bezier.obj $(GUILIBS)

bezier.obj : bezier.c
    $(CC) $(CFLAGS) bezier.c
```

BEZIER.C

```
/*---------------------------------------
   BEZIER.C -- Bezier Splines Demo
               (c) Charles Petzold, 1996
  ---------------------------------------*/

#include <windows.h>

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Bezier";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Bezier Splines",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

void DrawBezier(HDC hdc, POINT apt[])
     {
```

```
     PolyBezier(hdc, apt, 4);

     MoveToEx(hdc, apt[0].x, apt[0].y, NULL);
     LineTo  (hdc, apt[1].x, apt[1].y);

     MoveToEx(hdc, apt[2].x, apt[2].y, NULL);
     LineTo  (hdc, apt[3].x, apt[3].y);
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static POINT apt[4];
     HDC          hdc;
     int          cxClient, cyClient;
     PAINTSTRUCT  ps;

     switch(iMsg)
          {
          case WM_SIZE:
               cxClient = LOWORD(lParam);
               cyClient = HIWORD(lParam);

               apt[0].x = cxClient / 4;
               apt[0].y = cyClient / 2;

               apt[1].x = cxClient / 2;
               apt[1].y = cyClient / 4;

               apt[2].x =     cxClient / 2;
               apt[2].y = 3 * cyClient / 4;

               apt[3].x = 3 * cxClient / 4;
               apt[3].y =     cyClient / 2;

               return 0;

          case WM_MOUSEMOVE:
               if(wParam & MK_LBUTTON || wParam & MK_RBUTTON)
                    {
                    hdc = GetDC(hwnd);

                    SelectObject(hdc, GetStockObject(WHITE_PEN));
                    DrawBezier(hdc, apt);

                    if(wParam & MK_LBUTTON)
                         {
                         apt[1].x = LOWORD(lParam);
                         apt[1].y = HIWORD(lParam);
                         }

                    if(wParam & MK_RBUTTON)
                         {
                         apt[2].x = LOWORD(lParam);
                         apt[2].y = HIWORD(lParam);
                         }

                    SelectObject(hdc, GetStockObject(BLACK_PEN));
                    DrawBezier(hdc, apt);
                    ReleaseDC(hwnd, hdc);
                    }

               return 0;
          case WM_PAINT:
               InvalidateRect(hwnd, NULL, TRUE);
```

```
            hdc = BeginPaint(hwnd, &ps);

            DrawBezier(hdc, apt);

            EndPaint(hwnd, &ps);
            return 0;

        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
        }

    return DefWindowProc(hwnd, iMsg, wParam, lParam);
    }
```

Рис. 4.14 Программа BEZIER

Поскольку эта программа использует логику обработки мыши (об этом пойдет речь в главе 6), здесь не будут объясняться принципы ее функционирования (это было бы преждевременным). Вместо этого, поэкспериментируйте со сплайнами Безье. В этой программе две крайние точки установлены на половине высоты и на 1/4 и 3/4 ширины рабочей зоны окна. Двумя контрольными точками можно манипулировать: первой — нажатием левой кнопки мыши и перемещением мыши, второй — нажатием правой кнопки мыши и перемещением. На рис. 4.15 показан типовой вид окна программы.

Кроме самого сплайна Безье программа также отображает слева прямую линию из первой контрольной точки в первую крайнюю точку (или начальную точку) и прямую линию из второй контрольной точки в конечную точку справа.

Сплайны Безье считаются полезными для компьютерного проектирования благодаря следующим характеристикам:

Во-первых, немного попрактиковавшись, вы можете легко манипулировать кривой для получения нужной формы.

Во-вторых, сплайны Безье очень легко управляются. В некоторых формах сплайнов кривая не может быть проведена через все определяющие точки. Сплайны Безье всегда "привязаны" к двум конечным точкам. (Это первое допущение, которое берет начало в формулах Безье.) Кроме того, существуют сплайны с бесконечными кривыми, которые имеют свои особенности. В компьютерном проектировании редко встречаются подобные типы сплайнов. Как правило, кривые Безье всегда ограничены четырехэлементной ломаной, называемой "выпуклым корпусом" (convex hull), которая получается соединением конечных и контрольных точек.

В-третьих, в сплайнах Безье существует связь между конечными и контрольными точками. Кривая всегда является касательной к прямой, соединяющей начальную точку и первую контрольную точку, и направленной в ту же сторону. (Это иллюстрируется программой BEZIER.) Кривая также является касательной к прямой, соединяющей конечную точку и вторую контрольную точку, и направленной в ту же сторону. Это еще два допущения на основе формул Безье.

В-четвертых, сплайны Безье в основном хорошо смотрятся. Понятно, что это критерий субъективный, но так считают многие.

До появления Windows 95 сплайны Безье создавались с помощью функции *Polyline*. Вам следовало также знать параметрические уравнения, описывающие сплайны Безье. Начальная точка ($x0,y0$), конечная точка ($x3, y3$). Две контрольные точки ($x1, y1$) и ($x2, y2$). Кривая, отображаемая в интервале t от 0 до 1 описывалась так:

$$x(t) = (1-t)^3x0 + 3t(1-t)^2x1 + 3t^2(1-t)x2 + t^3x3$$

$$y(t) = (1-t)^3y0 + 3t(1-t)^2y1 + 3t^2(1-t)y2 + t^3y3$$

В Windows 95 эти формулы знать не нужно. Для того, чтобы нарисовать одну или более связанных сплайнов Безье, используйте:

```
PolyBezier(hdc, pt, iCount);
```

или

```
PolyBezierTo(hdc, pt, iCount);
```

В обоих случаях *pt* — массив структур типа POINT. В функции *PolyBezier* первые четыре точки идут в таком порядке: начальная точка, первая контрольная точка, вторая контрольная точка, конечная точка кривой Безье. Каждая следующая кривая Безье требует три новых точки, поскольку начальная точка следующей кривой есть

конечная точка предыдущей и т. д. Параметр *iCount* всегда равен единице плюс умноженному на три числу связанных кривых, которые вы хотите отобразить.

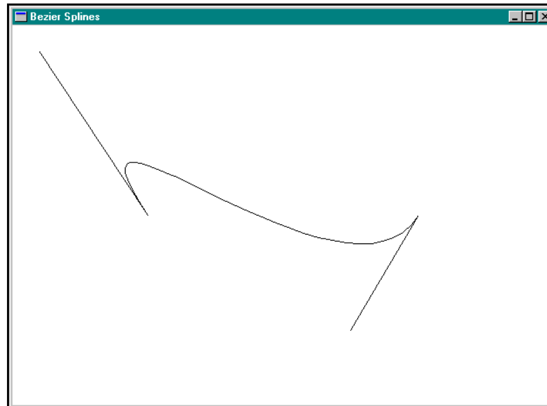


Рис. 4.15 Окно программы BEZIER

Функция *PoliBezierTo* использует текущее положение пера как начальную точку. Первый и все последующие сплайны требуют только три точки. Когда функция возвращает управление, текущая позиция пера устанавливается в конечную точку последней кривой.

Одно предупреждение: Когда вы рисуете набор связанных сплайнов Безье, в точке связи будет плавный переход, только если вторая контрольная точка первой кривой Безье, конечная точка первой кривой (она же начальная точка второй кривой) и первая контрольная точка второй кривой лежат на одной прямой.

Использование стандартных перьев

Когда вы вызываете одну из функций рисования кривых, которые только что рассматривались, Windows для рисования линии использует перо, выбранное в контексте устройства в данный момент. Перо определяет цвет линии, ширину и ее стиль, который может быть сплошным (solid), точечным (dotted) или пунктирным (dashed). Перо, устанавливаемое в контексте устройства по умолчанию называется BLACK_PEN (черное перо, черный карандаш). Это перо рисует сплошные черные линии толщиной в один пиксель независимо от режима отображения (mapping mode). BLACK_PEN — это одно из трех "стандартных" перьев, поддерживаемых Windows. Два других — это WHITE_PEN (белое перо) и NULL_PEN (пустое перо). NULL_PEN это перо, которое ничего не рисует. Вы можете также создавать свои собственные перья.

В программах для Windows обычно для ссылки на перо используется описатель (handle). Заголовочный файл Windows содержит определение типа HPEN, описатель пера (handle to a pen). Вы можете определить переменную, например, *hPen*, используя такое определение:

```
HPEN hPen;
```

Вы получаете описатель одного из стандартных перьев, вызывая функцию *GetStockObject*. Например, предположим, вы хотите использовать стандартное перо WHITE_PEN. Его описатель можно получить так:

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

Теперь вы должны сделать это перо выбранным в контексте устройства текущим пером. Для этого необходимо вызвать функцию *SelectObject*:

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

После этого вызова все линии, которые вы рисуете, будут использовать WHITE_PEN до тех пор, пока вы не выберете другое перо в контекст устройства или пока не освободите контекст устройства.

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

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

Если затем вы захотите вернуться к использованию пера BLACK_PEN, вы можете получить описатель этого стандартного пера и выбрать его в контекст устройства в одной инструкции:

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

SelectObject возвращает описатель того пера, которое уже было выбрано в контексте устройства. Если вы начинаете работать с только что полученным описателем контекста устройства и вызываете:

```
hPen = SelectObject(hdc, GetStockObject(WHITE_PEN));
```

то текущим выбранным пером в контексте устройства становится WHITE_PEN, а переменная *hPen* становится описателем пера BLACK_PEN. Вы можете выбрать BLACK_PEN в контекст устройства, используя вызов:

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

Создание, выбор и удаление перьев

Хотя перья, определенные как стандартные объекты, несомненно, удобны, вы ограничены использованием только сплошного черного пера, сплошного белого пера или пустого пера. Если вы хотите большего, то вы должны создавать свои собственные перья. Здесь приведена последовательность действий: вы создаете "логическое перо", которое только описывает перо, используя функции *CreatePen* или *CreatePenIndirect*. (Вы можете также использовать функцию *ExtCreatePen*, которая будет обсуждаться далее в этой главе.) Эти функции возвращают описатель логического пера. Вы выбираете перо в контекст устройства путем вызова *SelectObject*. Затем вы можете рисовать линии, используя это новое перо. Только одно перо может быть одновременно выбрано в контексте устройства. После того, как вы освободите контекст устройства (или выберете в контекст устройства другое перо), вы можете удалить созданное вами перо, используя *DeleteObject*. После того, как вы это сделаете, значение описателя пера становится недействительным.

Логическое перо — объект GDI. Вы создаете и используете перо, но оно не принадлежит вашей программе. В действительности оно принадлежит модулю GDI. Перо — это один из шести объектов GDI, которые вы можете создавать. Другие пять — это кисти, битовые образы, регионы, шрифты и палитры. За исключением палитр все эти объекты выбираются в контекст устройства, используя функцию *SelectObject*.

Три правила управляют использованием таких объектов GDI как перья:

- Обязательно удаляйте все созданные вами объекты GDI.
- Не удаляйте объекты GDI, пока они выбраны в действительном контексте устройства.
- Не удаляйте стандартные объекты.

Это разумные правила, но они иногда могут немного подвести. Рассмотрим на примерах, как действуют эти правила.

Вызов функции *CreatePen* выглядит так:

```
hPen = CreatePen(iPenStyle, iWidth, rgbColor);
```

Параметр *iPenStyle* определяет, какого типа линии будут отображаться: сплошная, точечная или пунктирная. Этот параметр может принимать одно значение из следующего списка идентификаторов, приведенных в заголовочном файле Windows. На рис. 4.16 показаны линии каждого типа.

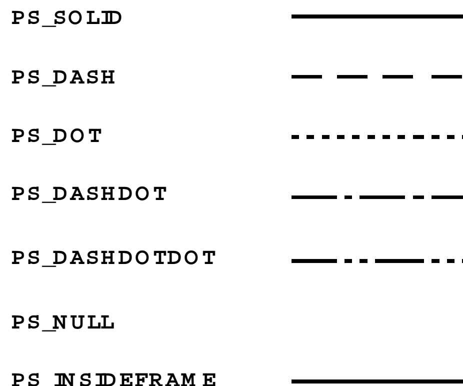


Рис. 4.16 Семь стилей пера

Для стилей PS_SOLID, PS_NULL и PS_INSIDEFRAME параметр *iWidth* — ширина пера. Ширина *iWidth* равная 0, заставляет Windows использовать перо шириной в один пиксель. Стандартные перья имеют ширину в 1 пиксель. Если вы зададите точечный или пунктирный стиль линии с физической шириной больше 1, Windows, тем не менее, будут использовать сплошное перо.

Параметр *rgbColor* функции *CreatePen* — это беззнаковое длинное целое, задающее цвет пера. Для перьев всех стилей, кроме PS_INSIDEFRAME, когда вы выбираете перо в контекст устройства, Windows преобразует значение этого параметра в ближайший чистый цвет, какой может представить устройство. Только перья со стилем PS_INSIDEFRAME могут использовать полутона, и только если их ширина больше 1.

Стиль PS_INSIDEFRAME, когда используется с функциями, определяющими закрашенные области, имеет еще одну особенность. Для всех стилей, кроме PS_INSIDEFRAME, если перо используется для рисования контура шириной более 1 пикселя, то оно центрируется таким образом, что часть линии может оказаться за пределами ограничивающего прямоугольника. Для стиля PS_INSIDEFRAME вся линия целиком рисуется внутри ограничивающего прямоугольника.

Вы можете также создать перо, определив структуру типа LOGPEN — "логическое перо" (logical pen) и вызвав функцию *CreatePenIndirect*. Если ваша программа использует несколько различных перьев, которые вы инициализируете в своей программе, этот метод наиболее эффективен. В начале вы определяете переменную типа LOGPEN, например, *logpen*:

```
LOGPEN logpen;
```

Эта структура имеет три члена: *lopnStyle* (UINT) — стиль пера, *lopnWidth* (POINT) — ширина пера в логических единицах измерения, *lopnColor* (COLORREF) — цвет пера. Член структуры *lopnWidth* имеет тип POINT, но Windows использует только величину *lopnWidth.x* как ширину пера и игнорирует значение *lopnWidth.y*. Затем вы создаете перо, передавая адрес структуры в функцию *CreatePenIndirect*:

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

Вы можете также получить информацию логического пера для уже существующего пера. Если у вас есть описатель пера, вы можете скопировать данные, определяющие логическое перо в структуру типа LOGPEN, используя вызов *GetObject*:

```
GetObject(hPen, sizeof(LOGPEN),(LPVOID) &logpen);
```

Обратите внимание, что функции *CreatePen* и *CreatePenIndirect* не требуют описателя контекста устройства. Эти функции создают логические перья, которые никак не связаны с контекстом устройства до тех пор, пока вы не вызовите *SelectObject*. Например, вы можете использовать одно логическое перо для нескольких различных устройств, таких как дисплей и принтер.

Ниже представлен метод создания, выбора и удаления перьев. Предположим, ваша программа использует три пера — черное шириной 1, красное шириной 3 и черное точечное. Вы можете сначала определить переменные для хранения описателей этих перьев:

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

В процессе обработки сообщения WM_CREATE вы можете создать три пера:

```
hPen1 = CreatePen(PS_SOLID, 1, 0);
hPen2 = CreatePen(PS_SOLID, 3, RGB(255, 0, 0));
hPen3 = CreatePen(PS_DOT,  0, 0);
```

В процессе обработки сообщения WM_PAINT (или в любой момент, когда у вас есть действительный контекст устройства) вы можете выбрать одно из этих перьев в контекст устройства и рисовать, используя его:

```
SelectObject(hdc, hPen2);
[функции рисования линий]
SelectObject(hdc, hPen1);
[другие функции рисования линий]
```

В процессе обработки сообщения WM_DESTROY вы можете удалить три пера, созданные вами ранее:

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

Это наиболее общий, магистральный метод создания, выбора и удаления перьев, но он требует резервирования памяти для логических перьев на все время работы вашей программы. Вместо этого вы можете создать перья в процессе обработки сообщения WM_PAINT и удалить их после вызова *EndPaint*. (Вы можете удалить их и до вызова *EndPaint*, но вы должны быть осторожны и не удалить перо, выбранное в контекст устройства.)

Кроме того вы можете создать перья на "лету" и объединить вызовы функций *CreatePen* и *SelectObject* в одну инструкцию:

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

Теперь, когда вы рисуете линии, вы будете использовать красное точечное перо. Когда вы закончите рисовать красным точечным пером, вы можете удалить его. Но, как можно удалить это перо, если не сохранен его

описатель? Новый вызов *SelectObject* возвращает описатель пера, которое было раньше выбрано в контексте устройства. Таким образом, вы можете удалить перо путем выбора стандартного пера BLACK_PEN в контекст устройства и удаления значения, возвращаемого функцией *SelectObject*:

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

Рассмотрим другой метод. Когда вы выбираете только что созданное перо в контекст устройства, сохраните описатель, возвращаемый функцией *SelectObject*:

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

Что такое *hPen*? Если это первый вызов *SelectObject* после получения описателя контекста устройства, то *hPen* — это описатель стандартного пера BLACK_PEN. Вы можете теперь выбрать это перо в контекст устройства и удалить перо, созданное вами (описатель, возвращаемый вторым вызовом функции *SelectObject*) в одной инструкции:

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

Закрашивание пустот

Использование точечных и штриховых перьев ставит интересный вопрос: что будет с пустотами между точками и штрихами? Цвет этих пробелов или пустот зависит от режима фона (background mode) и атрибутов цвета фона, определенных в контексте устройства. Режим фона по умолчанию равен OPAQUE, т. е. Windows заполняет пустоты цветом фона, который, по умолчанию, белый. Это согласуется с работой стандартной кисти WHITE_BRUSH, которую многие программы используют в классе окна для стирания фона окна.

Вы можете изменить цвет фона, который Windows будет использовать для закрашивания пустот, вызвав:

```
SetBkColor(hdc, rgbColor);
```

Также как и для перьев, Windows преобразует этот цвет фона к чистому цвету. Вы можете определить текущий цвет фона, выбранный в контексте устройства, вызвав функцию *GetBkColor*.

Вы можете также отменить заполнение пустот системой Windows, изменив режим фона на TRANSPARENT:

```
SetBkMode(hdc, TRANSPARENT);
```

Windows будет игнорировать цвет фона и не будет заполнять пустоты. Вы можете определить текущий режим фона (как TRANSPARENT, так и OPAQUE), путем вызова *GetBkMode*.

Режимы рисования

Представление линий, отображаемых на дисплее, зависит также от режима рисования (drawing mode), установленного в контексте устройства. Представление цветной линии основывается не только на цвете пера, но и на цвете той области дисплея, где эта линия отображается. Подумайте о возможности использовать одно и то же перо для рисования черной линии на белом фоне и белой линии на черном фоне без знаний о том, какого цвета фон. Было бы это для вас удобным? Вы можете все проверить, применяя различные режимы рисования.

Когда Windows использует перо для рисования линии, на самом деле осуществляется поразрядная логическая операция между пикселями пера и пикселями принимающей поверхности устройства. Выполняемая поразрядная логическая операция над пикселями носит название "растровой операции" (raster operation или ROP). Поскольку рисование линий требует только двух пиксельных шаблонов (пера и приемной поверхности), логическая операция называется "бинарной растровой операцией" (binary raster operation или ROP2). Windows определяет 16 ROP2 кодов, показывающих, как Windows оперирует с пикселями пера и приемника. В контексте устройства по умолчанию режим рисования определяется как R2_COPYPEN, что означает простое копирование системой Windows пикселей пера в приемник и привычным при работе с перьями. Существует также еще 15 других ROP2 кодов.

Откуда взялись эти 16 различных ROP2 кодов? Для того, чтобы показать зачем они введены, рассмотрим монохромную систему. Цвет приемника (цвет рабочей области окна) может быть либо черным (0) либо белым (1). Перо также может быть либо белым, либо черным. Существует четыре комбинации использования черного или белого пера на черном или белом приемнике: белое перо на белом приемнике, белое перо на черном приемнике, черное перо на белом приемнике и черное перо на черном приемнике.

Что произойдет с приемником после рисования? Один вариант — это линия всегда будет черной, независимо от цвета пера или приемника: Это режим рисования, имеющий один из ROP2 кодов, а именно код R2_BLACK. Другой вариант — это линия будет черной, кроме комбинации, когда и перо и приемник — черные. В этом случае линия будет белой. Хотя это может показаться странным, в Windows есть соответствующий режим рисования, который называется R2_NOTMERGEPEN. Windows выполняет поразрядную операцию OR над пикселями пера и приемника, а затем инвертирует результат.

В приведенной ниже таблице показаны 16 ROP2 режимов рисования. В таблице показано, как цвет пера (P) комбинируется с цветом приемника (D) для получения результирующего цвета. Столбец с заголовком "Булева операция" (Boolean Operation) использует нотацию языка С, показывающую, как комбинируются пиксели пера и приемника.

Перо (Pen (P)): **Приемник (Destination (D)):**	**1** **1**	**1** **0**	**0** **1**	**0** **0**	**Булева операция (Boolean operation)**	**Режим рисования (Drawing mode)**
Результаты: (Results)	0	0	0	0	0	R2_BLACK
	0	0	0	1	~(P | D)	R2_NOTMERGEPEN
	0	0	1	0	~P & D	R2_MASKNOTPEN
	0	0	1	1	~P	R2_NOTCOPYPEN
	0	1	0	0	P & ~D	R2_MASKPENNOT
	0	1	0	1	~D	R2_NOT
	0	1	1	0	P ^ D	R2_XORPEN
	0	1	1	1	~(P & D)	R2_NOTMASKPEN
	1	0	0	0	P & D	R2_MASKPEN
	1	0	0	1	~(P ^ D)	R2_NOTXORPEN
	1	0	1	0	D	R2_NOP
	1	0	1	1	~P | D	R2_MERGENOTPEN
	1	1	0	0	P	R2_COPYPEN (по умолчанию)
	1	1	0	1	P | ~D	R2_MERGEPENNOT
	1	1	1	0	P | D	R2_MERGEPEN
	1	1	1	1	1	R2_WHITE

Вы можете установить новый режим рисования в контексте устройства:

```
SetROP2(hdc, iDrawMode);
```

Параметр *iDrawMode* должен быть равен одному из значений, приведенных в столбце "Режим рисования" таблицы. Вы можете определить текущий режим рисования, используя функцию:

```
iDrawMode = GetROP2(hdc);
```

По умолчанию в контексте устройства режим рисования устанавливается в R2_COPYPEN, что означает простой перенос цвета пера в приемник. В режиме R2_NOTCOPYPEN рисование ведется белым цветом, если перо черное, и черным — если перо белое. В режиме R2_BLACK рисование ведется всегда черным цветом независимо от цвета пера или фона. Аналогично, в режиме R2_WHITE рисование ведется всегда белым цветом. Режим R2_NOP означает "нет операции": в этом режиме приемник остается неизменным.

Мы начали с рассмотрения примера работы на чисто монохромной системе. В действительности же, на монохромном дисплее Windows может отображать оттенки серого путем смешения черных и белых пикселей. При рисовании пером на таком полутоновом фоне Windows просто осуществляет поразрядные операции по принципу пиксель с пикселем. В режиме R2_NOT рисование ведется цветом, обратным цвету приемника, независимо от цвета пера. Этот режим используется тогда, когда вы не знаете цвет фона, потому что он гарантирует, что пиксели всегда будут видимы. (Ну, почти гарантирует — если фон на 50% серый, то перо будет практически невидимым.) В программе BLOKOUT в главе 6 будет продемонстрировано использование R2_NOT.

Рисование закрашенных областей

Давайте сделаем следующий шаг вперед от рисования линий к рисованию фигур. В Windows имеется семь функций для рисования закрашенных фигур, имеющих границу. Эти функции приведены в таблице:

Функция	**Фигура**
Rectangle	Прямоугольник
Ellipse	Эллипс
RoundRect	Прямоугольник со скругленными углами
Chord	Дуга кривой эллипса, концы которой соединены хордой
Pie	Кусок, вырезанный из эллипса
Polygon	Многоугольник
PolyPolygon	Множество многоугольников

Windows рисует контур фигуры, используя текущее перо, выбранное в контексте устройства. Текущий режим фона, цвет фона и режим рисования — все используются при рисовании этого контура, как будто Windows рисует линию. Все, что мы уже изучили про линии, применимо и к рамкам, ограничивающим эти фигуры.

Фигура закрашивается текущей кистью, выбранной в контексте устройства. По умолчанию, это стандартная кисть WHITE_BRUSH. Следовательно, внутренняя область фигуры будет закрашена белым цветом. Windows имеет шесть стандартных (stock) кистей: WHITE_BRUSH (белая кисть), LTGRAY_BRUSH (светло-серая кисть), GRAY_BRUSH (серая кисть), DKGRAY_BRUSH (темно-серая кисть), BLACK_BRUSH (черная кисть) и NULL_BRUSH или HOLLOW_BRUSH (пустая кисть).

Вы можете выбрать одну из стандартных кистей в контекст устройства точно также, как и стандартное перо. Windows определяет HBRUSH как описатель кисти, поэтому вам следует сначала определить переменную типа описателя кисти:

```
HBRUSH hBrush;
```

Вы можете получить описатель кисти GRAY_BRUSH, вызвав *GetStockObject*:

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

Вы можете выбрать эту кисть в контекст устройства, вызвав *SelectObject*:

```
SelectObject(hdc, hBrush);
```

Теперь, когда вы рисуете одну из указанных фигур, их внутренняя область закрашивается серым.

Если вы хотите нарисовать фигуру без рамки, выберите перо NULL_PEN в контекст устройства:

```
SelectObject(hdc, GetStockObject(NULL_PEN));
```

Если вы хотите нарисовать только контур фигуры без закрашивания внутренней области, выберите кисть NULL_BRUSH в контекст устройства:

```
SelectObject(hdc, GetStockObject(NULL_BRUSH));
```

Вы можете также создать собственные кисти аналогично тому, как вы можете создать собственные перья. Скоро мы рассмотрим эту тему.

Функция *Polygon* и режим закрашивания многоугольника

Первые пять функций рисования многоугольников мы уже рассмотрели. Функция *Polygon* — шестая из функций рисования ограниченных и закрашенных фигур. Вызов этой функции очень похож на вызов функции *Polyline*:

```
Polygon(hdc, pt, iCount);
```

Параметр *pt* — это массив структур типа POINT, *iCount* — число точек. Если последняя точка в массиве не совпадает с первой, то Windows добавляет линию, соединяющую последнюю и первую точки. (Функция *Polyline* этого не делает.)

Windows закрашивает внутреннюю область фигуры текущей кистью, учитывая, какой из режимов закрашивания многоугольников установлен текущим в контексте устройства. По умолчанию режим закрашивания равен ALTERNATE (попеременный), означающий, что Windows закрашивает только те фрагменты внутренней области многоугольника, которые получаются путем соединения линий с нечетными номерами (1, 3, 5 и т. д.). Другие фрагменты внутренней области не закрашиваются. Вы можете установить режим закрашивания WINDING (сквозной), в котором Windows закрашивает все внутренние области. Вы устанавливаете режим закрашивания так:

```
SetPolyFillMode(hdc, iMode);
```

Для иллюстрации режимов закрашивания рассмотрим пример с пятиконечной звездой. На рис. 4.17 звезда, находящаяся слева, нарисована в режиме ALTERNATE, а звезда, находящаяся справа — в режиме WINDING.

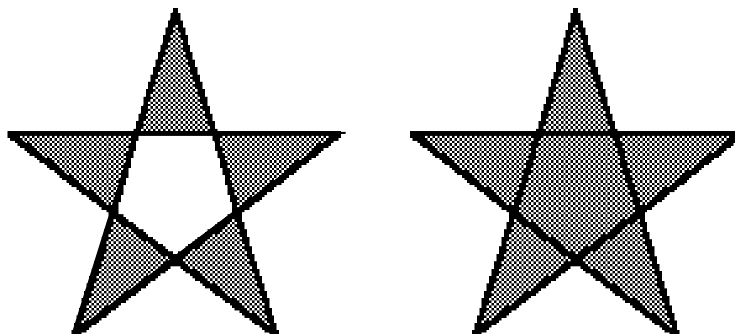


Рис. 4.17 Фигуры, нарисованные в двух режимах закрашивания многоугольника: ALTERNATE (слева) и WINDING (справа)

Закрашивание внутренней области

Внутренняя область фигур, соответствующих функциям *Rectangle, RoundRect, Ellipse, Chord, Pie, Polygon* и *PolyPolygon* закрашивается текущей кистью (или шаблоном "pattern"), выбранной в контексте устройства. Кисть — это 8×8 битовый образ, который размножается в горизонтальном и вертикальном направлении при закрашивании области.

Когда Windows использует полутона для отображения большего числа цветов, чем доступно на дисплее, она использует кисть. На монохромных системах Windows может использовать полутона, состоящие из черных и белых пикселей, для создания 64 разных оттенков серого. Более точно, Windows может создать 64 различных монохромных кисти. Для чистого черного цвета все биты в 8x8 растровом образе равны 0. Для получения первого оттенка серого один из 64 битов устанавливается в 1 (т. е. делается белым); два бита для получения второго оттенка серого и т. д. до тех пор, пока все биты не будут установлены в 1 для чисто белого цвета. На цветных видеосистемах полутона — это тоже битовые образы, но с гораздо более широким набором доступных цветов.

Windows содержит четыре функции, позволяющие вам создавать логические кисти. Выбор кисти в контекст устройства осуществляется функцией *SelectObject*. Так же как и логические перья, логические кисти — тоже объекты GDI. Любая кисть, созданная вами, должна быть удалена. Но нельзя удалять кисть до тех пор, пока она выбрана в контексте устройства.

Ниже приведена функция для создания логической кисти:

```
hBrush = CreateSolidBrush(rgbColor);
```

Слово "solid" в имени функции означает, что создается кисть, имеющая чистый цвет. Когда вы выбираете кисть в контекст устройства, Windows создает 8×8 битовый образ для полутонов и использует его для кисти.

Вы можете также создать штриховую кисть (hatch), состоящую из горизонтальных, вертикальных или диагональных линий. Кисти этого типа используются в основном при закрашивании внутренней области столбиковых диаграмм и при выводе на плоттер. Ниже приведена функция для создания штриховой кисти:

```
hBrush = CreateHatchBrush(iHatchStyle, rgbColor);
```

Параметр *iHatchStyle* задает стиль штриховки. Он может принимать одно из следующих значений: HS_HORIZONTAL, HS_VERTICAL, HS_FDIAGONAL, HS_BDIAGONAL, HS_CROSS и HS_DIAGCROSS. На рис. 4.18 показан фрагмент штриховки для каждого из указанных стилей.

Рис. 4.18 Шесть стилей штриховки для кисти

Параметр *rgbColor* функции *CreateHatchBrush* задает цвет штриховых линий. Когда вы выбираете кисть в контекст устройства, Windows преобразует этот цвет к ближайшему чистому цвету. Промежутки между штриховыми линиями закрашиваются в соответствии с режимом фона и цветом фона, определенными в контексте устройства. Если режим фона равен OPAQUE, то цвет фона, который преобразуется к ближайшему чистому цвету, используется для закрашивания промежутков между штриховыми линиями. В этом случае ни штриховые линии, ни цвет фона не могут быть полутонами. Если режим фона равен TRANSPARENT, то Windows рисует штриховые линии и не зарисовывает промежутки между ними.

В связи с тем, что кисти — это всегда битовые матрицы 8×8, внешний вид штриховых кистей сильно зависит от разрешения устройства, на котором они отображаются. Каждый из образцов, приведенных на рис. 4.18, изображен в прямоугольной области размером 32×16 пикселей, т. е. битовый образ размером 8×8 был повторен четыре раза по горизонтали и два раза по вертикали. На лазерном принтере с разрешением 300 точек на дюйм такие же 32x16 пиксельные прямоугольники будут иметь размер 1/9 дюйма в ширину и 1/19 дюйма в высоту.

Вы можете также создавать свои собственные кисти, основанные на битовых шаблонах, используя функцию *CreatePatternBrush*:

```
hBrush = CreatePatternBrush(hBitmap);
```

Эта функция подробнее будет рассмотрена в следующей главе при изучении битовых шаблонов.

Windows также содержит функцию, включающую в себя три других функции, строящих кисти (*CreateSolidBrush, CreateHatchBrush, CreatePatternBrush*):

```
hBrush = CreateBrushIndirect(&logbrush);
```

Переменная *logbrush* имеет тип структуры LOGBRUSH "логическая кисть" (logical brush). Ниже приведены три поля этой структуры. Значение поля *lbStyle* определяет, как Windows будет интерпретировать два других поля:

lbStyle (UINT)	**lbColor (COLORREF)**	**lbHatch (LONG)**
BS_SOLID	Цвет кисти	Игнорируется
BS_HOLLOW	Игнорируется	Игнорируется
BS_HATCHED	Цвет штриховых линий	Стиль штриховки
BS_PATTERN	Игнорируется	Описатель битового шаблона

Раньше мы использовали функцию *SelectObject* для выбора логического пера в контекст устройства, функцию *DeleteObject* — для удаления логического пера, и функцию *GetObject* — для получения информации о логическом пере. Вы можете использовать эти же три функции применительно к кистям. Получив описатель логической кисти, вы можете выбрать ее в контекст устройства, используя *SelectObject*:

```
SelectObject(hdc, hBrush);
```

Позднее вы удалите созданную кисть с помощью функции *DeleteObject*:

```
DeleteObject(hBrush);
```

Но никогда не удаляйте кисть, установленную текущей в контексте устройства. Если вам нужна информация о кисти, вы можете вызвать *GetObject*:

```
GetObject(hBrush, sizeof(LOGBRUSH),(LPVOID) &logbrush);
```

В этом вызове *logbrush* — это структура типа LOGBRUSH.

Режим отображения

До сих пор предполагалось, что мы рисуем в пиксельных координатах с началом координат в левом верхнем углу рабочей области окна. Да, это так по умолчанию, но это не единственная возможность.

Один из атрибутов контекста устройства, который фактически влияет на все, что вы рисуете в рабочей области, это режим отображения (mapping mode). Четыре других атрибута контекста устройства — начало координат окна (window origin), начало координат области вывода (viewport origin), протяженность окна (window extents) и протяженность области вывода (viewport extents) — полностью зависят от значения атрибута режима отображения.

Большинство рисующих функций GDI требуют в качестве параметров координаты или размеры. Например, функция *TextOut*:

```
TextOut(hdc, x, y, szBuffer, iLength);
```

Параметры *x* и *y* задают начальную позицию текста. Параметр *x* — это горизонтальная позиция, а параметр *y* — вертикальная позиция. Часто запись (*x, y*) используется для указания этой точки.

В функции *TextOut*, как фактически во всех функциях GDI, эти значения координат задаются в "логических единицах измерения". Windows должна преобразовать логические единицы в "физические единицы, единицы измерения устройства", т. е. пиксели. Результат этого преобразования определяется режимом отображения, началом координат окна и области вывода, растяжением окна и области вывода. Режим отображения задает также направление осей координат *x* и *y*, т. е. определяет, в каком направлении на экране возрастает значение координаты *x* — влево или вправо и в каком направлении возрастает значение координаты *y* — вверх или вниз.

В Windows определены восемь режимов отображения. Они приведены в таблице с использованием идентификаторов, определенных в заголовочных файлах Windows:

Режим отображения	**Логические единицы**	**Направление увеличения**	
		ось x	**ось y**
MM_TEXT	Пиксели	вправо	вниз
MM_LOMETRIC	0.1 мм	вправо	вверх
MM_HIMETRIC	0.01 мм	вправо	вверх
MM_LOENGLISH	0.01 дюйма	вправо	вверх
MM_HIENGLISH	0.001 дюйма	вправо	вверх
MM_TWIPS*	1/1440 дюйма	вправо	вверх
MM_ISOTROPIC	Произвольные (x=y)	выбирается	выбирается

* Слово "twip" производное от "twentieth of a point" — одна двадцатая часть точки. Точка, как единица измерения, равна примерно 1/72 дюйма, однако часто в графических системах таких как GDI, считается точно равной 1/72 дюйма. *Twip* — это 1/20 точки и, следовательно, 1/1440 дюйма.

Режим отображения	Логические единицы	Направление увеличения	
		ось x	ось y
MM_ANISOTROPIC	Произвольные (x!=y)	выбирается	выбирается

Вы можете устанавливать режим отображения, используя функцию:

```
SetMapMode(hdc, iMapMode);
```

Параметр *iMapMode* — это один из восьми идентификаторов режима отображения. Определить текущий режим отображения вы можете путем вызова:

```
iMapMode = GetMapMode(hdc);
```

По умолчанию установлен режим MM_TEXT. В этом режиме отображения логические единицы эквивалентны физическим единицам, что позволяет нам (или заставляет нас) работать непосредственно в терминах пикселей. В следующем вызове функции *TextOut*:

```
TextOut(hdc, 8, 16, szBuffer, iLength);
```

текст начинается в точке, отстоящей на 8 пикселей слева и на 16 пикселей сверху от границы рабочей области.

Если установлен режим отображения MM_LOENGLISH, то логические единицы — это сотые доли дюйма:

```
SetMapMode(hdc, MM_LOENGLISH);
```

Теперь вызов функции *TextOut* может выглядеть так:

```
TextOut(hdc, 50, -100, szBuffer, iLength);
```

Текст начинается на расстоянии 0,5 дюйма от левого края и на расстоянии 1-го дюйма от верхнего края рабочей области. (Почему у координаты *y* стоит знак минус, станет понятно позже, когда режимы отображения будут рассматриваться более подробно.) Другие режимы отображения позволяют программе задавать координаты вывода в миллиметрах, размерах точки принтера или произвольных единицах.

Если вы чувствуете себя уверенно при работе с пикселями, то вам не нужно использовать другие, отличные от MM_TEXT, режимы отображения. Если же вы хотите рисовать образы в реальных единицах измерения, таких как миллиметры или дюймы, то вы можете получить нужную вам информацию из функции *GetDeviceCaps* и осуществлять свое собственное масштабирование. Другие режимы отображения просто освобождают вас от этой рутинной работы.

Независимо от режима отображения все координаты, которые вы задаете в функциях Windows, должны быть знаковыми короткими целыми (signed short integer) числами в интервале от —32768 до 32767. Некоторые функции Windows, использующие начальную и конечную точки прямоугольника, также требуют, чтобы ширина и высота прямоугольника были меньше чем 32767.

Координаты устройства (физические координаты) и логические координаты

Вы можете спросить: если использовать режим отображения MM_LOENGLISH, можно ли получать сообщения WM_SIZE в терминах сотых долей дюйма? Конечно, нет. Windows продолжает использовать координаты устройства для всех сообщений (таких как WM_MOVE, WM_SIZE и WM_MOUSEMOVE), для всех функций, не принадлежащих GDI, и даже для некоторых функций GDI. Посмотрите на это с такой точки зрения: режим отображения — это атрибут контекста устройства, поэтому он начинает работать только тогда, когда вы используете функции GDI, требующие передачи им описателя контекста устройства. *GetSystemMetrics* — не является функцией GDI, следовательно, она будет продолжать возвращать размеры в координатах устройства, т. е. в пикселях. И хотя функция *GetDeviceCaps* — функция GDI и требует описателя контекста устройства, Windows продолжает возвращать единицы измерения устройства для индексов HORZRES и VERTRES, поскольку одна из задач этой функции — дать программе сведения о размерах устройства в пикселях.

Тем не менее, величины в структуре TEXTMETRIC, которую можно получить из функции *GetTextMetrics*, задаются в логических координатах. В режиме отображения MM_LOENGLISH функция *GetTextMetrics* возвращает информацию о ширине и высоте символов в сотых долях дюйма. Когда вы вызываете функцию *GetTextMetrics* для получения информации о ширине и высоте символов, режим отображения должен быть установлен таким, каким он будет, когда вы вызовите функцию рисования, использующую эти размеры. Поскольку в этой главе рассматриваются различные функции GDI, ваше внимание будет обращаться на то, какие координаты они используют — логические или физические. Все функции, которые мы рассматривали до сих пор, используют логические координаты, за исключением тех, которые определяют закрашивание пустот между точками и штрихами в линиях, и между штрихами в штриховых шаблонах. Работа этих функций не зависит от режима отображения.

Системы координат устройства

Windows преобразует логические координаты, заданные в функциях GDI, в координаты устройства. Перед тем, как мы рассмотрим логические системы координат, используемые в различных режимах отображения, давайте разберемся с системой координат устройства, которую Windows определяет для экрана дисплея. Хотя мы работали в основном с рабочей областью окна, Windows использует еще два других пространства координат устройства в разные моменты времени. Во всех системах координат устройства единицами всегда считаются пиксели. Значения горизонтальной координаты *x* возрастает слева направо, значения вертикальной координаты *y* — сверху вниз.

Когда мы работаем с экраном целиком, мы работаем в терминах "экранных координат." Левый верхний угол экрана — точка (0, 0). Экранные координаты используются в сообщениях WM_MOVE (для окон верхнего уровня, не дочерних) и в следующих функциях Windows: *CreateWindow* и *MoveWindow* (обе для не дочерних окон), *GetMessagePos*, *GetCursorPos, SetCursirPos, GetWindowRect, WindowFromPoint* и *SetBrushOrgEx*. Эти функции, в основном, не имеют окна, ассоциированного с ними (например, две функции работы с курсором), или функции, которые должны переместить или найти окно на основе положения некоторой точки экрана. Если вы используете функцию *CreateDC* с параметром DISPLAY для получения контекста устройства всего экрана, то логические координаты, указанные в вызовах функций GDI, будут преобразованы в координаты устройства.

"Полные координаты окна" (whole-window coordinates) определяют окно программы целиком, включая заголовок, меню, полосы прокрутки и рамку. Для обычного окна точка (0, 0) — левый верхний угол рамки окна. Полные координаты окна редко используются в Windows, но если вы получите контекст устройства, используя функцию *GetWindowDC*, то логические координаты в вызовах функций GDI будут преобразовываться в координаты всего окна.

Третья система координат устройства — это та, с которой мы работали больше всего, — система координат рабочей области окна. Точка (0, 0) — верхний левый угол рабочей области окна. Когда вы получаете контекст устройства, используя функции *GetDC* или *BeginPaint*, логические координаты в вызовах функций GDI преобразуются в координаты рабочей области окна.

Вы можете конвертировать координаты рабочей области окна в координаты экрана и наоборот, используя функции *ClientToScreen* и *ScreenToClient*. Вы можете также получить местоположение и размеры окна целиком в экранных координатах, используя функцию *GetWindowRect*. Эти три функции предоставляют достаточно возможностей для любых преобразований координат.

Область вывода и окно

Режим отображения определяет, как Windows преобразует логические координаты, заданные в параметрах функций GDI, в координаты устройства, конкретная система координат которого зависит от того, какой функцией вы получили контекст устройства. Для дальнейшего рассмотрения режимов отображения нам необходимо определить некоторые дополнительные термины: Говорят, что режим отображения определяет преобразование "окна" (window) — логические координаты, в "область вывода" (viewport) — координаты устройства.

Использование слов "окно" и "область вывода" не совсем удачно. В других языках графического интерфейса "область вывода" часто определяется как "область отсечения" (clipping region). Мы использовали термин "окно", имея в виду область экрана, захваченную программой. Мы должны оставить в стороне наше предвзятое мнение об этих терминах на время обсуждения.

Область вывода описывается в терминах координат устройства (пикселях). Чаще всего область вывода — это то же самое, что и рабочая область, хотя область вывода может описываться также и в полных координатах окна или в координатах экрана, если вы получили контекст устройства из функций *GetWindowDC* или *CreateDC*. Точка (0, 0) — левый верхний угол рабочей области (или окна целиком, или всего экрана). Значения координаты *x* возрастают слева направо, а значения координаты *y* — сверху вниз.

"Окно" описывается в терминах логических координат. Ими могут быть пиксели, миллиметры, дюймы или любые другие единицы, какие вы захотите. В вызовах функций GDI вы задаете логические координаты.

Для всех режимов отображения Windows преобразует оконные (логические) координаты в координаты области вывода (координаты устройства), используя следующие формулы:

$$xViewport = (xWindow - xWinOrg) \times (xViewExt/xWinExt) + xViewOrg$$

$$yViewport = (yWindow - yWinOrg) \times (yViewExt/yWinExt) + yViewOrg,$$

где (xWindow, yWindow) — логическая точка для преобразования, (xViewport, yViewport) — преобразованная точка в координатах устройства. Если координаты устройства — это координаты рабочей области или окна целиком, то Windows должна также преобразовать их в координаты экрана перед выводом объекта.

Эти формулы используют две точки, задающие начала координат (origin) окна и области вывода: (*xWinOrg, yWinOrg*) — начало координат окна в логических координатах; (*xViewOrg, yViewOrg*) — начало координат области

вывода в координатах устройства. В контексте устройства, установленном по умолчанию, обе эти точки установлены в точку (0, 0), но они могут быть изменены. Эти формулы гарантируют, что точка с логическими координатами (*xViewOrg, yViewOrg*) всегда преобразуется в точку с физическими координатами (*xViewOrg, yViewOrg*).

Эти формулы используют также две точки, определяющие "протяженность" (extent): (*xWinExt, yWinExt*) — протяженность окна в логических координатах; (*xViewExt, yViewExt*) — протяженность области вывода в координатах устройства. В большинстве режимов отображения протяженности определяются самими режимами и не могут быть изменены. Каждая протяженность сама по себе ничего не значит. Только отношение протяженности области вывода к протяженности окна является коэффициентом масштабирования при пересчете логических координат в координаты устройства. Протяженность может быть отрицательной. Это означает, что величина логической координаты *x* не обязательно должна возрастать при перемещении вправо, а величина логической координаты *y* — необязательно возрастать при движении вниз.

Windows может также преобразовывать координаты устройства (физические) в координаты окна (логические):

$$xWindow = (xViewport - xViewOrg) \times (xWinExt/xViewExt) + xWinOrg$$

$$yWindow = (yViewport - yViewOrg) \times (yWinExt/yViewExt) + yWinOrg$$

Windows имеет также две функции, которые позволяют вам в программе преобразовывать координаты устройства в логические координаты и логические координаты в координаты устройства. Следующая функция преобразует точки устройства в логические точки:

```
DPtoLP(hdc, pPoints, iNumber);
```

Переменная *pPoints* — это указатель на массив структур типа POINT, *iNumber* — число преобразуемых точек. Вы обнаружите, что эта функция очень полезна для преобразования размера рабочей области, полученного от функции *GetClientRect* (которая всегда оперирует с координатами устройства), в логические координаты:

```
GetClientRect(hwnd, &rect);

DPtoLP(hdc,(PPOINT) &rect, 2);
```

Следующая функция преобразует логические точки в физические точки:

```
LPtoDP(hdc, pPoints, iNumber);
```

Работа в режиме MM_TEXT

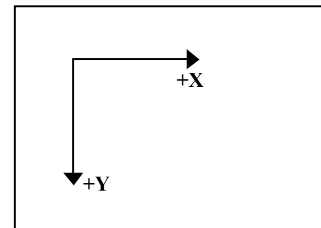
В режиме MM_TEXT заданы следующие величины начал координат и протяженностей:

Начало координат окна: (Window origin)	(0, 0)	Может быть изменено
Начало координат области вывода: (Viewport origin)	(0, 0)	Может быть изменено
Протяженность окна: (Window extent)	(1, 1)	Не может быть изменена
Протяженность области вывода: (Viewport extent)	(1, 1)	Не может быть изменена

Отношение протяженности области вывода к протяженности окна равно 1, таким образом, масштабирование между логическими и физическими координатами не производится. Формулы, приведенные выше, принимают вид:

$$xViewport = xWindow - xWinOrg + xViewOrg$$

$$yViewport = yWindow - yWinOrg + yViewOrg$$



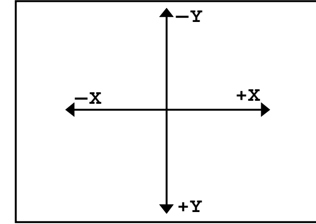
Этот режим отображения называется "текстовым" не потому, что он наиболее удобен для вывода текста, а из-за направления осей координат. Мы читаем текст слева направо и сверху вниз, и MM_TEXT аналогично задает направления увеличения координат:

Windows имеет функции *SerViewportOrgEx* и *SetWindowOrgEx* для изменения начала координат области вывода и окна. Эти функции обладают эффектом сдвига осей координат таким образом, что логическая точка (0, 0) не будет далее соответствовать левому верхнему углу рабочей области. Чаще всего вы будете использовать *SetViewportOrgEx* или *SetWindowOrgEx*, но не обе функции одновременно.

Вот как работают эти функции. Если вы переносите начало координат области вывода в точку (*xViewOrg, yViewOrg*), то логическая точка (0, 0) будет соответствовать физической точке с координатами (*xViewOrg, yViewOrg*). Если вы переносите начало координат окна в точку (*xWinOrg, yWinOrg*), то логическая точка (*xWinOrg, yWinOrg*) будет соответствовать физической точке с координатами (0, 0), т. е. левому верхнему углу рабочей области.

Например, предположим, что рабочая область вашего окна имеет ширину *cxClient* и высоту *cyClient* пикселей. Если вы хотите установить начало логической системы координат — точку (0, 0) — в центр рабочей зоны окна, вы можете это сделать так:

```
SetViewportOrgEx(hdc, cxClient / 2, cyClient / 2, NULL);
```



Аргументы функции *SetViewportOrgEx* всегда задаются в координатах устройства. Логическая точка (0, 0) будет теперь отображаться в точку с физическими координатами (*cxClient/2, cyClient/2*). Теперь вы используете рабочую область так, как будто бы она имела представленную ниже систему координат:

Значения логической координаты *x* могут изменяться в диапазоне от —*cxClient/2* до *cxClient/2*. Значения логической координаты *y* могут изменяться в диапазоне от — *cyClient/2* до *cyClient/2*. Правый нижний угол рабочей области — точка с логическими координатами (*cxClient/2, cyClient/2*). Если вы хотите вывести текст, начиная от верхнего левого угла рабочей зоны, имеющего физические координаты (0, 0), вам необходимо задать отрицательные координаты:

```
TextOut(hdc, -cxClient / 2, -cyClient / 2, "Hello", 5);
```

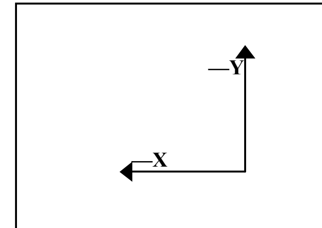
Вы можете добиться того же результата, используя функцию *SetWindowOrgEx* вместо функции *SetViewportOrgEx*:

```
SetWindowOrgEx(hdc, -cxClient / 2, -cyClient / 2, NULL);
```

Аргументы функции *SetWindowOrgEx* всегда задаются в логических координатах. После этого вызова логическая точка (—*cxClient/2,* —*cyClient/2*) соответствует физической точке (0, 0) — левому верхнему углу рабочей области.

То, чего вы не должны делать (до тех пор, пока вы не будете знать, к чему это приведет) — это использовать обе функции совместно:

```
SetViewportOrgEx(hdc, cxClient / 2, cyClient / 2, NULL);

SetWindowOrgEx(hdc, -cxClient / 2, -cyClient / 2, NULL);
```



Это означает, что логическая точка (—*cxClient/2,* —*cyClient/2*) соответствует физической точке (*cxClient/2, cyClient/2*), представляя вам такую систему координат:

Вы можете получить текущее положение начала координат области вывода и окна, используя функции:

```
GetViewportOrgEx(hdc, &pt);

GetWindowOrgEx(hdc, &pt);
```

где *pt* — структура типа POINT (точка).

Функция *GetViewportOrgEx* возвращает значение в координатах устройства, а функция *GetWindowOrgEx* — в логических координатах.

Вы можете изменить начала координат области вывода и окна так, чтобы сдвинуть экранный вывод в рабочую зону вашего окна — например, в ответ на изменение пользователем состояния полосы прокрутки. Изменение начала координат области вывода или окна не приводит к немедленному сдвигу экранного вывода. Например, в программе SYSMETS2 из главы 2 мы использовали значение *iVscrollPos* (текущее положение вертикальной полосы прокрутки) для вычисления соответствующей координаты *y* экранного вывода:

```
case WM_PAINT :
       BeginPaint(hwnd, &ps);

       for (i = 0; i < NUMLINES; i++)
       {
              y = cyChar *(1 — iVscrollPos + i);
              [вывод текста]
       }

       EndPaint(hwnd, &ps);
       return 0;
```

Мы можем добиться того же результата, используя функцию *SetWindowOrgEx*:

```
case WM_PAINT :
       BeginPaint(hwnd, &ps);

       SetWindowOrgEx(ps.hdc, 0, cyChar * iVscrollPos);

       for (i = 0; i < NUMLINES; i++)
       {
```

```
        y = cyChar *(1 + i);
        [вывод текста]
    }

    EndPaint(hwnd, &ps);
    return 0;
```

Теперь вычисление координаты *y* для функции *TextOut* не требует значения *iVscrollPos*. Это означает, что вы можете поместить функции вывода текста в подпрограмму, не передавая в нее значение *iVscrollPos*, так как мы настраиваем вывод текста путем изменения начала координат окна.

Если у вас есть опыт работы с прямоугольной (Декартовой) системой координат, то перенос логической точки (0, 0) в центр рабочей области, как мы сделали ранее, может показаться вам полезным. Тем не менее, тут есть небольшая проблема в режиме MM_TEXT: обычно в Декартовой системе координат значение координаты *y* увеличивается при перемещении вверх, а в режиме MM_TEXT — вниз. В этом смысле режим MM_TEXT несколько странноват, тогда как следующие пять режимов отображения делают это корректно.

Метрические режимы отображения

Windows включает пять режимов отображения для выражения логических координат в физических единицах измерения. Поскольку логические координаты по осям *x* и *y* преобразуются в одинаковые физические единицы измерения, эти режимы отображения помогают вам рисовать круглые окружности и квадратные квадраты.

Пять метрических режимов отображения (metric mapping modes) приведены ниже в порядке возрастания точности. Для сравнения в двух правых столбцах приведены размеры логических единиц в дюймах и миллиметрах (мм).

Режим отображения	**Логическая единица**	**Дюймы**	**Миллиметры**
MM_LOENGLISH	0.01 дюйма	0.01	0.254
MM_LOMETRIC	0.1 миллиметра	0.00394	0.1
MM_HIENGLISH	0.001 дюйма	0.001	0.0254
MM_TWIPS*	1/1440 дюйма	0.000694	0.0176
MM_HIMETRIC	0.01 миллиметра	0.000394	0.01

Для того, чтобы дать вам представление о том, как разрешение режима MM_TEXT соотносится с этими разрешениями, скажем, что на стандартном дисплее VGA, каждый пиксель которого имеет размер 0.325 мм в ширину и высоту, физические VGA — координаты грубее, чем логические координаты в любом из метрических режимов отображения.

На лазерном принтере с разрешением 300 точек/дюйм каждый пиксель имеет размер 0.0033 дюйма — это более высокое разрешение, чем в режимах MM_LOENGLISH и MM_LOMETRIC, но более низкое, чем в режимах MM_HIENGLISH, MM_TWIPS и MM_HIMETRIC.

Начала координат и протяженности, заданные по умолчанию, приведены ниже:

Начало координат окна: (Window origin)	(0, 0)	Может быть изменено
Начало координат области вывода: (Viewport origin)	(0, 0)	Может быть изменено
Протяженность окна: (Window extent)	(?, ?)	Не может быть изменена
Протяженность области вывода: (Viewport extent)	(?, ?)	Не может быть изменена

Протяженности окна и области вывода зависят от режима отображения и коэффициента сжатия (aspect ratio) устройства (отношения высоты пикселя к его ширине). Как уже отмечалось ранее, протяженности сами по себе не имеют смысла. Имеет смысл только их отношение. Приведем формулы преобразования координат еще раз:

$$xViewport = (xWindow - xWinOrg) \times (xViewExt/xWinExt) + xViewOrg$$

$$yViewport = (yWindow - yWinOrg) \times (yViewExt/yWinExt) + yViewOrg,$$

В режиме MM_LOENGLISH, например, Windows вычисляет протяженности таким образом, чтобы соблюдались следующие соотношения:

xViewExt/xWinExt = число пикселей по горизонтали в 0.01 дюйма

-yViewExt/yWinExt = число пикселей по вертикали в 0.01 дюйма, взятое со знаком минус

Для многих дисплейных устройств (таких как VGA), коэффициент сжатия будет меньше 1. Поскольку Windows работает только с целыми числами, использование коэффициента сжатия более рационально, чем использование

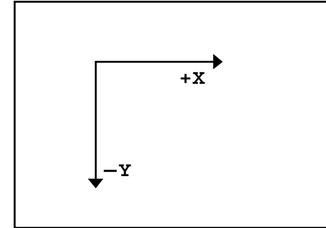
---

* Определенная выше единица измерения *twip* равна 1/20 точки (которая равна 1/72 дюйма) и точно равна 1/1440 дюйма.

абсолютных значений масштабных коэффициентов, для снижения погрешности при преобразовании логических и физических координат.

Обратите внимание на знак минус перед отношением протяженностей для вертикальной оси. Этот минус изменяет направление оси *y*.

Для пяти указанных режимов отображения, величина координаты *y* возрастает при движении вверх. Начала координат окна и области вывода по умолчанию равны (0, 0). Этот факт имеет интересное следствие. Когда вы впервые переключаетесь в один из этих пяти режимов отображения, система координат выглядит так:



Есть только один метод, позволяющий вам отобразить что-нибудь в рабочей области — это использование отрицательных значений координаты *y*. Например, этот код:

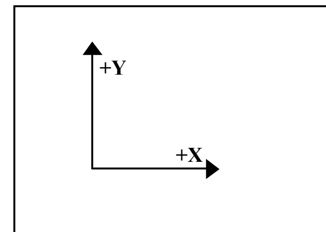
```
SetMapMode(hdc, MM_LOENGLISH);

TextOut(hdc, 100, -100, "Hello", 5);
```

выводит "Hello" с отступом в один дюйм слева и сверху рабочей области.

Чтобы избежать этого и работать в привычной системе координат, надо установить логическую точку (0, 0) в левый нижний угол рабочей зоны. Считая высоту рабочей зоны в пикселях равной *cyClient*, вы можете сделать это, вызвав функцию *SetViewportOrgEx*:

```
SetViewportOrgEx(hdc, 0, cyClient, NULL);
```

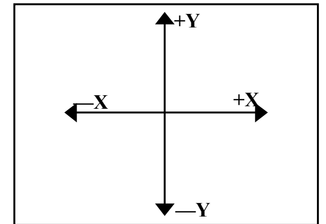
Теперь система координат выглядит так:



Аналогично, вы можете установить логическую точку (0, 0) в центр рабочей области:

```
SetViewportOrgEx(hdc, cxClient / 2, cyClient / 2, NULL);
```

Система координат будет выглядеть так:



Теперь мы имеем настоящую Декартову систему координат с одинаковыми логическими единицами измерения по осям *x* и *y* — дюймами, миллиметрами или единицами измерения twips.

Вы можете также использовать функцию *SetWindowOrgEx* для изменения логической точки (0, 0), но эта задача несколько сложнее, поскольку параметры функции должны задаваться в логических координатах. Вам следовало бы сначала преобразовать пару значений (*cxClient, cyClient*) в логические координаты, используя функцию *DPtoLP*. Считая, что переменная *pt* имеет тип структуры POINT, приведенный ниже код перемещает логическую точку (0,0) в центр рабочей области:

```
pt.x = cxClient;
pt.y = cyClient;
DPtoLP(hdc, &pt, 1);
SetWindowOrgEx(hdc, -pt.x / 2, -pt.y / 2, NULL);
```

Ваши собственные режимы отображения

Два оставшихся режима отображения называются MM_ISOTROPIC и MM_ANISOTROPIC. Это два режима отображения, в которых Windows позволяет вам изменять протяженность области вывода и окна, и тем самым менять коэффициент масштабирования, который использует Windows для преобразования логических координат в физические. Слово *isotropic* означает одинаковый во всех направлениях; *anisotropic* — неодинаковый. Также как и в метрических режимах отображения, рассмотренных ранее, MM_ISOTROPIC использует одинаковые измерения по осям. Логические единицы измерения по оси *x* имеют такое же представление в физических единицах, как и логические единицы измерения по оси *y*. Это помогает тогда, когда вам необходимо создать изображение с правильным относительным размером, независимо от относительного размера пикселя дисплея.

Отличие между режимом MM_ISOTROPIC и метрическими режимами отображения в том, что в режиме MM_ISOTROPIC вы можете управлять физическим размером логической единицы измерения. Вы можете сделать физический размер логической единицы измерения таким, что ваши рисунки всегда будут целиком содержаться в рабочей области окна, либо в уменьшенном, либо в увеличенном виде. Например, программа стрелочных часов ANACLOCK (analog clock) из главы 7 — это пример изотропного изображения. Часы всегда имеют круглую форму. Как только вы изменяете размеры окна, соответственно сразу же меняется размер изображения. Программа для Windows может обрабатывать изменение размеров изображения путем соответствующего изменения протяженностей окна и области вывода. В этом случае программа может использовать одни и те же логические единицы измерения при вызове функций рисования независимо от размеров окна.

Иногда режим отображения MM_TEXT и другие метрические режимы называют полностью принудительными режимами отображения. Это означает, что вы не имеете возможности изменять протяженности окна и области вывода, а значит, и масштаб преобразования логических координат в координаты устройства. MM_ISOTROPIC — это частично принудительный режим отображения. Windows позволяет вам изменять протяженности окна и области вывода, но система преобразует их таким образом, чтобы логические координаты *x* и *y* имели одинаковое представление о физических единицах измерения. Режим отображения MM_ANISPTROPIC — непринудительный. Вы можете изменять протяженности окна и области вывода. Система эти значения не преобразует.

Режим отображения MM_ISOTROPIC

Режим отображения MM_ISOTROPIC идеален для использования выбранных пользователем осей координат при сохранении равенства логических единиц измерения по обеим осям. Прямоугольники с равными логическими высотой и шириной отображаются как квадраты, эллипсы с равными логическими высотой и шириной отображаются как окружности.

Когда вы впервые устанавливаете режим отображения MM_ISOTROPIC, Windows использует те же протяженности окна и области вывода, что и в режиме MM_LOMETRIC. (Тем не менее, будьте внимательны!) Разница состоит в том, что теперь вы можете изменять протяженности в соответствии со своими предпочтениями, используя функции *SetWindowExtEx* и *SetViewportExtEx*. Затем Windows изменит их таким образом, чтобы логические координаты по обеим осям имели одинаковое представление в физических координатах.

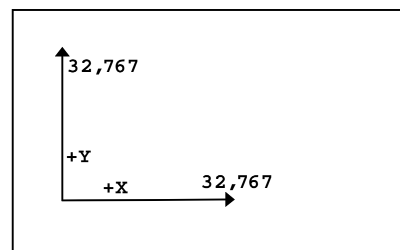
Чаще всего вы будете использовать параметры функции *SetWindowExtEx* как желаемые логические размеры логического окна, а параметры функции *SetViewportExtEx* — как текущую высоту и ширину рабочей области. Когда Windows преобразует эти протяженности, она стремится привести в соответствие логическое окно к физической области вывода, что может привести в результате к тому, что часть области вывода окажется за пределами логического окна. Нужно вызывать функцию *SetWindowExtEx* до вызова функции *SetViewportExtEx* для того, чтобы сделать использование пространства рабочей области максимально эффективным.

Например, предположим, вы хотите иметь традиционную реальную систему координат с одним квадрантом, где точка (0, 0) — левый нижний угол рабочей области, а ширина и высота меняются в интервале от 0 до 32767. Вы также хотите, чтобы единицы по осям *x* и *y* имели одинаковое физическое представление. Для этого сделайте следующее:

```
SetMapMode(hdc, MM_ISOTROPIC);
SetWindowExtEx(hdc, 32767, 32767, NULL);
SetViewportExtEx(hdc, cxClient, -cyClient, NULL);
SetViewportOrgEx(hdc, 0, cyClient, NULL);
```

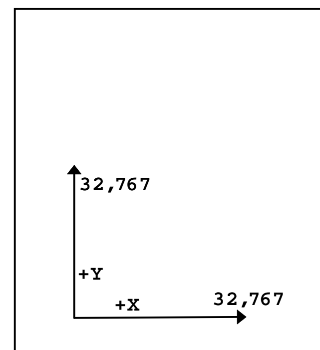
Если затем вы получите значения протяженностей окна и области вывода при помощи *GetWindowExtEx* и *GetViewportExtEx*, то они не будут равны заданным вами значениям. Windows преобразует протяженности на базе коэффициента сжатия устройства отображения (aspect ratio) так, чтобы логические единицы измерения по обеим осям имели одинаковые физические размеры.

Если ширина рабочей области больше чем высота (в физических единицах), то Windows изменяет протяженность по *x* так, что логическое окно становится уже, чем физическая область вывода. Логическое окно помещается в левой части рабочей области:



Вы не можете отображать что-либо в правой части рабочей области за границей оси *x*, поскольку это требует задания логической координаты *x* со значением, превышающим 32767.

Если высота рабочей области меньше чем ширина (в физических единицах), то Windows изменяет протяженность по *y*. Логическое окно помещается в нижней части рабочей области:

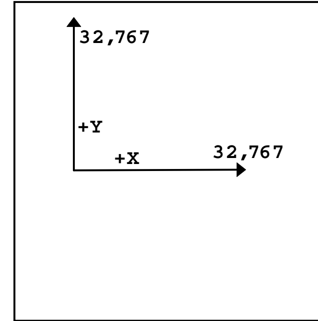


Теперь вы не можете отображать что-либо в верхней части рабочей области, поскольку это требует задания логической координаты *y* со значением, превышающим 32767.

Если вы предпочитаете, чтобы логическое окно всегда находилось слева вверху рабочей области, то вы можете изменить приведенный выше код следующим образом:

```
SetMapMode(hdc, MM_ISOTROPIC);
SetWindowExtEx(hdc, 32767, 32767, NULL);
SetViewportExtEx(hdc, cxClient, -cyClient, NULL);
SetWindowOrgEx(hdc, 0, 32767, NULL);
```

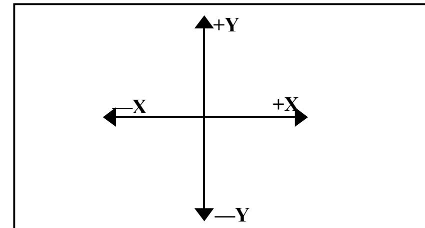
Делая вызов функции *SetWindowOrgEx*, мы хотим, чтобы логическая точка (0, 32767) соответствовала точке с физическими координатами (0, 0). Теперь, если высота рабочей области окажется больше ширины, то система координат будет расположена следующим образом:



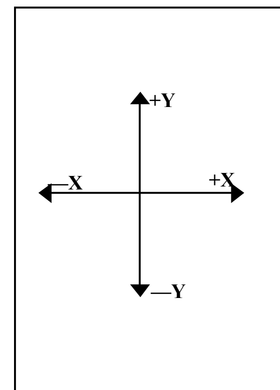
Для изображений, похожих на изображение из программы ANACLOCK, вы можете использовать обычную, имеющую четыре квадранта, Декартову систему координат с масштабируемыми по вашему выбору осями в четырех направлениях и логической точкой (0, 0) в центре рабочей области. Если вы, например, хотите, чтобы диапазон значений по каждой из осей был от 0 до 1000, используйте такой код:

```
SetMapMode(hdc, MM_ISOTROPIC);
SetWindowExtEx(hdc, 1000, 1000, NULL);
SetViewportExtEx(hdc, cxClient / 2, -cyClient / 2, NULL);
SetViewportOrgEx(hdc, cxClient / 2, cyClient / 2, NULL);
```

Если ширина рабочей области будет больше высоты, то система логических координат будет выглядеть так:



Система логических координат также будет центрирована, если высота рабочей области окажется больше ширины:



Запомните, что отсечения протяженностей окна и области вывода не предполагается. При вызове функций GDI вы можете использовать логические координаты *x* и *y* со значениями меньшими —1000 и больше +1000. В зависимости от формы рабочей области эти точки могут быть видимыми и невидимыми.

В режиме отображения MM_ISOTROPIC можно сделать логические единицы измерения намного больше, чем пиксели. Например, предположим, что вы хотите установить режим отображения так, чтобы точка (0, 0) была в левом верхнем углу экрана, значения координаты *y* увеличивались при движении вниз (как в режиме MM_TEXT), а логические координаты равнялись $^{1}/_{16}$ дюйма. Этот режим отображения позволит вам нарисовать линейку, начинающуюся в левом верхнем углу рабочей области, с делениями, равными $^{1}/_{16}$ дюйма:

```
SetMapMode(hdc, MM_ISOTROPIC);
SetWindowExtEx(hdc, 160 * GetDeviceCaps(hdc, HORZSIZE) / 254,
               160 * GetDeviceCaps(hdc, VERTSIZE) / 254, NULL);
SetViewportExtEx(hdc, GetDeviceCaps(hdc, HORZRES),
                 GetDeviceCaps(hdc, VERTRES), NULL);
```

В этом коде протяженность области вывода устанавливается равной величине всего экрана в пикселях. Протяженности окна должны быть установлены в размеры всего экрана в единицах, равных $^{1}/_{16}$ дюйма. Используя индексы HORZSIZE и VERTSIZE в функции *GetDeviceCaps*, мы получаем размеры экрана в миллиметрах. Если бы мы работали с числами с плавающей точкой, мы могли бы преобразовать миллиметры в дюймы путем деления на 25.4, а затем преобразовать дюймы в единицы, равные $^{1}/_{16}$ дюйма, умножив результат деления на 16. Но, поскольку, мы работаем с целыми числами, мы должны умножить миллиметры на 160 и разделить на 254.

Для большинства устройств вывода этот код делает логическую единицу измерения много большей, чем физическая единица измерения. Все, что вы выводите на устройство, будет иметь физические координаты, определенные с точностью до 1/16 дюйма. Вы не можете нарисовать две горизонтальные линии, отстоящие друг от друга на 1/32 дюйма, так как это потребовало бы задания дробных логических координат.

MM_ANISOTROPIC: растягивание изображения

Когда вы устанавливаете протяженности области вывода и окна в режиме отображения MM_ISOTROPIC, Windows преобразует эти значения так, чтобы логические единицы по обеим осям имели одинаковое выражение в физических единицах. В режиме отображения MM_ANISOTROPIC Windows не осуществляет этого преобразования. Это означает, что режим MM_ANISOTROPIC не обязательно создает правильный коэффициент сжатия (aspect ratio).

С одной стороны, можно использовать режим MM_ANISOTROPIC в тех случаях, когда имеются произвольные координаты рабочей области, так же как и в режиме MM_ISOTROPIC. Приведенный ниже код устанавливает точку (0, 0) в левый нижний угол рабочей области и с диапазоном по осям *x* и *y* от 0 до 32767:

```
SetMapMode(hdc, MM_ANISOTROPIC);
SetWindowExtEx(hdc,32767, 32767, NULL);
SetViewportExtEx(hdc, cxClient, -cyClient, NULL);
SetViewportOrgEx(hdc, 0, cyClient, NULL);
```

В режиме MM_ISOTROPIC похожий фрагмент программы приводил к тому, что часть рабочей области оказывалась за границами осей координат. В режиме MM_ANISOTROPIC правый верхний угол рабочей области — это всегда точка (32767, 32767) независимо от размеров. Если рабочая область не квадратная, то логические координаты *x* и *y* будут иметь различные физические размерности.

В предыдущем разделе говорилось о том, что в режиме отображения MM_ISOTROPIC в рабочей области можно создавать изображения, подобные изображению программы ANACLOCK, где оси *x* и *y* ранжированы от -1000 до 1000. Вы можете сделать нечто похожее в режиме MM_ANISOTROPIC:

```
SetMapMode(hdc, MM_ANISOTROPIC);
SetWindowExtEx(hdc, 1000, 1000, NULL);
SetViewportExtEx(hdc, cxClient / 2, -cyClient / 2, NULL);
SetViewportOrgEx(hdc, cxClient / 2, cyClient / 2, NULL);
```

Разница состоит в том, что в режиме MM_ANISOTROPIC часы, как правило, представлены в виде эллипса, а не окружности.

С другой стороны, вы можете использовать MM_ANISOTROPIC для установки фиксированных, но не равных друг другу, единиц измерения. Например, если ваша программа занимается только выводом текста, вы можете установить грубые координаты на базе высоты и ширины простого символа:

```
SetMapMode(hdc, MM_ANISOTROPIC);
SetWindowExtEx(hdc, 1, 1, NULL);
SetViewportExtEx(hdc, cxChar, cyChar, NULL);
```

(Здесь предполагается, что *cxChar* и *cyChar* — ширина и высота символа в пикселях для непропорционального шрифта.) Теперь вы можете в вызове функции *TextOut* задавать координаты символов как строку и столбец, не используя пиксельные координаты. Например, следующая инструкция выводит текст "Hello" с отступом в три символа слева и два символа сверху:

```
TextOut(hdc, 3, 2, "Hello", 5);
```

Это очень похоже на работу в среде MS DOS, а не Windows.

Когда вы впервые устанавливаете режим отображения MM_ANISOTROPIC, он всегда наследует значения протяженностей от предыдущего установленного режима. Это может быть очень удобно. Режим MM_ANISOTROPIC как бы снимает "блокировку" протяженности, т. е. позволяет изменять значения протяженностей. В этом его отличие от полностью принудительных метрических режимов отображения. Например, предположим, используется режим отображения MM_LOENGLISH, и требуется, чтобы логическая единица измерения равнялась 0.01 дюйма. Причем нежелательно, чтобы значения по координате *y* увеличивались при движении вверх, нужно, чтобы, как в режиме MM_TEXT, значения координаты *y* увеличивались при движении вниз. Ниже приведен код, реализующий это:

```
SIZE   size;
[другие строки программы]
SetMapMode(hdc, MM_LOENGLISH);
SetMapMode(hdc, MM_ANISOTROPIC);

GetViewportExtEx(hdc, &size);

SetViewportExtEx(hdc, size.cx, -size.cy, NULL );
```

Сначала мы устанавливаем режим отображения MM_LOENGLISH. Затем мы даем возможность изменять протяженности, устанавливая режим отображения MM_ANISOTROPIC. Функция *GetViewportExtEx* записывает протяженности области вывода в поля структуры SIZE. Затем мы вызываем функцию *SetViewportExtEx* с теми же значениями протяженностей, за исключением того, что протяженность по оси *y* делается отрицательной.

Программа WHATSIZE

Мы будем использовать различные режимы отображения по мере того, как будем далее изучать функции GDI в следующих главах. Сейчас же давайте взглянем на размер рабочей области в терминах дюймов и миллиметров. Программа WHATSIZE, приведенная на рис. 4.19, отображает размер рабочей области в терминах единиц, ассоциированных с шестью полностью принудительными режимами отображения: MM_TEXT, MM_LOMETRIC, MM_HIMETRIC, MM_LOENGLISH, MM_HIENGLISH и MM_TWIPS.

WHATSIZE.MAK

```
#-----------------------
# WHATSIZE.MAK make file
#-----------------------
```

```
whatsize.exe : whatsize.obj
     $(LINKER) $(GUIFLAGS) -OUT:whatsize.exe whatsize.obj $(GUILIBS)

whatsize.obj : whatsize.c
     $(CC) $(CFLAGS) whatsize.c
```

WHATSIZE.C

```
/*-----------------------------------------
   WHATSIZE.C -- What Size is the Window?
                 (c) Charles Petzold, 1996
  -----------------------------------------*/

#include <windows.h>
#include <stdio.h>

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "WhatSize";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "What Size is the Window?",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

void Show(HWND hwnd, HDC hdc, int xText, int yText, int iMapMode,
           char *szMapMode)
     {
     char szBuffer [60];
     RECT rect;

     SaveDC(hdc);
```

```
     SetMapMode(hdc, iMapMode);
     GetClientRect(hwnd, &rect);
     DPtoLP(hdc,(PPOINT) &rect, 2);
     RestoreDC(hdc, -1);
     TextOut(hdc, xText, yText, szBuffer,
               sprintf(szBuffer, "%-20s %7d %7d %7d %7d", szMapMode,
                    rect.left, rect.right, rect.top, rect.bottom));
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char  szHeading [] =
                    "Mapping Mode            Left   Right     Top  Bottom";
     static char  szUndLine [] =
                    "------------            ----   -----     ---  ------";
     static int   cxChar, cyChar;
     HDC          hdc;
     PAINTSTRUCT  ps;
     TEXTMETRIC   tm;

     switch(iMsg)
          {
          case WM_CREATE:
               hdc = GetDC(hwnd);
               SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));

               GetTextMetrics(hdc, &tm);
               cxChar = tm.tmAveCharWidth;
               cyChar = tm.tmHeight + tm.tmExternalLeading;

               ReleaseDC(hwnd, hdc);
               return 0;

          case WM_PAINT:
               hdc = BeginPaint(hwnd, &ps);
               SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));

               SetMapMode(hdc, MM_ANISOTROPIC);
               SetWindowExtEx(hdc, 1, 1, NULL);
               SetViewportExtEx(hdc, cxChar, cyChar, NULL);

               TextOut(hdc, 1, 1, szHeading, sizeof szHeading - 1);
               TextOut(hdc, 1, 2, szUndLine, sizeof szUndLine - 1);

               Show(hwnd, hdc, 1, 3, MM_TEXT,      "TEXT(pixels)");
               Show(hwnd, hdc, 1, 4, MM_LOMETRIC,  "LOMETRIC(.1 mm)");
               Show(hwnd, hdc, 1, 5, MM_HIMETRIC,  "HIMETRIC(.01 mm)");
               Show(hwnd, hdc, 1, 6, MM_LOENGLISH, "LOENGLISH(.01 in)");
               Show(hwnd, hdc, 1, 7, MM_HIENGLISH, "HIENGLISH(.001 in)");
               Show(hwnd, hdc, 1, 8, MM_TWIPS,     "TWIPS(1/1440 in)");

               EndPaint(hwnd, &ps);
               return 0;
          case WM_DESTROY:
               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

Рис. 4.19 Программа WHATSIZE

Для упрощения вывода информации с использованием функции *TextOut* программа WHATSIZE использует режим отображения MM_ANISOTROPIC с логическими координатами, установленными на базе размеров символа:

```
SetMapMode(hdc, MM_ANISOTROPIC);
SetWindowExtEx(hdc, 1, 1, NULL);
SetViewportExtEx(hdc, cxChar, cyChar, NULL);
```

После этого программа может задавать логические координаты при вызове функции *TextOut* в координатах строки и столбца при использовании непропорционального шрифта.

Когда программе WHATSIZE нужно получить размер рабочей зоны для каждого из шести режимов отображения, она сохраняет текущий контекст устройства, устанавливает новый режим отображения, получает координаты рабочей области, преобразует их в логические координаты и восстанавливает предыдущий режим отображения до того, как отобразить информацию. Этот код содержится в функции *Show* программы WHATSIZE:

```
SaveDC(hdc);

SetMapMode(hdc, iMapMode);
GetClientRect(hwnd, &rect);
DPtoLP(hdc,(PPOINT) &rect, 2);

RestoreDC(hdc, -1);
```

На рис. 4.20 представлен типичный вывод программы WHATSIZE.

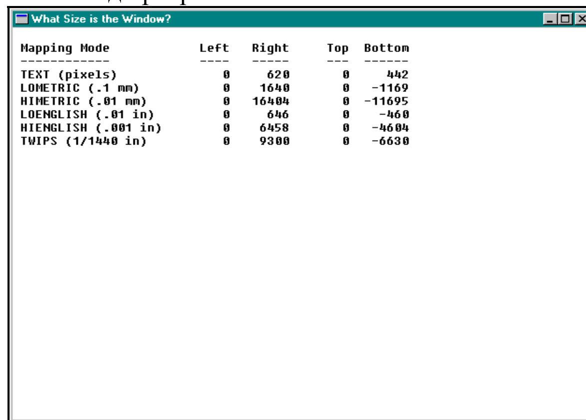


Рис. 4.20 Типичный вывод программы WHATSIZE

Обратите внимание, что здесь использовался идентификатор SYSTEM_FIXED_FONT для выбора непропорционального шрифта. Мы вскоре обсудим это.

Прямоугольники, регионы и отсечение

Microsoft Windows 95 включает несколько функций рисования, которые работают со структурами типа RECT (прямоугольник) и "регионами" (regions). Регион — это область экрана, представляющая собой комбинацию прямоугольников, полигонов и эллипсов.

Работа с прямоугольниками

Ниже приведены три функции рисования, требующие указателя на структуру прямоугольника:

```
FillRect(hdc, &rect, hBrush);
FrameRect(hdc, &rect, hBrush);
InvertRect(hdc, &rect);
```

Параметр *rect* в этих функциях представляет собой структуру типа RECT, имеющую четыре поля: *left, top, right, bottom*. Координаты в этой структуре представляются в логических единицах.

Функция *FillRect* закрашивает прямоугольник (не включая правую и нижнюю координаты) заданной кистью. Эта функция не требует, чтобы кисть была предварительно выбрана в контекст устройства.

Функция *FrameRect* использует кисть для рисования прямоугольной рамки, но не закрашивает внутреннюю область прямоугольника. Использование кисти для рисования рамки может показаться несколько странным, поскольку в функциях, рассмотренных нами ранее, таких как *Rectangle*, для рисования рамки использовалось текущее перо. *FrameRect* позволяет вам рисовать прямоугольную рамку, причем не только чистым цветом. Эта рамка имеет ширину, равную одной логической единице. Если логические единицы больше физических единиц, то рамка будет иметь ширину 2 или более пикселей.

Функция *InvertRect* инвертирует все пиксели в прямоугольнике, устанавливая все единичные биты в ноль, а нулевые — в единицу. Таким образом, функция переводит белую область в черную, черную — в белую, зеленую — в фиолетовую.

Windows содержит также девять функций, позволяющих вам легко манипулировать со структурами типа RECT. Например, для установки всех полей структуры RECT в заданные значения, при обычной записи вам пришлось бы написать следующее:

```
rect.left = xLeft;
rect.top = yTop;
rect.right = xRight;
rect.bottom = yBottom;
```

Вызывая функцию *SetRect*, вы можете добиться того же результата в одной строке программы:

```
SetRect(&rect, xLeft, yTop, xRight, yBottom);
```

Другие восемь функций тоже могут быть удобны в использовании, если вам необходимо сделать одно из следующих действий:

- Переместить прямоугольник на заданное число координат вдоль осей *x* и *y*:

  ```
  OffsetRect(&rect, x, y);
  ```

- Увеличить или уменьшить размеры прямоугольника:

  ```
  InflateRect(&rect, x, y);
  ```

- Установить поля структуры прямоугольника в ноль:

  ```
  SetRectEmpty(&rect);
  ```

- Скопировать один прямоугольник в другой:

  ```
  CopyRect(&DestRect, &SrcRect);
  ```

- Получить пересечение двух прямоугольников:

  ```
  IntersectRect(&DestRect, &SrcRect1, &SrcRect2);
  ```

- Получить объединение двух прямоугольников:

  ```
  UnionRect(&DestRect, &SrcRect1, &SrcRect2);
  ```

- Определить, является ли прямоугольник пустым:

  ```
  bEmpty = IsRectEmpty(&rect);
  ```

- Определить, содержится ли точка внутри прямоугольника:

  ```
  bInRect = PtInRect(&rect, point);
  ```

В большинстве случаев код, соответствующий этим функциям, достаточно прост. Например, вы можете повторить работу функции *CopyRect* так:

```
DestRect = SrcRect;
```

Случайные прямоугольники

В любой графической системе есть своя забавная программа, которая работает непрерывно, просто выводя причудливые серии изображений, имеющих случайные размеры и цвета. Такую программу можно сделать и в Windows. Но это не так просто, как кажется. Надеемся, что вы сможете реализовать это, но только не вставкой цикла *while* (TRUE) в обработку сообщения WM_PAINT. Конечно, такой вариант будет работать, но ваша программа перестанет реагировать на другие сообщения. Ее нельзя будет завершить или свернуть.

Единственно возможная альтернатива состоит в установке таймера Windows для посылки сообщений WM_TIMER в функцию вашего окна. (На этом мы остановимся в главе 7.) При обработке каждого сообщения WM_TIMER вы получаете контекст устройства с помощью функции *GetDC*, рисуете случайный прямоугольник, и затем освобождаете контекст устройства функцией *ReleaseDC*. Правда это уменьшает привлекательность вашей программы, поскольку она рисует случайные прямоугольники не так быстро, как возможно. Она ждет очередного сообщения WM_TIMER, а это зависит от точности системных часов.

В Windows существует достаточно "мертвого времени" — времени, когда все очереди сообщений пусты, и Windows только и делает, что ожидает ввода с клавиатуры или мыши. Можем ли мы как-нибудь получить управление в течение "мертвого времени" и нарисовать прямоугольник, возвращая управление, как только в

очереди сообщений программы появятся сообщения? Да, в этом и состоит одна из основных задач функции *PeekMessage*. Ниже приведен пример обращения к функции *PeekMessage*:

```
PeekMessage(&msg, NULL, 0, 0, PM_REMOVE);
```

Первые четыре параметра функции (указатель на структуру MSG, описатель окна и два значения, указывающих диапазон сообщений) идентичны параметрам функции *GetMessage*. Установка второго, третьего и четвертого параметров в NULL или 0, означает, что мы хотим получать из функции *PeekMessage* все сообщения для всех окон программы. Последний параметр установлен в PM_REMOVE, чтобы сообщения удалялись из очереди. Чтобы сообщения не удалялись из очереди, вы можете установить его в PM_NOREMOVE. Поэтому функция *PeekMessage* имеет префикс "peek", а не "get". Это позволяет программе проверять следующее сообщение в очереди без его удаления.

Функция *GetMessage* не возвращает управление до тех пор, пока не извлечет сообщение из очереди сообщений. Напротив, функция *PeekMessage* всегда сразу возвращает управление независимо от того, есть ли сообщения в очереди или нет. Если в очереди сообщений есть хоть одно сообщение, то возвращаемое функцией *PeekMessage* значение равно TRUE (не ноль). Если же в очереди нет сообщений, то это значение равно FALSE (0).

Это позволяет нам переписать обычный цикл сообщений:

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

таким образом:

```
while(TRUE)
{
        if(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
                if(msg.message == WM_QUIT)
                        break;
                TranslateMessage(&msg);
                DispatchMessage(&msg);
        }
        else
        {
        [другие строки программы для выполнения работы]
        }
}
return msg.wParam;
```

Обратите внимание, что сообщение WM_QUIT проверяется особо. Вы не должны это делать в обычном цикле сообщений, поскольку функция *GetMessage* возвращает FALSE (0) при извлечении сообщения WM_QUIT. А возвращаемое значение функции *PeekMessage* показывает, извлечено ли сообщение из очереди. Поэтому проверка сообщения WM_QUIT необходима.

Если значение, возвращенное функцией *PeekMessage* равно TRUE, то сообщение обрабатывается обычным образом. В противном случае программа может выполнить какую-либо работу (например, нарисовать очередной случайный прямоугольник) перед тем, как вернуть управление Windows.

(В документации по Windows сказано, что вы не можете использовать функцию *PeekMessage* для удаления из очереди сообщения WM_PAINT, хотя на самом деле особой проблемы в этом нет. Более того, функция *GetMessage* тоже не удаляет сообщение WM_PAINT из очереди сообщений. Есть только один путь удалить сообщение WM_PAINT из очереди. Это можно осуществить, сделав активным недействительный регион рабочей области окна. Для этого используются функции *ValidateRect, ValidateRgn* или пара функций *BeginPaint* и *EndPaint*. Если вы стандартным образом обрабатываете сообщение WM_PAINT после его извлечения из очереди функцией *PeekMessage*, то у вас не будет никаких проблем. Вы не можете использовать приведенный ниже код для очистки очереди сообщений:

```
while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE));
```

Эта инструкция извлекает и уничтожает все сообщения из очереди сообщений за исключением WM_PAINT. Если в очереди окажется сообщение WM_PAINT, вы получите бесконечный цикл.)

Функция *PeekMessage* занимала значительно более важное место в предыдущих версиях Windows, так как 16-битная Windows работала на основе невытесняющей многозадачности (об этом будет рассказано в главе 14). Программа Windows Terminal использовала цикл *PeekMessage* для проверки данных, поступающих от коммуникационного порта. Программа Print Manager использовала эту функцию для печати. Кроме того Windows-

приложения, осуществлявшие вывод на печать, как правило, также использовали цикл *PeekMessage*. В вытесняющей многозадачной среде Windows 95 программа может создать множество потоков выполнения. Мы рассмотрим это в следующих главах.

Пользуясь только функцией *PeekMessage*, мы, тем не менее, можем написать программу, неумолимо выводящую случайные прямоугольники. Программа RANDRECT приведена на рис. 4.21.

RANDRECT.MAK

```
#------------------------
# RANDRECT.MAK make file
#------------------------

randrect.exe : randrect.obj
     $(LINKER) $(GUIFLAGS) -OUT:randrect.exe randrect.obj $(GUILIBS)

randrect.obj : randrect.c
     $(CC) $(CFLAGS) randrect.c
```

RANDRECT.C

```
/*------------------------------------------
   RANDRECT.C -- Displays Random Rectangles
                 (c) Charles Petzold, 1996
  ------------------------------------------*/

#include <windows.h>
#include <stdlib.h>

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

int cxClient, cyClient;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "RandRect";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;
     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Random Rectangles",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

     while(TRUE)
```

```
          {
          if(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
               {
               if(msg.message == WM_QUIT)
                    break;

               TranslateMessage(&msg);
               DispatchMessage(&msg);
               }
          else
               DrawRectangle(hwnd);
          }

     return msg.wParam;
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     switch(iMsg)
          {
          case WM_SIZE:
               cxClient = LOWORD(lParam);
               cyClient = HIWORD(lParam);
               return 0;
          case WM_DESTROY:
               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }

void DrawRectangle(HWND hwnd)
     {
     HBRUSH hBrush;
     HDC    hdc;
     RECT   rect;

     if(cxClient == 0 || cyClient == 0)
          return;

     SetRect(&rect, rand() % cxClient, rand() % cyClient,
                    rand() % cxClient, rand() % cyClient);


     hBrush = CreateSolidBrush(RGB(rand() % 256, rand() % 256,
                                              rand() % 256));
     hdc = GetDC(hwnd);

     FillRect(hdc, &rect, hBrush);

     ReleaseDC(hwnd, hdc);

     DeleteObject(hBrush);
     }
```

Рис. 4.21 Программа RANDRECT

Программа применяет функции *SetRect* и *FillRect*, которые были описаны выше, с координатами прямоугольника и цветом сплошной кисти, вычисленными на основе случайных величин, полученных от функции *rand* языка С. В главе 14 будет приведена другая версия этой программы, основанная на принципе многопоточности.

Создание и рисование регионов

Регион — это описание области дисплея, состоящей из комбинации прямоугольников, многоугольников и эллипсов. Вы можете использовать регионы для рисования или для отсечения. Регион для отсечения, другими словами, ограничения рисования в заданной области рабочей зоны выбирается в контекст устройства. Регионы, так же как перья, кисти и битовые образы тоже являются объектами GDI. Любой регион, созданный вами ранее, следует удалять с помощью функции *DeleteObject*.

Когда вы создаете регион, Windows возвращает описатель региона, имеющий тип HRGN. Простейший тип региона — это прямоугольник. Вы можете создать прямоугольный регион одним из двух способов:

```
hRgn = CreateRectRgn(xLeft, yTop, xRight, yBottom);
```

или

```
hRgn = CreateRectRgnIndirect(&rect);
```

Вы можете также создать эллиптические регионы, используя:

```
hRgn = CreateEllipticRgn(xLeft, yTop, xRight, yBottom);
```

или

```
hRgn = CreateEllipticRgnIndirect(&rect);
```

Функция *CreateRoundRectRgn* строит прямоугольный регион со скругленными углами.

Создание многоугольного региона похоже на использование функции *Polygon*:

```
hRgn = CreatePolygonRgn(&point, iCount, iPolyFillMode);
```

Параметр *point* — это массив структур типа POINT, *iCount* — число точек, *iPolyFillMode* — равен либо ALTERNATE, либо WINDING. Вы можете также создать регион из множества многоугольников, используя функцию *CreatePolyPolygonRgn*.

Вы спросите: "Ну и что?" Что особенного делают эти регионы? Ниже приведена функция, которая иллюстрирует возможности регионов:

```
iRgnType = CombineRgn(hDestRgn, hSrcRgn1, hSrcRgn2, iCombine);
```

Она комбинирует два исходных региона (*hSrcRgn1* и *hSrcRgn2*) и строит третий, на который ссылается *hDestRgn*. Все три описателя регионов еще до вызова функции должны быть действительными, однако дополнительный регион, описываемый ранее *hDestRgn*, уничтожается. (Когда вы используете эту функцию, вы можете сначала сделать так, чтобы *hDestRgn* ссылался на маленький прямоугольный регион.)

Параметр *iCombine* описывает, как объединяются 2 региона с описателями *hSrcRgn1* и *hSrcrgn2*:

Значение *iCombine*	Новый регион
RGN_AND	Область пересечения двух исходных регионов
RGN_OR	Объединение двух исходных регионов
RGN_XOR	Объединение двух исходных регионов за исключением области пересечения
RGN_DIFF	Часть региона *hSrcRgn1*, не входящая в регион *hSrcRg2*
RGN_COPY	Регион *hSrcRgn1*

Величина *iRgnType*, возвращаемая от функции *CombineRect*, принимает одно из следующих значений: NULLREGION, показывающее, что регион пуст; SIMPLEREGION, показывающее, что регион представляет собой простой прямоугольник, эллипс или многоугольник; COMPLEXREGION, показывающее, что регион представляет собой комбинацию прямоугольников, эллипсов или многоугольников; ERROR, означающее, что произошла ошибка.

Получив описатель региона, вы можете использовать его в следующих четырех функциях рисования:

```
FillRgn(hdc, hRgn, hBrush);
FrameRgn(hdc, hRgn, hBrush, xFrame, yFrame);
InvertRgn(hdc, hRgn);
PaintRgn(hdc, hRgn);
```

Функции *FillRgn*, *FrameRgn* и *InvertRgn* похожи на функции *FillRect*, *FrameRect* и *InvertRect*. Параметры *xFrame* и *yFrame* функции *FrameRect* — это логические ширина и высота рамки, которая будет нарисована вокруг региона. Функция *PaintRgn* закрашивает внутреннюю область региона текущей выбранной в контекст устройства кистью. Во всех этих функциях предполагается, что регион определен в логических координатах.

Когда вы заканчиваете работу с регионом, вы можете его удалить, используя ту же самую функцию *DeleteObject*, что и для удаления других объектов GDI:

```
DeleteObject(hRgn);
```

Отсечения: прямоугольники и регионы

Регионы также могут принимать участие в отсечении. Функция *InvalidateRect* делает недействительным прямоугольную область дисплея и генерирует сообщение WM_PAINT. Например, вы можете использовать функцию *InvalidateRect* для обновления рабочей области и генерации сообщения WM_PAINT:

```
InvalidateRect(hwnd, NULL, TRUE);
```

Вы можете получить координаты недействительного прямоугольника, вызвав функцию *GetUpdateRect*, и вы можете сделать действительным прямоугольник в рабочей области, используя функцию *ValidateRect*. Когда вы получаете сообщение WM_PAINT, координаты недействительного прямоугольника доступны из полей структуры PAINTSTRUCT, которые заполняются при вызове функции *BeginPaint*. Этот недействительный прямоугольник также определяет "регион отсечения". Вы не можете рисовать за пределами региона отсечения.

Windows содержит две функции, похожие на *InvalidateRect* и *ValidateRect*, работающие с регионами, а не с прямоугольниками:

```
InvalidateRgn(hwnd, hRgn, bErase);
```

и

```
ValidateRgn(hwnd, hRgn);
```

Когда вы получаете сообщение WM_PAINT как результат того, что регион стал недействительным, регион отсечения не обязательно будет прямоугольным.

Вы можете создать свой собственный регион отсечения, выбрав регион в контекст устройства и используя одну из двух функций:

```
SelectObject(hdc, hRgn);
```

или

```
SelectClipRgn(hdc, hRgn);
```

Регион отсечения задается в координатах устройства.

GDI делает копию региона отсечения, поэтому вы можете удалить объект-регион после выбора его в контекст устройства. Windows содержит также несколько функций для манипуляций с регионом отсечения, таких как *ExcludeClipRect* для исключения прямоугольника из региона отсечения, *IntersectClipRect* для создания нового региона отсечения, который представляет собой пересечение предыдущего региона отсечения и прямоугольника, и *OffsetClipRgn* для перемещения региона отсечения в другую часть рабочей области.

Программа CLOVER

Программа CLOVER формирует регион, состоящий из четырех эллипсов, выбирает этот регион в контекст устройства, а затем рисует набор линий, исходящих из центра рабочей области окна. Линии будут отображаться только в области, определенной регионом. Результат работы программы приведен на рис. 4.22.

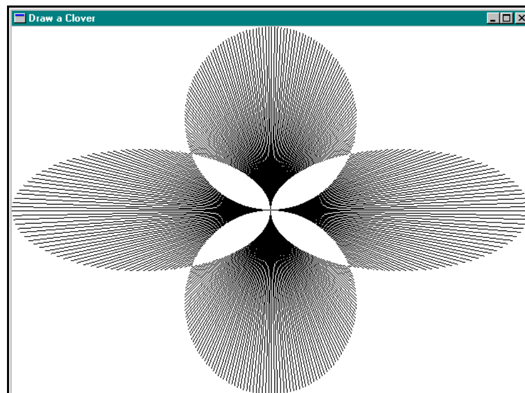


Рис. 4.22 Вывод программы CLOVER, нарисованный с использованием сложного региона отсечения

Для того, чтобы нарисовать такой рисунок традиционными методами, вам пришлось бы вычислять конечные точки для каждой прямой по формулам для расчета кривой эллипса. Используя сложный регион отсечения, вы можете рисовать линии и оставить Windows расчеты конечных точек. Программа CLOVER приведена на рис. 4.23.

CLOVER.MAK

```
#----------------------
# CLOVER.MAK make file
#----------------------

clover.exe : clover.obj
     $(LINKER) $(GUIFLAGS) -OUT:clover.exe clover.obj $(GUILIBS)

clover.obj : clover.c
     $(CC) $(CFLAGS) clover.c
```

CLOVER.C

```
/*--------------------------------------------------
   CLOVER.C -- Clover Drawing Program using Regions
               (c) Charles Petzold, 1996
  --------------------------------------------------*/

#include <windows.h>
#include <math.h>

#define TWO_PI(2.0 * 3.14159)

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Clover";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Draw a Clover",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

```
          }
     return msg.wParam;
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HRGN hRgnClip;
     static int  cxClient, cyClient;
     double      fAngle, fRadius;
     HCURSOR     hCursor;
     HDC         hdc;
     HRGN        hRgnTemp[6];
     int         i;
     PAINTSTRUCT ps;

     switch(iMsg)
          {
          case WM_SIZE:
               cxClient = LOWORD(lParam);
               cyClient = HIWORD(lParam);

               hCursor = SetCursor(LoadCursor(NULL, IDC_WAIT));
               ShowCursor(TRUE);

               if(hRgnClip)
                    DeleteObject(hRgnClip);

               hRgnTemp[0] = CreateEllipticRgn(0, cyClient / 3,
                                         cxClient / 2, 2 * cyClient / 3);
               hRgnTemp[1] = CreateEllipticRgn(cxClient / 2, cyClient / 3,
                                         cxClient, 2 * cyClient / 3);
               hRgnTemp[2] = CreateEllipticRgn(cxClient / 3, 0,
                                         2 * cxClient / 3, cyClient / 2);
               hRgnTemp[3] = CreateEllipticRgn(cxClient / 3, cyClient / 2,
                                         2 * cxClient / 3, cyClient);
               hRgnTemp[4] = CreateRectRgn(0, 0, 1, 1);
               hRgnTemp[5] = CreateRectRgn(0, 0, 1, 1);
               hRgnClip    = CreateRectRgn(0, 0, 1, 1);

               CombineRgn(hRgnTemp[4], hRgnTemp[0], hRgnTemp[1], RGN_OR);
               CombineRgn(hRgnTemp[5], hRgnTemp[2], hRgnTemp[3], RGN_OR);
               CombineRgn(hRgnClip,    hRgnTemp[4], hRgnTemp[5], RGN_XOR);

               for(i = 0; i < 6; i++)
                    DeleteObject(hRgnTemp[i]);

               SetCursor(hCursor);
               ShowCursor(FALSE);
               return 0;

          case WM_PAINT:
               hdc = BeginPaint(hwnd, &ps);

               SetViewportOrgEx(hdc, cxClient / 2, cyClient / 2, NULL);
               SelectClipRgn(hdc, hRgnClip);

               fRadius = _hypot(cxClient / 2.0, cyClient / 2.0);

               for(fAngle = 0.0; fAngle < TWO_PI; fAngle += TWO_PI / 360)
                    {
                    MoveToEx(hdc, 0, 0, NULL);
                    LineTo(hdc,(int)( fRadius * cos(fAngle) + 0.5),
                                (int)(-fRadius * sin(fAngle) + 0.5));
                    }
```

```
            EndPaint(hwnd, &ps);
            return 0;

        case WM_DESTROY:
            DeleteObject(hRgnClip);
            PostQuitMessage(0);
            return 0;
        }
    return DefWindowProc(hwnd, iMsg, wParam, lParam);
    }
```

Рис. 4.23 Программа CLOVER

Поскольку регионы всегда используют координаты устройства, программа CLOVER должна перестраивать регион каждый раз при получении сообщения WM_SIZE. Это может занять несколько секунд. Программа начинает работу, создавая четыре эллиптических региона, которые запоминаются в первых четырех элементах массива *hRgnTemp*. Затем программа строит три фиктивных региона:

```
hRgnTemp [4] = CreateRectRgn(0, 0, 1, 1);
hRgnTemp [5] = CreateRectRgn(0, 0, 1, 1);
hRgnClip     = CreateRectRgn(0, 0, 1, 1);
```

Затем комбинируются два эллиптических региона слева и справа рабочей области:

```
CombineRgn(hRgnTem[4], hRgnTemp[0], hRgnTemp[1], RGN_OR);
```

Затем аналогично комбинируются два эллиптических региона сверху и снизу рабочей области:

```
CombineRgn(hRgnTem[5], hRgnTemp[2], hRgnTemp[3], RGN_OR);
```

Окончательно, эти два комбинированных региона объединяются в *hRgnClip*:

```
CombineRgn(hRgnClip, hRgnTemp[4], hRgnTemp[5], RGN_XOR);
```

Идентификатор RGN_XOR используется для исключения области пересечения из результирующего региона. Затем, все шесть временных регионов удаляются:

```
for(i = 0; i < 6; i++)
       DeleteObject(hRgnTemp[i]);
```

Обработка сообщения WM_PAINT проста, принимая во внимание результаты. Начало координат области вывода (viewport) устанавливается в центр рабочей зоны (чтобы сделать рисование линий более простым), и регион, созданный при обработке сообщения WM_CREATE, выбирается в контекст устройства в качестве региона отсечения:

```
SetViewportOrg(hdc, xClient / 2, yClient / 2);
SelectClipRgn(hdc, hRgnClip);
```

Теперь осталось только нарисовать линии — 360 штук, отстоящих друг от друга на один градус. Длина каждой линии — переменная *fRadius*, задается равной расстоянию от центра до угла рабочей области:

```
fRadius = _hypot(xClient / 2.0, yClient / 2.0);
for(fAngle = 0.0; fAngle < TWO_PI; fAngle += TWO_PI / 360)
{
       MoveToEx(hdc, 0, 0, NULL);
       LineTo(hdc,(int)(fRadius * cos(fAngle) + 0.5), (int)(-fRadius * sin(fAngle) + 0.5));
}
```

При обработке сообщения WM_DESTROY регион удаляется:

```
DeleteObject(hRgnClip);
```

Пути

Путь — это набор прямых линий и кривых, хранящийся внутри GDI. Пути (paths) были введены в Windows в версии Windows NT. Они также поддерживаются и в Windows 95. На первый взгляд пути и регионы могут показаться очень похожими, и, в самом деле, вы можете конвертировать путь в регион и использовать путь для отсечения. Тем не менее, мы рассмотрим их отличия.

Создание и воспроизведение путей

Для того, чтобы начать определение пути, вы просто вызываете функцию:

```
BeginPath(hdc);
```

После этого вызова любая рисуемая вами линия (прямая, дуга или сплайн Безье) будет запоминаться внутри GDI как часть пути и не будет воспроизводиться в контексте устройства. Часто пути состоят из связанных друг с другом линий. Для создания связанных линий вы используете функции *LineTo, PolylineTo* и *BezierTo*, каждая из которых рисует линию, начинающуюся из текущего положения пера. Если вы изменяете текущее положение пера, используя функцию *MoveToEx*, или если вы вызываете какую-либо другую функцию рисования линии, или если вы вызываете одну из функций окна или области вывода, влияющих на изменение текущего положения пера, то вы создаете новый подпуть в рамках пути. Таким образом, путь состоит из одного или нескольких подпутей, причем каждый подпуть — это серия связанных линий.

Каждый подпуть в рамках пути может быть открыт или закрыт. Подпуть закрыт, если в нем первая точка первой связанной линии и последняя точка последней связанной линии совпадают. Подпуть закрывается вызовом функции *CloseFigure*. Эта функция закрывает подпуть, добавляя, если необходимо, прямую линию. Любой последующий вызов функции рисования начинает новый подпуть. В конце вы завершаете определение пути, вызывая функцию:

```
EndPath(hdc);
```

Теперь вы можете вызвать одну из следующих пяти функций:

```
StrokePath(hdc);
FillPath(hdc);
StrokeAndFillPath(hdc);
hRgn = PathToRegion(hdc);
SelectClipPath(hdc, iCombine);
```

Любая из этих функций уничтожает определение пути после завершения.

StrokePath рисует путь, используя текущее перо. Вы можете удивиться: в чем смысл? Почему нельзя пропустить все эти штучки с путем, и нарисовать линии нормально? Ответ вы получите очень скоро.

Другие четыре функции закрывают все открытые пути прямыми линиями. Функция *FillPath* закрашивает путь, используя текущую кисть, в соответствии с текущим режимом закрашивания многоугольников. Функция *StrokeAndFillPath* выполняет оба указанных действия. Вы можете также преобразовать путь в регион или использовать путь как область отсечения. Параметр *iCombine* — одна из RGN-констант, используемых в функции *CombineRgn*, и показывает, как путь должен комбинироваться с текущим регионом отсечения.

Пути являются более гибкими структурами по сравнению с регионами для закрашивания и отсечения, потому что регион может быть определен только как комбинация прямоугольников, эллипсов и полигонов. Пути же могут состоять из сплайнов Безье и (по крайней мере в Windows NT) дуг. В GDI пути и регионы хранятся совершенно по-разному. Путь — это определение набора прямых и кривых, а регион (в общем смысле) — набор скан-линий.

Расширенные перья

Когда вы вызываете *StrokePath*, путь воспроизводится с использованием текущего пера. Ранее в этой главе рассматривалась функция *CreatePen*, которая используется для создания объекта "перо". Одновременно с поддержкой путей в Windows NT и Windows 95 введена расширенная функция пера, называемая *ExtCreatePen*, она применяется, когда бывает удобнее создать сглаженный путь, чем рисовать линии без использования пути. Функция *ExtCreatePen* выглядит так:

```
hPen = ExtCreatePen(iStyle, iWidth, &lBrush, 0, NULL);
```

Вы можете использовать эту функцию для обычного рисования линий, но в таком случае некоторые из возможностей не реализуются в Windows 95. Даже при воспроизведении путей с ее помощью остаются параметры, которые так и не поддерживаются Windows 95, что показано выше, когда два последних параметра заданы равными 0 и NULL.

Для первого параметра функции *ExtCreatePen* вы можете использовать любой из стилей, описанных ранее для функции *CreatePen*. Кроме того, вы можете комбинировать эти стили со стилем PS_GEOMETRIC (при этом параметр *iWidth* означает ширину линии в логических единицах измерения для преобразования) или PS_COSMETIC (при этом параметр *iWidth* должен быть равен 1). В Windows 95 точечные и штриховые перья должны иметь стиль PS_COSMETIC. (Это ограничение отсутствует в Windows NT.)

Одним из параметров функции *CreatePen* является цвет; однако, функция *ExtCreatePen* для задания цвета пера стиля PS_GEOMETRIC использует кисть. Такие кисти могут быть определены как битовые образы.

Когда вы рисуете широкие линии, вас, вероятно, интересует то, как будут представлены концы линий. Когда прямые или кривые соединены, вас может также заинтересовать то, как будут представлены места соединения линий. При использовании перьев, созданных функцией *CreatePen*, эти концы и места соединения всегда будут скругленными. При работе с перьями, созданными функцией *ExtCreatePen*, у вас есть выбор. (В действительности

в Windows 95 такая возможность существует только при сглаживании пути; Windows NT обладает большей гибкостью.) Представление концов линий может быть задано путем применения в функции *ExtCreatePen* один из следующих стилей пера:

```
PS_ENDCAP_ROUND
PS_ENDCAP_SQUARE
PS_ENDCAP_FLAT
```

Аналогично, места соединения линий в пути могут быть заданы так:

```
PS_JOIN_ROUND
PS_JOIN_BEVEL
PS_JOIN_MITER
```

Стиль "bevel" отрезает конец места соединения, а стиль "miter" заостряет его. Это лучше всего иллюстрируется программой ENDJOIN, приведенной на рис. 4.24.

ENDJOIN.MAK

```
#----------------------
# ENDJOIN.MAK make file
#----------------------

endjoin.exe : endjoin.obj
     $(LINKER) $(GUIFLAGS) -OUT:endjoin.exe endjoin.obj $(GUILIBS)

endjoin.obj : endjoin.c
     $(CC) $(CFLAGS) endjoin.c
```

ENDJOIN.C

```
/*----------------------------------------
   ENDJOIN.C -- Ends and Joins Demo
               (c) Charles Petzold, 1996
  ----------------------------------------*/

#include <windows.h>

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "EndJoin";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(WNDCLASSEX);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Ends and Joins Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL);
```

```
     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int  iEnd  [] = { PS_ENDCAP_ROUND, PS_ENDCAP_SQUARE,
                              PS_ENDCAP_FLAT };
     static int  iJoin [] = { PS_JOIN_ROUND,   PS_JOIN_BEVEL,
                              PS_JOIN_MITER };
     static int  cxClient, cyClient;
     HDC         hdc;
     int         i;
     LOGBRUSH    lb;
     PAINTSTRUCT ps;

     switch(iMsg)
          {
          case WM_SIZE:
               cxClient = LOWORD(lParam);
               cyClient = HIWORD(lParam);
               return 0;
          case WM_PAINT:
               hdc = BeginPaint(hwnd, &ps);

               SetMapMode(hdc, MM_ANISOTROPIC);
               SetWindowExtEx(hdc, 100, 100, NULL);
                   SetViewportExtEx(hdc, cxClient, cyClient, NULL);

               lb.lbStyle = BS_SOLID;
               lb.lbColor = RGB(128, 128, 128);
               lb.lbHatch = 0;


               for(i = 0; i < 3; i++)
                    {
                    SelectObject(hdc,
                         ExtCreatePen(PS_SOLID | PS_GEOMETRIC |
                                      iEnd [i] | iJoin [i], 10,
                                      &lb, 0, NULL));

                    BeginPath(hdc);

                    MoveToEx(hdc, 10 + 30 * i, 25, NULL);
                    LineTo  (hdc, 20 + 30 * i, 75);
                    LineTo  (hdc, 30 + 30 * i, 25);


                    EndPath(hdc);

                    StrokePath(hdc);

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

```
                MoveToEx(hdc, 10 + 30 * i, 25, NULL);
                LineTo  (hdc, 20 + 30 * i, 75);
                LineTo  (hdc, 30 + 30 * i, 25);
                }

           EndPaint(hwnd, &ps);
           return 0;

           case WM_DESTROY:
           PostQuitMessage(0);
           return 0;
      }

  return DefWindowProc(hwnd, iMsg, wParam, lParam);
  }
```

Рис. 4.24 Программа ENDJOIN

Программа рисует три фигуры типа буквы V широкой линией, используя стили концов и мест соединения в порядке, указанном выше. Программа также выводит три одинаковых линии, используя стандартное черное перо. Это сделано для того, чтобы показать, насколько широкие линии отличаются от обычных тонких линий. Результат работы программы приведен на рис. 4.25.

Рис. 4.25 Вид экрана программы ENDJOIN.

Есть надежда, что теперь вам стало понятно, почему Windows 95 поддерживает функцию *StrokePath*: Когда вы рисуете две линии отдельно, GDI рисует концы каждой из них. Если же они входят в путь, определенный в GDI, то линии рисуются с местом их соединения.

Bits and Blts

Битовые или растровые образы (bitmap) представляют собой один из двух методов хранения графической информации в программах для Windows 95. Битовый образ — это цифровое представление изображения. Каждый пиксель соответствует одному или более битам в растровом образе. Монохромные битовые образы требуют всего один бит для хранения информации об одном пикселе; цветные битовые образы требуют дополнительных битов для представления цвета каждого пикселя. Второй формой хранения графической информации является метафайл (metafile), его мы рассмотрим в этой главе позднее. Метафайл — это описание изображения, а не его цифровое представление.

Битовые образы и метафайлы занимают свое определенное место в компьютерной графике. Битовые образы чаще всего используются для хранения очень сложных изображений реального мира, таких как цифровые фотографии или видеоролики. Метафайлы более удобны для хранения изображений, выполненных человеком или компьютером, таких как архитектурные чертежи. И битовые образы и метафайлы могут находиться в памяти, могут быть сохранены на диске в виде файлов, могут передаваться между приложениями Windows через буфер обмена (clipboard).

Вы можете строить битовые образы вручную, используя программу Paint из официальной версии Windows 95. Затем вы включаете их в качестве ресурсов в файл описания ресурсов и загружаете в программу, используя

функцию *LoadBitmap*, как показано в главе 9. В главе 10 мы увидим, как битовые образы могут использоваться вместо текста в меню. Они могут также использоваться для создания кистей.

Битовые образы имеют два существенных недостатка. Во-первых, они очень зависимы от оборудования. Большинство реальных устройств — цветные. Представление цветного битового образа на монохромном устройстве часто неудовлетворительно. Другая проблема состоит в том, что битовые образы часто рассчитаны на конкретное разрешение и коэффициент сжатия изображения. Хотя битовые образы могут быть и растянуты и сжаты, это требует, соответственно, либо дублирования, либо перемещения строк или столбцов пикселей, и может привести к искажениям изображения. Метафайл может быть преобразован практически к любому размеру без искажений.

Вторым существенным недостатком битовых образов является то, что для их хранения требуется много памяти. Например, битовый образ, представляющий экран 16-цветного дисплея VGA в режиме 640x480 пикселей, требует 150 килобайт. Метафайлы обычно требуют значительно меньше памяти, чем битовые образы. Размер памяти для битового образа зависит от размера изображения и числа цветов, в нем содержащихся. Размер памяти для метафайла зависит от сложности изображения и конкретного числа инструкций GDI для его воспроизведения.

Единственное преимущество битовых образов над метафайлами — это, конечно, скорость. Копирование битовых образов на экран обычно осуществляется значительно быстрее, чем воспроизведение метафайла.

Цвета и битовые образы

Каждый пиксель изображения соответствует одному или более битам битового образа. Для представления монохромного изображения требуется один бит на пиксель. Для представления цветного изображения требуется более одного бита на пиксель. Число цветов, которые могут быть представлены в битовом образе равно 2 в степени "число битов на пиксель". Например, для представления 16 цветов в битовом образе требуется 4 бита на пиксель, для представления 256 цветов — 8 битов на пиксель. Для полноцветного битового образа необходимо 24 бита на пиксель, по 8 битов для каждого из цветов RGB — красного, зеленого, синего.

До появления версии Windows 3.0 объектами GDI были только битовые образы, поддерживаемые Windows. Работа с ними осуществлялась через описатель битового образа. Эти битовые образы были либо монохромными, либо имели такую же цветовую структуру, как и реальные графические устройства вывода, например, видеотерминал. Битовый образ, совместимый с 16-цветным VGA, имел, соответственно, четыре цветовых плоскости. Проблема заключалась в том, что эти цветные битовые образы не могли быть сохранены и использованы на графических устройствах вывода, имеющих иную цветовую организацию, например, на устройстве, имеющем 8 бит на пиксель, и способном воспроизвести 256 цветов.

Начиная с Windows 3.0, был введен новый формат битовых образов, названный независимым от устройства битовым образом (device independent bitmap) или DIB. В DIB содержалась таблица цветов, отражавшая соответствие двоичного представления пикселей цветам RGB. DIB могут быть выведены на любом растровом графическом устройстве. Проблема состоит только в том, что цвета из DIB должны быть преобразованы к ближайшим цветам, которые реально может воспроизвести устройство.

Битовые образы, не зависящие от устройства (DIB)

Формат DIB называют независящим от устройства потому, что он содержит таблицу цветов. Таблица цветов описывает то, как значения пикселей преобразуются в значения RGB цветов. Эта таблица цветов не обязательно может быть совместимой с конкретным графическим устройством вывода. Формат DIB — это расширенный формат битового образа, поддерживаемого в OS/2 1.1 Presentation Manager. Заголовочные файлы Windows содержат некоторые структуры для работы с битовыми образами OS/2.

После введения DIB битовые образы — объекты GDI, стали иногда именоваться "зависимыми от устройства" битовыми образами (device dependent bitmap) (DDBs). Они зависят от устройства потому, что они должны быть совместимы с конкретным устройством графического вывода. DIB не является объектом GDI. GDI не может хранить DIB. Поддержку DIB в блоке памяти должна осуществлять ваша программа. Если DIB становится объектом GDI, он сразу преобразуется в зависящий от устройства битовый образ, совместимый с реальным устройством вывода. DIB в основном применяются для обмена между программами. Они могут передаваться между программами путем записи в файл или путем копирования в буфер обмена.

DIB может быть преобразован в зависящий от устройства битовый образ и являющийся объектом GDI; в этом случае информация о цветах, независящая от устройства, теряется. Битовый образ — объект GDI может также использоваться для построения DIB. В этом случае DIB будет содержать таблицу цветов, совместимую с тем графическим устройством вывода, с которым совместим битовый образ — объект GDI.

Если вам необходимо сохранить информацию битового образа в файле или прочитать файл битового образа или передать информацию битового образа с помощью буфера обмена в формате, независящем от устройства, то нужен DIB. Однако, если вам необходимо только создать или использовать монохромные битовые образы, или,

если вам в вашей собственной программе нужны битовые образы, совместимые с дисплеем, то в этом случае проще использовать битовые образы как объекты GDI.

Файл DIB

Вы можете создать независящий от устройства битовый образ, и сохранить его в файле на диске, используя Microsoft Developer Studio или программу Paint, входящую в официальную версию Windows 95. Чаще всего эти файлы имеют расширение .BMP, хотя некоторые DIB могут храниться в файлах с расширением .DIB.

Файл DIB начинается с секции заголовка, определенной структурой BITMAPFILEHEADER. Эта структура имеет пять полей:

Поле	Размер	Описание
bfType	WORD	Байты"BM" для битовых образов
bfSize	DWORD	Общий размер файла
bfReserved1	WORD	Установлено в 0
bfReserved2	WORD	Установлено в 0
bfOffBits	DWORD	Смещение битов битового образа от начала файла

За этой информацией следует другой заголовок, определенный структурой BITMAPINFOHEADER. Структура имеет 11 полей:

Поле	Размер	Описание
biSize	DWORD	Размер структуры в байтах
biWidth	LONG	Ширина битового образа в пикселях
biHeight	LONG	Высота битового образа в пикселях
biPlanes	WORD	Установлено в 1
biBitCount	WORD	Число битов цвета на пиксель (1, 4, 8, 24)
biCompression	DWORD	Схема компрессии (если нет — 0)
biSizeImage	DWORD	Размер битов битового образа в байтах (нужен только при компрессии)
biXPelsPerMeter	LONG	Разрешение в пикселях на метр по горизонтали
biYPelsPerMeter	LONG	Разрешение в пикселях на метр по вертикали
biClrUsed	DWORD	Число цветов, используемых в изображении
biClrImportant	DWORD	Число важных цветов в изображении

Все поля, следующие за полем *biBitCount*, могут быть по умолчанию установлены в 0 (или их может вообще не быть в файле). В этом случае длина структуры будет равна 16 байтам. Кроме описанных выше полей, она может также содержать дополнительные поля.

Если *biClrUsed* установлено в 0 и число битов цвета на пиксель равно 1, 4 или 8, то за структурой BITMAPINFOHEADER следует таблица цветов, состоящая из двух или более структур RGBQUAD. Структура RGBQUAD определяет значение RGB цвета:

Поле	Размер	Описание
rgbBlue	BYTE	Интенсивность голубого
rgbGreen	BYTE	Интенсивность зеленого
rgbRed	BYTE	Интенсивность красного
rgbReserved	BYTE	Равно 0

Число структур RGBQUAD обычно определяется значением поля *biBitCount* : 2 структуры RGBQUAD при 1 цветовом бите, 16 при 4 цветовых битах, 256 при 8 битах цвета. Однако, если значение в поле *biClrUsed* не равно нулю, то в нем содержится число структур RGBQUAD, входящих в таблицу цветов.

За таблицей цветов следует массив битов, определяющих битовый образ. Этот массив начинается с нижней строки пикселей. Каждая строка начинается с самого левого пикселя. Каждый пиксель представлен 1, 4, 8 или 256 битами. Для монохромных битовых образов с 1 битом цвета на пиксель первый пиксель в каждой строке представляется наиболее значащим битом первого байта в каждой строке. Если этот бит равен 0, то цвет пикселя определяется из первой структуры RGBQUAD таблицы цветов. Если он равен 1, то цвет пикселя определяется из второй структуры RGBQUAD таблицы цветов.

В случае 16-цветного битового образа с 4 битами на пиксель первый пиксель каждой строки представляется четырьмя самыми значащими битами первого байта в каждой строке. Цвет каждого пикселя определяется путем использования этого 4-х битного значения как индекса для любого из 16 входов таблицы цветов.

В случае 256-цветного битового образа каждый байт соответствует одному пикселю. Цвет каждого пикселя определяется путем использования этого 8-ми битного значения как индекса для любого из 256 входов таблицы цветов.

Если битовый образ содержит 24 бита для представления цвета одного пикселя, то каждый набор из 3-х байтов — это RGB-цвет пикселя. Таблица цветов отсутствует, если значение поля *biClrUsed* структуры BITMAPINFOHEADER не равно 0.

В любом случае, каждая строка данных битового образа имеет размер, кратный 4 байтам. Для удовлетворения этого требования, если необходимо, строка расширяется вправо.

Формат битового образа, поддерживаемый в OS/2 1.1 и более поздних версиях — очень похож. Он начинается с структуры BITMAPFILEHEADER, а затем следует структура BITMAPCOREHEADER, имеющая размер 12 байтов. (Вы можете определить, когда файл битового образа имеет этот формат или формат Windows, проверив первое поле этой структуры.) Таблица цветов состоит из структур типа RGBTRIPLE, а не структур типа RGBQUAD.

Начиная с Windows 95, определен третий информационный заголовок, называемый BITMAPV4HEADER. (Windows 95 также известна как Windows 4.0, следовательно V4 означает "version 4".) Он содержит некоторую дополнительную информацию для правильного воспроизведения цветов на различных устройствах.

Упакованный формат хранения DIB

Если имеется файл DIB, который нужно прочитать в Windows-программе, вы можете считать его непосредственно в блок выделенной памяти. Этот блок известен как "Упакованный формат хранения DIB" (The packed-DIB Memory format). Он содержит все компоненты файла DIB, кроме структуры BITMAPFILEHEADER. Таким образом, этот блок памяти начинается со структуры информационного заголовка, затем следует таблица цветов (если она существует), затем непосредственно биты битового образа. Упакованный формат хранения DIB используется для копирования DIB в/из буфера обмена.

Вы можете также использовать упакованный формат хранения DIB для отображения битовых образов на экране, используя функции *SetDIBitsToDevice* или *StretchDIBits*, для создания кисти на базе DIB (*CreateDIBPatternBrush*) или для создания зависящего от устройства GDI битового образа (device dependent GDI bitmap) из DIB (*CreateDIBitmap*).

В любом случае вам следует вычислить адрес в глобальном блоке памяти, с которого начинаются биты битового образа. Вы можете это сделать, проверив поля структуры информационного заголовка и определив размер таблицы цветов.

Отображение DIB

Windows имеет две функции для отображения DIB из блока памяти, хранящего упакованный формат DIB. Основная функция — это *StretchDIBits*, которая позволяет сжимать и растягивать битовый образ, и которая может реализовывать различные растровые операции, описываемые далее в этой главе. Функция *SetDIBitsToDevice* немного проще, поскольку она отображает битовый образ без растяжения или сжатия и не реализует растровые операции.

Обеим функциям требуется указатель на структуру BITMAPINFOHEADER — начало блока памяти DIB, и указатель на биты битового образа. Функция *StretchDIBits* обладает большими возможностями — позволяет вам отображать на экране любой прямоугольник с битовым образом с заданными логическими шириной и высотой.

Преобразование DIB в объекты "битовые образы"

Если у вас 16-ти или 256-цветный дисплей, и вы хотите отобразить на нем полноцветный, 24 бита на пиксель, битовый образ, то вы заметите, что требуется некоторое время для его отображения. Это происходит потому, что драйвер устройства должен выполнить поиск ближайшего цвета для каждого пикселя битового образа.

Вы можете увеличить скорость отображения, преобразовав DIB в зависящий от устройства GDI битовый образ, используя функцию *CreateDIBitmap*. Поиск ближайшего цвета будет выполнен только один раз, а затем объект — битовый образ получит формат, соответствующий формату дисплея.

Несмотря на имя, функция *CreateDIBitmap* не строит DIB. Она создает зависящий от устройства объект GDI — битовый образ из описания DIB и возвращает описатель этого объекта. Этот битовый образ GDI совместим с устройством графического вывода, описатель которого передается функции в качестве первого параметра. Как и при отображении DIB, GDI должен преобразовать цвета, независящие от устройства, в цвета конкретного устройства.

Вслед за этим вызовом вы можете отображать битовый образ, выбирая его в контекст памяти (memory device context), и используя функцию *BitBlt*, как показано ниже в этой главе.

Вы можете также использовать функцию *CreateDIBitmap* для создания неинициализированного битового образа — объекта GDI:

```
hBitmap = CreateDIBitmap(hdc, &bmih, 0, NULL, NULL, 0);
```

Существуют также две функции для установки и чтения битов битового образа. Первая функция устанавливает биты:

```
SetDIBits(hdc, hBitmap, iStart, iNum, pBits, &bmi, iUsage);
```

Последние три параметра такие же, как у функции *CreateDIBitmap*. Параметр *iStart* определяет начальную скан-линию, адресуемую *pBits*. Он лежит в интервале от 0 (для нижней скан-линии) до высоты битового образа в пикселях — 1 (для верхней скан-линии). Параметр *iNum* задает число скан-линий, устанавливаемых в битовом образе.

Функция *GetDIBits* имеет такие же параметры:

```
GetDIBits(hdc, hBitmap, iStart, iNum, pBits, &bmi, iUsage);
```

В этом случае *pBits* указывает на буфер для записи битов битового образа. Функция устанавливает поля структуры BITMAPINFO для того, чтобы можно было определить размеры битового образа и таблицы цветов.

Битовый образ — объект GDI

Формат битового образа, предложенный в Windows 1.0, очень ограничен и почти полностью зависит от устройства вывода, для которого он создан. Для хранения файлов битовых образов на диске вам следует использовать формат DIB, а не устаревший формат битового образа. Однако, когда вам необходим битовый образ исключительно для использования в вашей программе, работа с битовым образом, зависящим от устройства, окажется значительно проще и производительней.

Создание битовых образов в программе

Windows содержит пять функций, которые позволяют вам в программе создать зависящий от устройства битовый образ — объект GDI. Первая функция *CreateDIBitmap* рассматривалась выше. Вот другие функции:

```
hBitmap = CreateBitmap(cxWidth, cyHeight, iPlanes, iBitsPixel, pBits);
hBitmap = CreateBitmapIndirect(&bitmap);
hBitmap = CreateCompatibleBitmap(hdc, cxWidth, cyHeight);
hBitmap = CreateDiscardableBitmap(hdc, cxWidth, cyHeight);
```

Во всех случаях параметры *cxWidth* и *cyHeight* — это ширина и высота битового образа в пикселях. В функции *CreateBitmap* параметры *iPlanes* и *iBitsPixel* — это число цветовых плоскостей и число битов цвета на пиксель в битовом образе. Хотя бы один из этих двух параметров должен быть равен 1. Если оба параметра равны 1, то функция строит монохромный битовый образ. (Мы вскоре остановимся на том, как цветовые плоскости и биты цвета представляют цвет.)

В функции *CreateBitmap* параметр *pBits* может быть установлен в NULL, если вы создаете неинициализированный битовый образ. Созданный битовый образ будет содержать случайные данные. В функциях *CreateCompatibleBitmap* и *CreateDiscardableBitmap* Windows использует контекст устройства, описываемый параметром *hdc*, для получения числа цветовых плоскостей и числа битов цвета на пиксель. Битовый образ, создаваемый этими функциями, будет неинициализированным.

Функция *CreateBitmapIndirect* схожа с функцией *CreateBitmap* за исключением того, что она использует структуру типа BITMAP для задания битового образа. Следующая таблица показывает поля этой структуры:

Поле	Тип	Описание
bmType	LONG	Установлено в 0
bmWidth	LONG	Ширина битового образа в пикселях
bmHeight	LONG	Высота битового образа в пикселях
bmWidthBytes	LONG	Ширина битового образа в байтах (должна быть четной)
bmPlanes	WORD	Число цветовых плоскостей
bmBitsPixel	WORD	Число битов цвета на пиксель
bmBits	LPVOID	Указатель на массив битов

Поле *bmWidthBytes* должно быть четным числом — минимальным четным числом байтов, необходимым для хранения одной скан-линии. Массив битов, на который указывает *bmBits*, должен быть организован на базе поля *bmWidthBytes*. Если *bm* — структура типа BITMAP, то вы можете вычислить значение поля *bmWidthBytes*, используя следующее выражение:

```
bm.bmWidthBytes =(bm.bmWidth * bm.bmBitsPixel + 15) / 16 * 2;
```

Если Windows не может создать битовый образ (в основном из-за недостатка памяти), она возвратит NULL. Вам следует проверять возвращаемые значения каждой из функций создания битовых образов, особенно, если вы строите большие битовые образы.

После создания битового образа вы уже не можете изменить размер, число цветовых плоскостей или число битов цвета на пиксель. Вам надо будет создать новый битовый образ и передать биты из исходного битового образа в новый. Если вы имеете описатель битового образа, вы можете узнать его размер и цветовую организацию следующим образом:

```
GetObject(hBitmap, sizeof(BITMAP),(LPVOID) &bitmap);
```

Эта функция копирует информацию о битовом образе в структуру (с именем bitmap) типа BITMAP. Эта функция не устанавливает поле *bmBits*. Для доступа к битам битового образа вам нужно вызвать функцию:

```
GetBitmapBits(hBitmap, dwCount, pBits);
```

Она копирует *dwCount* бит в символьный массив по указателю *pBits*. Для того, чтобы быть уверенными, что все биты битового образа скопируются в этот массив, вы можете вычислить параметр *dwCount* на основе значений полей структуры битового образа:

```
dwCount =(DWORD) bitmap.bmWidthBytes * bitmap.bmHeight * bitmap.bmPlanes;
```

Вы можете также задать Windows скопировать символьный массив, содержащий биты битового образа, обратно в существующий битовый образ, используя функцию:

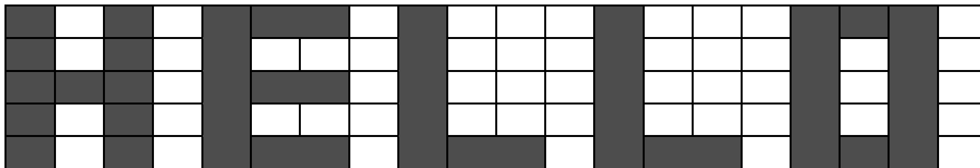
```
SetBitmapBits(hBitmap, dwCoint, pBits);
```

Поскольку битовые образы — это объекты GDI, вы должны удалить каждый созданный вами битовый образ:

```
DeleteObject(hBitmap);
```

Формат монохромного битового образа

Для монохромного битового образа формат битов относительно прост и может быть получен прямо из изображения, которое вы хотите создать. Например, предположим, вы хотите создать битовый образ, имеющий такой вид:



Вы можете записать ряды битов (0 для черного и 1 для белого), которые в точности соответствуют этой сетке. Читая эти биты слева направо, вы можете записать каждые восемь битов в виде байта в шестнадцатеричной системе счисления. Если значение ширины битового образа не кратно 16, то добавьте справа столько нулевых байтов, сколько необходимо для того, чтобы получить четное число байтов:

```
0 1 0 1 0 0 0 1 0 1 1 1 0 1 1 1 0 0 0 1 = 51 77 10 00
0 1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 0 1 = 57 77 50 00
0 0 0 1 0 0 1 1 0 1 1 1 0 1 1 1 0 1 0 1 = 13 77 50 00
0 1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 0 1 = 57 77 50 00
0 1 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 = 51 11 10 00
```

Ширина равна 20 пикселям, высота в скан-линиях равна 5 и ширина в байтах равна 4. Вы можете задать структуру BITMAP для этого битового образа следующим выражением:

```
static BITMAP bitmap = { 0, 20, 5, 4, 1, 1};
```

и сохранить биты в массиве типа BYTE:

```
static BYTE byBits [] = { 0x51, 0x77, 0x10, 0x00,
                          0x57, 0x77, 0x50, 0x00,
                          0x13, 0x77, 0x50, 0x00,
                          0x57, 0x77, 0x50, 0x00,
                          0x51, 0x11, 0x10, 0x00 };
```

Создание битового образа с помощью функции *CreateBitmapIndirect* осуществляется в два этапа:

```
bitmap.bmBits =(LPVOID) byBits;
```

```
hBitmap = CreateBitmapIndirect(&bitmap);
```

Другой вариант:

```
hBitmap = CreateBitmapIndirect(&bitmap);
SetBitmapBits(hBitmap, sizeof(byBits), byBits);
```

Вы можете также создать битовый образ в одной инструкции:

```
hBitmap = CreateBitmap(20, 5, 1, 1, byBits);
```

Формат цветного битового образа

Цветной битовый образ в предыдущих версиях Windows немного более сложен и очень сильно зависим от устройства. Цветной битовый образ организован так, чтобы облегчить вывод битов на конкретное физическое устройство. Битовый образ организован или как набор цветовых плоскостей или имеет набор цветовых битов на пиксель. Это зависит от устройства вывода, для которого предназначен битовый образ.

Давайте сначала рассмотрим битовый образ, у которого *bmBitsPixel* равно 1 (что означает, что в нем 1 бит цвета на пиксель), а *bmPlanes* больше 1. Цветной битовый образ для 16-цветного VGA — это хороший пример. Windows использует четыре цветовых плоскости VGA для представления 16 цветов, поэтому *bmPlanes* равно 4. Массив битов начинается с верхней скан-линии. Цветовые плоскости для каждой скан-линии хранятся последовательно: красная плоскость — первой, зеленая плоскость — второй, затем голубая плоскость и плоскость интенсивности. Затем следует вторая скан-линия битового образа.

Битовый образ может также представлять цвет как некоторое число битов на пиксель. Предположим, что видеомонитор может представить 256 цветов, используя 8 бит (1 байт) на пиксель. Для каждой скан-линии первый байт представляет цвет самого левого пикселя, второй байт представляет цвет следующего пикселя, и т. д. Значение *bmWidthBytes* структуры BITMAP отражает увеличенную ширину каждой скан-линии в байтах, а значение *bmWidth* — это число пикселей в скан-линии.

Здесь есть одна тонкость: битовый образ не содержит никакой информации о том, как эти цветовые плоскости или биты цвета соответствуют реальным цветам устройства вывода. Конкретный битовый образ предназначен только для устройства, имеющего такую же организацию, как и битовый образ. Поэтому не рекомендуется использовать функции *CreateBitmap* или *CreateBitmapIndirect* для создания инициализированного цветного битового образа. Вам следует применять эти функции только для создания инициализированных или неинициализированных монохромных битовых образов.

Для создания цветного битового образа используйте функцию *CreateCompatibleBitmap*, которая гарантирует, что формат будет совместим с реальным графическим устройством отображения. Вы задаете изображение цветного битового образа, выбирая его в контекст памяти (рассматривается ниже) и, затем, рисуете в этом контексте или используете функцию *BitBlt*.

При создании цветного битового образа, для которого совместимость с реальным графическим устройством вывода необязательна, следует использовать DIB.

Контекст памяти

Две функции — *SetDIBitsToDevice* и *StretchDIBits* позволяют вам воспроизвести массив битов на устройстве вывода. Тем не менее, даже если у вас есть описатель битового образа, то нет функции для рисования битового образа на поверхности отображения контекста устройства. Вы будете тщетно искать функцию, имеющую такой вид:

```
DrawBitmap(hdc, hBitmap, xStart, yStart); // Такой функции нет !!!
```

Эта функция копировала бы битовый образ в контекст устройства, заданный параметром *hdc*, начиная с логической точки (*xStart, yStart*). Мы напишем свою собственную функцию *DrawBitmap* позднее в этой главе. Однако для этого вам необходимо познакомиться с некоторыми концепциями, начиная с контекста памяти.

Контекст памяти (memory device context) — это контекст, имеющий поверхность отображения (display surface), существующую только в памяти. Вы можете создать контекст памяти, используя функцию *CreateCompatibleDC*:

```
hdcMem = CreateCompatibleDC(hdc);
```

Описатель *hdc* — описатель действительного открытого контекста устройства. Функция *CreateCompatibleDC* возвращает описатель контекста памяти. При создании контекста памяти все атрибуты устанавливаются в значения по умолчанию. Вы можете делать почти все, что вы захотите с этим контекстом памяти. Вы можете устанавливать атрибуты в значения, отличные от значений по умолчанию, получать текущие значения атрибутов, выбирать в него перья, кисти и регионы. И, конечно, вы можете даже рисовать на нем. Но это не имеет смысла делать прямо сейчас. И вот почему.

Когда вы впервые создаете контекст памяти, он имеет поверхность отображения, содержащую только 1 монохромный пиксель. Это очень маленькая поверхность отображения. (Не полагайтесь на то, что функция *GetDeviceCaps* сообщит вам это. Все значения HORZSIZE, VERTSIZE, HORZRES, VERTRES, BITSPIXEL и PLANES для *hdcMem* будут установлены в значения, связанные с исходным *hdc*. Если бы функция *GetDeviceCaps* в действительности возвращала истинные значения, связанные с контекстом памяти, когда он был впервые создан, то для индексов HORZRES, VERTRES, BITSPIXEL и PLANES эти значения были бы равны 1.) Вам надо

увеличить поверхность отображения контекста памяти. Это осуществляется выбором битового образа в контекст памяти:

```
SelectObject(hdcMem, hBitmap);
```

Теперь поверхность отображения *hdcMem* имеет те же ширину, высоту и организацию цвета, что и битовый образ, описываемый *hBitmap*. Если начало координат окна и области вывода установлены по умолчанию, то логическая точка (0, 0) контекста памяти соответствует левому верхнему углу битового образа.

Если битовый образ уже содержит некоторое изображение, то это изображение становится частью поверхности отображения контекста памяти. Любые изменения битового образа (например, с использованием функции *SetBitmapBits* для установки другого массива битов в битовом образе) отразятся на поверхности отображения. Все, что вы рисуете в контексте памяти, немедленно отражается в битовом образе. Короче, битовый образ — это поверхность отображения контекста памяти.

Ранее были рассмотрены различные функции для создания битовых образов. Вот одна из них:

```
hBitmap = CreateCompatibleBitmap(hdc, xWidth, yHeight);
```

Если *hdc* — это обычный описатель контекста устройства для дисплея или принтера, то число цветовых плоскостей и число битов на пиксель созданного битового образа совпадает с соответствующими характеристиками устройства. Однако, если *hdc* — это описатель контекста памяти (и в него еще не выбран битовый образ), то функция *CreateCompatibleBitmap* возвращает монохромный битовый образ шириной *xWidth* и высотой *yHeight* пикселей.

Битовый образ — объект GDI. Ранее в этой главе было показано, как использовать функцию *SelectObject* для выбора перьев, кистей или регионов в контекст устройства, а далее мы узнаем, как применять эту функцию для выбора шрифта в контекст устройства. С помощью функции *SelectObject* можно выбирать эти четыре объекта GDI в контекст памяти. Однако, вы не можете выбирать битовый образ в обычный контекст устройства — только в контекст памяти.

Когда закончится работа с контекстом памяти, его надо удалить:

```
DeleteDC(hdcMem);
```

Итак, вы можете сказать: "Очень хорошо. Но мы еще не решили проблему отображения битового образа на экране. Все что мы сделали — научились выбирать его в контекст памяти. Что теперь?" Теперь мы должны научиться как переносить биты из одного контекста устройства в другой с помощью функции *BitBlt*.

Мощная функция *BitBlt*

Компьютерная графика включает в себя процедуру записи пикселей на устройство отображения. Ранее мы уже рассматривали некоторые пути выполнения этой задачи, но для больших и сложных манипуляций с пикселями в Windows есть только функции *BitBlt, PatBlt* и *StretchBlt*. *BitBlt* означает перенос блоков битов (bit block transfer). Функция *BitBlt* переносит пиксели, другими словами, это — универсальная растровая функция (raster blaster). Термин "transfer" не совсем справедлив по отношению к функции *BitBlt*. Она делает больше, чем просто перенос пикселей, — она осуществляет одну из 256 логических растровых операций над тремя наборами пикселей.

Функция *PatBlt*

PatBlt (pattern block transfer) — это простейшая из трех blt-функций. Она существенно отличается от функций *BitBlt* и *StretchBlt* тем, что использует только один контекст устройства. Но для начала функция *PatBlt* подходит вполне.

Раньше мы уже встречались с атрибутом контекста устройства, называемым режимом рисования. Этот атрибут может быть установлен в одно из 16 значений, соответствующих бинарной растровой операции (ROP2). Когда вы рисуете линию, режим рисования определяет тип логической операции, которую Windows реализует над пикселями пера и пикселями приемного контекста устройства. Функция *PatBlt* похожа на функции рисования линий с тем исключением, что она изменяет содержимое прямоугольной области приемного контекста устройства, а не только линии. Она выполняет логическую операцию с пикселями в этом прямоугольнике и в шаблоне (pattern). Шаблон — это просто другое название кисти (brush). Поэтому функция *PatBlt* использует кисть, выбранную в данный момент в контексте устройства.

Синтаксис вызова функции *PatBlt* таков:

```
PatBlt(hdc, xDest, yDest, xWidth, yHeight, dwROP);
```

Параметры *xDest, yDest, xWidth, yHeight* задаются в логических координатах. Логическая точка (*xDest, yDest*) задает левый верхний угол прямоугольника. Он имеет ширину *xWidth* и высоту *yHeight* единиц. (Смотри следующий раздел, озаглавленный "Координаты *Blt*", для более подробного определения этих величин.) Это и есть

та прямоугольная область, которую изменяет функция *PatBlt*. Логическая операция, которую выполняет функция *PatBlt* над кистью и приемным контекстом устройства, определяется параметром *dwROP*, представляющим собой двойное слово (32-битное целое) ROP кода. Этот код не имеет отношения ни к одному из кодов ROP2, используемых в режиме рисования.

В Windows существует 256 ROP2 кодов. Они определяют всевозможные логические комбинации исходной (source) области отображения, приемной (destination) области отображения и шаблона (или кисти). Драйвер устройства для видеомонитора поддерживает все 256 растровых операций, посредством использования "компилятора" (compiler) типов. Этот компилятор использует 32-разрядный ROP код для генерации последовательности машинных инструкций, реализующих эту логическую операцию над пикселями дисплея, а затем выполняет эти инструкции. Старшее слово 32-разрядного ROP кода — число от 0 до 255. Младшее слово — число, которое помогает компилятору драйвера устройства в генерации машинных кодов для этой логической операции. Пятнадцать из этих 256 кодов имеют имена.

Поскольку функция *PatBlt* использует только приемный контекст устройства и шаблон (и не использует исходный контекст устройства), она может реализовать только подмножество этих 256 ROP кодов — 16 ROP кодов, использующих только приемный контекст устройства и шаблон. Поддерживаемые функцией *PatBlt* растровые операции приведены ниже в таблице. Обратите внимание, что эта таблица очень похожа на таблицу ROP2 кодов.

Шаблон (Pattern (P)) Приемник (Destination (D))	1 1	1 0	0 1	0 0	Булева операция (Boolean operation)	ROP код	Имя
Результаты: (Results)	0	0	0	0	0	0x000042	BLACKNESS
	0	0	0	1	~(P | D)	0x0500A9	
	0	0	1	0	~P & D	0x0A0329	
	0	0	1	1	~P	0x0F0001	
	0	1	0	0	P & ~D	0x500325	
	0	1	0	1	~D	0x550009	DSTINVERT
	0	1	1	0	P ^ D	0x5A0049	PATINVERT
	0	1	1	1	~(P & D)	0x5F00E9	
	1	0	0	0	P & D	0xA000C9	
	1	0	0	1	~(P ^ D)	0xA50065	
	1	0	1	0	D	0xAA0029	
	1	0	1	1	~P | D	0xAF0229	
	1	1	0	0	P	0xF00021	PATCOPY
	1	1	0	1	P | ~D	0xF50225	
	1	1	1	0	P | D	0xFA0089	
	1	1	1	1	1	0xFF0062	WHITENESS

Для монохромного контекста устройства бит равный 1 соответствует белому пикселю, а бит равный 0 — черному пикселю. Целиком черный или целиком белый приемник и шаблон — наиболее простой пример для начала рассмотрения работы функции *PatBlt*. Например, если вы вызываете:

```
PatBlt(hdc, xDest, yDest, xWidth, yHeight, 0x5F00E9L);
```

то прямоугольная область с вершиной в логической точке (*xDest, yDest*) и имеющая ширину *xWidth* пикселей и высоту *yHeight* пикселей, будет закрашена черным цветом, только если приемник был белым и в контекст устройства была выбрана кисть WHITE_BRUSH. В противном случае приемник будет закрашен белым. Конечно, даже в монохромном контексте устройства приемник и кисть могут быть полутоновыми комбинациями черных и белых пикселей. В этом случае Windows выполняет логическую операцию по принципу "pixel by pixel", что может привести к некоторым странным результатам. Например, если приемник был закрашен кистью GRAY_BRUSH, и она является текущей выбранной в контексте устройства, то:

```
PatBlt(hdc, xDest, yDest, xWidth, yHeight, PATINVERT);
```

установит приемник в чисто белый или в чисто черный цвет, в зависимости от того, как пиксели приемника совпадут с пикселями полутоновой кисти.

Цвет добавляет больше сложностей. Windows осуществляет отдельную логическую операцию для каждой цветовой плоскости или для каждого набора битов цвета, в зависимости от того, как организована память устройства.

Некоторые из часто употребляемых случаев использования функции *PatBlt* приведены ниже. Если вам необходимо нарисовать черный прямоугольник, вы вызываете:

```
PatBlt(hdc, xDest, yDest, xWidth, yHeight, BLACKNESS);
```

Если вам необходимо нарисовать белый прямоугольник, вы вызываете:

```
PatBlt(hdc, xDest, yDest, xWidth, yHeight, WHITENESS);
```

Функция:

```
PatBlt(hdc, xDest, yDest, xWidth, yHeight, DSTINVERT);
```

всегда инвертирует цвет прямоугольника. Если кисть WHITE_BRUSH выбрана в контексте устройства, то функция:

```
PatBlt(hdc, xDest, yDest, xWidth, yHeight, PATINVERT);
```

также инвертирует прямоугольник.

Вспомните функцию *FillRect*, закрашивающую кистью прямоугольную область:

```
FillRect(hdc, &rect, hBrush);
```

Следующий код является эквивалентным функции *FillRect*:

```
hBrush = SelectObject(hdc, hBrush);
PatBlt(hdc, rect,left, rect.top,
       rect.right — rect.left,
       rect.bottom — rect.top, PATCOPY);
SelectObject(hdc, hBrush);
```

Фактически, это код, используемый Windows для реализации функции *FillRect*. Когда вы вызываете функцию:

```
InvertRect(hdc, &rect);
```

Windows транслирует ее в функцию:

```
PatBlt(hdc, rect,left, rect.top,
       rect.right — rect.left,
       rect.bottom — rect.top, DSTINVERT);
```

Координаты *Blt*

Когда описывался синтаксис функции *PatBlt*, упомянулось, что точка (*xDest, yDest*) задает верхний левый угол прямоугольника, и этот прямоугольник имеет ширину *xWidth* и высоту *yHeight* единиц. Это не совсем корректно. Только в функциях GDI *BitBlt*, *PatBlt* и *StretchBlt* логические координаты прямоугольника задаются в терминах логической ширины и высоты относительно одной вершины. Все другие функции GDI для рисования, использующие ограничивающий прямоугольник, требуют задания координат в терминах левого верхнего и правого нижнего углов. В режиме отображения MM_TEXT указанное описание параметров функции *PatBlt* верно. Однако, для метрических режимов отображения — не верно. Если вы используете положительные значения *xWidth* и *yHeight*, то точка с координатами (*xDest, yDest*) должна быть левым нижним углом прямоугольника. Если вы хотите, чтобы точка (*xDest, yDest*) была левым верхним углом прямоугольника, то параметр *yHeight* должен быть установлен равным высоте прямоугольника, взятой со знаком минус.

Более точно, прямоугольник, с которым работает функция *PatBlt*, имеет логическую ширину, задаваемую абсолютным значением *xWidth* и логическую высоту, задаваемую абсолютным значением *yHeight*. Эти два параметра могут быть отрицательными. Прямоугольник определяется двумя углами, имеющими логические координаты (*xDest, yDest*) и (*xDest* + *xWidth*, *yDest* + *yHeight*). Верхний левый угол прямоугольника всегда включается в область, изменяемую функцией *PatBlt*. Правый нижний угол — всегда за ее пределами. В зависимости от режима отображения и знаков параметров *xWidth* и *yHeight* левым верхним углом прямоугольника может быть точка:

```
(xDest, yDest)
```

или

```
(xDest, yDest + yHeight)
```

или

```
(xDest + xWidth, yDest)
```

или

```
(xDest + xWidth, yDest + yHeight)
```

Если вы установите режим отображения MM_LOENGLISH и захотите использовать функцию *PatBlt*, изменяющую зону квадратного дюйма в левом верхнем углу рабочей области, вы можете использовать:

```
PatBlt(hdc, 0, 0, 100, -100, dwROP);
```

или

```
PatBlt(hdc, 0, -100, 100, 100, dwROP);
```

или

```
PatBlt(hdc, 100, 0, -100, -100, dwROP);
```

или

```
PatBlt(hdc, 100, -100, -100, 100, dwROP);
```

Простейший путь задать правильные параметры функции *PatBlt* — это установить *xDest* и *yDest* в левый верхний угол прямоугольника. Если ваш режим отображения определяет координату *y* так, что она возрастает при движении вверх, то используйте отрицательную величину параметра *yHeight*. Если ваш режим отображения определяет координату *x* так, что она возрастает при движении влево (что почти не встречается), то используйте отрицательную величину параметра *xWidth*.

Перенос битов с помощью функции *BitBlt*

В некотором смысле функция *BitBlt* — это расширенная функция *PatBlt*. Она делает все то же, что и *PatBlt*, а также вовлекает второй контекст устройства в логическую операцию. Ниже приведен синтаксис функции:

```
BitBlt(hdcDest, xDest, yDest, xWidth, yHeight, hdcSrc, xSrc, ySrc, dwROP);
```

Вызов функции *BitBlt* модифицирует приемный контекст устройства (его описатель *hdcDst*) в рамках прямоугольника, заданного логической точкой (*xDesr, yDest*) и параметрами *xWidth* и *yHeight*, заданными в логических единицах. Эти параметры определяют прямоугольник в соответствии с тем, как описано в предыдущем разделе. Функция *BitBlt* также использует прямоугольник из контекста устройства источника (описатель контекста *hdcSrc*). Этот прямоугольник начинается в логической точке (*xSrc, ySrc*) и имеет ширину *xWidth* логических единиц и высоту *yHeight* логических единиц.

Функция *BitBlt* осуществляет логическую операцию над тремя элементами: кистью, выбранной в контексте устройства приемника, пикселями прямоугольника в контексте устройства источника и пикселями прямоугольника в контексте устройства приемника. Результат заносится в прямоугольник приемного контекста устройства. Вы можете использовать любой из 256 ROP кодов в качестве параметра *dwROP* функции *BitBlt*. Пятнадцать ROP кодов, имеющих имена, приведены в следующей таблице.

Шаблон: (Pattern) (P) Источник: (Source) (S) Приемник: (Destination) (D)	1 1 1 1 0 0 0 0 1 1 0 0 1 1 0 0 1 0 1 0 1 0 1 0	Булева операция	ROP код	Имя
Результат: (Result)	0 0 0 0 0 0 0 0	0	0x000042	BLACKNESS
	0 0 0 1 0 0 0 1	~(S|D)	0x1100A6	NOTSRCERASE
	0 0 1 1 0 0 1 1	~S	0x330008	NOTSRCCOPY
	0 1 0 0 0 1 0 0	S&~D	0x440328	SRCERASE
	0 1 0 1 0 1 0 1	~D	0x550009	DSTINVERT
	0 1 0 1 1 0 1 0	P^D	0x5A0049	PATINVERT
	0 1 1 0 0 1 1 0	S^D	0x660046	SRCINVERT
	1 0 0 0 1 0 0 0	S&D	0x8800C6	SRCAND
	1 0 1 1 1 0 1 1	~S|D	0xBB0226	MERGEPAINT
	1 1 0 0 0 0 0 0	P&S	0xC000CA	MERGECOPY
	1 1 0 0 1 1 0 0	S	0xCC0020	SRCCOPY
	1 1 1 0 1 1 1 0	S|D	0xEE0086	SRCPAINT
	1 1 1 1 0 0 0 0	P	0xF00021	PATCOPY
	1 1 1 1 1 0 1 1	P|~S|D	0xFB0A09	PATPAINT
	1 1 1 1 1 1 1 1	1	0xFF0062	WHITENESS

Обратите внимание на ряды из восьми нулей и восьми единиц, которые являются результатами логических операций. Двузначное шестнадцатиричное число, соответствующее этим битам, есть старшее слово ROP кода. Если мы можем создать таблицу результатов для тех шаблонов, источников и приемников, которые нам нужны, то мы можем легко определить ROP код из таблицы ROP кодов в разделе "References" пакета Microsoft Developer Studio. Мы сделаем это позднее. Если вы используете один из 16-ти ROP кодов, приведенных в предыдущей таблице, то можно работать с функцией *PatBlt* вместо *BitBlt*, поскольку вы не обращаетесь к контексту устройства источника.

Вы можете сделать так, что *hdcSrc* и *hdcDst* будут описывать один и тот же контекст устройства. В этом случае функция *BitBlt* выполняет логическую операцию над приемным прямоугольником, исходным прямоугольником и текущей кистью, выбранной в контексте устройства. Однако, существует некоторая доля риска при выполнении этой операции с контекстом устройства рабочей области. Если часть исходного прямоугольника закрыта другим окном, то Windows будет использовать пиксели этого окна как исходные. Windows ничего не знает о том, что какое-либо окно закрывает часть рабочей области вашего окна.

Тем не менее, примеры функции *BitBlt*, использующей один и тот же контекст устройства для источника и приемника, просты для понимания.

Функция:

```
BitBlt(hdc, 100, 0, 50, 100, hdc, 0, 0, SRCCOPY);
```

копирует прямоугольник с вершиной в логической точке (0, 0), шириной 50 и высотой 100 логических единиц в прямоугольную область с вершиной в логической точке (100,0).

Функция *DrawBitmap*

Функция *BitBlt* наиболее эффективна при работе с битовыми образами, которые выбраны в контекст памяти. Когда вы выполняете перенос блока битов (bit block transfer) из контекста памяти в контекст устройства вашей рабочей области, битовый образ, выбранный в контексте памяти переносится в вашу рабочую область.

Ранее упоминалась гипотетическая функция *DrawBitmap*, которая выводила бы битовый образ на поверхность отображения. Такая функция должна иметь следующий синтаксис:

```
DrawBitmap(hdc, hBitmap, xStart, yStart);
```

Было обещано, что мы ее напишем. Вот она:

```
void DrawBitmap(HDC hdc, HBITMAP hBitmap, int xStart, int yStart)
{
	BITMAP bm;
	HDC    hdcMem;
	DWORD  dwSize;
	POINT  ptSize, ptOrg;
	hdcMem = CreateCompatibleDC(hdc);
	SelectObject(hdcMem, hBitmap);
	SetMapMode(hdcMem, GetMapMode(hdc));
	GetObject(hBitmap, sizeof(BITMAP),(LPVOID) &bm);
	ptSize.x = bm.bmWidth;
	ptSize.y = bm.bmHeight;
	DPtoLP(hdc, &ptSize, 1);
	ptOrg.x = 0;
	ptOrg.y = 0;
	DPtoLP(hdcMem, &ptOrg, 1);
	BitBlt(
		hdc, xStart, yStart, ptSize.x, ptSize.y,
		hdcMem, ptOrg.x, ptOrg.y, SRCCOPY
	);
	DeleteDC(hdcMem);
}
```

Здесь предполагается, что вы не хотите растягивать или сжимать высоту или ширину битового образа. Таким образом, если ваш битовый образ имеет ширину 100 пикселей, то вы сможете с его помощью закрыть любой прямоугольник, имеющий ширину 100 пикселей, независимо от режима отображения.

Функция *DrawBitmap* сначала создает контекст памяти, используя функцию *CreateCompatibleDC*, затем выбирает в него битовый образ с использованием функции *SelectObject*. Режим отображения контекста памяти устанавливается таким же, как режим отображения контекста устройства вывода. Поскольку функция *BitBlt* работает с логическими координатами и логическими размерами, и учитывая то, что вы не предполагаете растягивать или сжимать битовый образ, параметры *xWidth* и *yHeight* функции *BitBlt* должны иметь значения в логических координатах, соответствующих размерам битового образа в физических координатах. Поэтому, функция *DrawBitmap* определяет размеры битового образа, используя функцию *GetObject*, и создает структуру POINT для сохранения в ней ширины и высоты. Затем она преобразует эту точку в логические координаты. Аналогичные действия осуществляются и в отношении начала координат битового образа — точки (0, 0) в координатах устройства.

Обратите внимание, что не имеет никакого значения, какая кисть выбрана в приемном контексте устройства (*hdc*), поскольку режим SRCCOPY не использует кисть.

Использование других ROP кодов

SRCCOPY — самое часто встречающееся значение параметра *dwROP* функции *BitBlt*. Вам будет трудно найти примеры использования других 255 ROP кодов. Поэтому здесь будет показано несколько примеров, в которых используются другие ROP коды.

Первый пример: пусть у вас есть монохромный битовый образ, который вы хотите перенести на экран. При этом, вы хотите отобразить битовый образ так, чтобы черные (0) биты не оказывали влияния на текущее содержание

рабочей области. Более того, вы хотите, чтобы для всех белых (1) битов рабочая область закрашивалась кистью, возможно цветной, созданной функцией *CreateSolidBrush*. Как это сделать?

Предполагается, что вы работаете в режиме отображения MM_TEXT, и что вы хотите отобразить битовый образ, начиная в точке (*xStart, yStart*) вашей рабочей области. У вас также есть описатель монохромного битового образа (*hBitmap*) и описатель цветной кисти (*hBrush*). Вы также знаете ширину и высоту битового образа, и они хранятся в переменной *bm* структуры BITMAP. Вот код программы:

```
hdcMem = CreateCompatibleDC(hdc);
SelectObject(hdcMem, hBitmap);
hBrush = SelectObject(hdc, hBrush);
BitBlt(hdc, xStart, yStart, bm.bmWidth, bm.bmHeight, hdcMem, 0, 0, 0xE20746L);
SelectObject(hdc, hBrush);
DeleteDC(hdcMem);
```

Функция *BitBlt* выполняет логическую операцию над приемным контекстом устройства (*hdc*), исходным контекстом устройства (*hdcMem*) и кистью, выбранной в приемном контексте устройства. Вы создаете контекст памяти, выбираете в него битовый образ, выбираете цветную кисть в контекст устройства вашей рабочей области, и вызываете *BitBlt*. Затем вы выбираете исходную кисть в контекст устройства вашего дисплея и удаляете контекст памяти.

Осталось объяснить значение ROP кода 0xE20746 приведенного фрагмента программы. Этот код задает Windows выполнение следующей логической операции:

```
((Destination ^ Pattern) & Source) ^ Destination
```

Если опять непонятно, попробуйте разобраться в следующем:

Pattern:	1	1	1	1	0	0	0	0
Source:	1	1	0	0	1	1	0	0
Destination:	1	0	1	0	1	0	1	0
Result:	?	?	?	?	?	?	?	?

Для каждого черного бита битового образа (который будет выбран в исходный контекст памяти), вы хотите, чтобы приемный контекст устройства оставался неизменным. Это означает, что везде, где Source равен 0, вы хотите, чтобы Result равнялся Destination:

Pattern:	1	1	1	1	0	0	0	0
Source:	1	1	0	0	1	1	0	0
Destination:	1	0	1	0	1	0	1	0
Result:	?	?	1	0	?	?	1	0

Полдела сделано. Теперь для каждого белого бита битового образа вы хотите, чтобы приемный контекст закрашивался шаблоном. Кисть, выбранная вами в приемный контекст устройства — это шаблон. Таким образом, везде где Source равен 1, вы хотите, чтобы Result равнялся Pattern:

Pattern:	1	1	1	1	0	0	0	0
Source:	1	1	0	0	1	1	0	0
Destination:	1	0	1	0	1	0	1	0
Result:	1	1	1	0	0	0	1	0

Это означает, что старшее слово ROP кода равняется 0xE2. Вы можете заглянуть в таблицу ROP кодов пакета Microsoft Developer Studio и обнаружить, что полный ROP код равен 0xE20746.

Если обнаружится, что вы перепутали белые и черные биты при создании битового образа, то это легко исправить, используя другую логическую операцию:

Pattern:	1	1	1	1	0	0	0	0
Source:	1	1	0	0	1	1	0	0
Destination:	1	0	1	0	1	0	1	0
Result:	1	0	1	1	1	0	0	0

Теперь старшее слово ROP кода равно 0xB8, а весь ROP код равен 0xB8074A, что соответствует логической операции:

```
((Destination ^ Pattern) & Source) ^ Pattern
```

Теперь второй пример: вы можете заметить, что значки и курсоры состоят из двух битовых образов. Использование двух битовых образов позволяет этим объектам быть прозрачными или инвертировать цвет

закрываемых ими фрагментов экрана. Для монохромного значка или курсора, эти два битовых образа кодируются следующим образом:

Bitmap1:	0	0	1	1
Bitmap2:	0	1	0	1
Result:	Черный	Белый	Цвет экрана	Инверсный цвет экрана

Windows выбирает битовый образ Bitmap1 в контекст памяти и использует функцию *BitBlt* с ROP кодом SRCAND для переноса битового образа на экран. Этот ROP код соответствует логической операции:

```
Destination & Source
```

Она сохраняет неизменными биты приемника, соответствующие единичным битам Bitmap1, и устанавливает в 0 биты, соответствующие нулевым битам Bitmap1. Затем Windows выбирает Bitmap2 в контекст устройства и использует функцию *BitBlt* с параметром SRCINVERT. Логическая операция такова:

```
Destination ^ Source
```

Данная операция сохраняет неизменными биты приемника, соответствующие нулевым битам Bitmap2, и инвертирует биты, соответствующие единичным битам Bitmap2.

Взгляните на первый и второй столбцы таблицы: Bitmap1 и SRCAND делают биты черными, а Bitmap2 и SRCINVERT инвертируют выбранные биты в белый цвет. Эти операции устанавливают белые и черные биты, которые составляют значок и курсор. Теперь посмотрите на третий и четвертый столбцы таблицы: Bitmap1 и SRCAND сохраняют дисплей неизменным, а Bitmap2 и SRCINVERT инвертируют цвета указанных битов. Эти операции делают значки и курсоры прозрачными или позволяют инвертировать цвет закрываемой области экрана.

Другой пример творческого использования ROP кодов приводится далее в этой главе при описании функции *GrayString*.

Дополнительные сведения о контексте памяти

Мы использовали контекст памяти для передачи существующих битовых образов на экран. Вы можете также использовать контекст памяти для рисования на поверхности битового образа. Мы сделаем это в программе GRAFMENU в главе 10. Во-первых, вы строите контекст памяти:

```
hdcMem = CreateCompatibleDC(hdc);
```

Затем вы создаете битовый образ желаемого размера. Если вы хотите создать монохромный битовый образ, его можно сделать совместимым с *hdcMem*:

```
hBitmap = CreateCompatibleBitmap(hdcMem, xWidth, yHeight);
```

Для создания битового образа с такой же организацией цветов, как и у видеотерминала, сделайте битовый образ совместимым с *hdc*:

```
hBitmap = CreateCompatibleBitmap(hdc, xWidth, yHeight);
```

Теперь вы можете выбрать битовый образ в контекст памяти:

```
SelectObject(hdcMem, hBitmap);
```

А затем вы можете рисовать в этом контексте памяти (т. е. на поверхности битового образа), используя все функции GDI, рассмотренные в этой главе. Когда вы впервые создаете битовый образ, он содержит случайные биты. Поэтому есть смысл начать с использования функции *PatBlt* с ROP кодом WHITENESS или BLACKNESS для стирания фона контекста памяти.

Когда вы закончите рисование в контексте памяти, просто удалите его:

```
DeleteDC(hdcMem);
```

Теперь битовый образ будет содержать все, что вы нарисовали, пока он был выбран в контекст памяти.

Программа SCRAMBLE, показанная на рис. 4.26, очень "грубая", и не следовало бы показывать ее вам. Но она использует контекст памяти, как временный буфер для операций *BitBlt*, меняющих местами содержимое двух прямоугольных фрагментов экрана.

SCRAMBLE.MAK

```
#------------------------
# SCRAMBLE.MAK make file
#------------------------
scramble.exe : scramble.obj
```

```
     $(LINKER) $(GUIFLAGS) -OUT:scramble.EXE scramble.obj $(GUILIBS)

scramble.obj : scramble.c
     $(CC) $(CFLAGS) scramble.c
```

SCRAMBLE.C

```
/*-------------------------------------------------
   SCRAMBLE.C -- Scramble(and Unscramble) Screen
                 (c) Charles Petzold, 1996
  -------------------------------------------------*/

#include <windows.h>
#include <stdlib.h>

#define NUM 200

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static int iKeep [NUM][4];
     HDC        hdc, hdcMem;
     int        cx, cy;
     HBITMAP    hBitmap;
     int        i, j, x1, y1, x2, y2;

     if(LockWindowUpdate(GetDesktopWindow()))
          {
          hdc     = CreateDC("DISPLAY", NULL, NULL, NULL);
          hdcMem  = CreateCompatibleDC(hdc);
          cx  = GetSystemMetrics(SM_CXSCREEN) / 10;
          cy  = GetSystemMetrics(SM_CYSCREEN) / 10;
          hBitmap = CreateCompatibleBitmap(hdc, cx, cy);

          SelectObject(hdcMem, hBitmap);

          srand((int) GetCurrentTime());

          for(i = 0; i < 2; i++)
               for(j = 0; j < NUM; j++)
                    {
                    if(i == 0)
                         {
                         iKeep [j] [0] = x1 = cx *(rand() % 10);
                         iKeep [j] [1] = y1 = cy *(rand() % 10);
                         iKeep [j] [2] = x2 = cx *(rand() % 10);
                         iKeep [j] [3] = y2 = cy *(rand() % 10);
                         }
                    else
                         {
                         x1 = iKeep [NUM - 1 - j] [0];
                         y1 = iKeep [NUM - 1 - j] [1];
                         x2 = iKeep [NUM - 1 - j] [2];
                         y2 = iKeep [NUM - 1 - j] [3];
                         }
                    BitBlt(hdcMem, 0, 0, cx, cy, hdc,  x1, y1, SRCCOPY);
                    BitBlt(hdc,  x1, y1, cx, cy, hdc,  x2, y2, SRCCOPY);
                    BitBlt(hdc,  x2, y2, cx, cy, hdcMem, 0, 0, SRCCOPY);

                    Sleep(10);
                    }

          DeleteDC(hdcMem);
```

```
        DeleteDC(hdc);
        DeleteObject(hBitmap);

        LockWindowUpdate(NULL);
        }

    return FALSE;
    }
```

Рис. 4.26 Программа SCRAMBLE

В программе SCRAMBLE нет оконной процедуры. В функции *WinMain* она получает контекст устройства для всего экрана:

```
hdc = CreateDC("DISPLAY", NULL, NULL, NULL);
```

а также контекст памяти:

```
hdcMem = CreateCompatibleDC(hdc);
```

Затем она определяет размеры экрана и делит их на 10:

```
xSize = GetSystemMetrics(SM_CXSCREEN) / 10;
ySize = GetSystemMetrics(SM_CYSCREEN) / 10;
```

Программа использует эти размеры для создания битового образа:

```
hBitmap = CreateCompatibleBitmap(hdc, xSize, ySize);
```

и выбирает его в контекст памяти:

```
SelectObject(hdcMem, hBitmap);
```

Используя функцию *rand* языка С, программа SCRAMBLE формирует четыре случайных величины, кратные значениям *xSize* и *ySize*:

```
x1 = xSize *(rand() % 10);
y1 = ySize *(rand() % 10);
x2 = xSize *(rand() % 10);
y2 = ySize *(rand() % 10);
```

Программа меняет местами два прямоугольных блока дисплея, используя три функции *BitBlt*. Первая копирует прямоугольник с вершиной в точке (*x1, y1*) в контекст памяти:

```
BitBlt(hdcMem, 0, 0, xSize, ySize, hdc, x1, y1, SRCCOPY);
```

Вторая копирует прямоугольник с вершиной в точке (*x2, y2*) в прямоугольную область с вершиной в точке (*x1, y1*):

```
BitBlt(hdc, x1, y1, xSize, ySize, hdc, x2, y2, SRCCOPY);
```

Третья копирует прямоугольник из контекста памяти в прямоугольную область с вершиной в точке (*x2, y2*):

```
BitBlt(hdc, x2, y2, xSize, ySize, hdcMem, 0, 0, SRCCOPY);
```

Этот процесс эффективно меняет местами содержимое двух прямоугольников на дисплее. SCRAMBLE делает это 200 раз, что может привести к полному беспорядку на экране. Но этого не происходит, потому что программа SCRAMBLE отслеживает свои действия, и перед завершением восстанавливает экран.

Вы можете также использовать контекст памяти для копирования содержимого одного битового образа в другой. Предположим, вы хотите создать битовый образ, содержащий только левый верхний квадрант другого битового образа. Если исходный битовый образ имеет описатель *hBitmap*, то вы можете скопировать его размеры в структуру типа BITMAP:

```
GetObject(hBitmap, sizeof(BITMAP),(LPVOID) &bm);
```

и создать новый неинициализированный битовый образ размером в одну четверть исходного:

```
hBitmap2 = CreateBitmap(bm.bmWidth / 2, bm.bmHeight / 2, bm.bmPlanes, bm.bmBitsPixel, NULL);
```

Теперь создаются два контекста памяти и в них выбираются исходный и новый битовые образы:

```
hdcMem1 = CreateCompatibleDC(hdc);
hdcMem2 = CreateCompatibleDC(hdc);

SelectObject(hdcMem1, hBitmap);
SelectObject(hdcMem2, hBitmap2);
```

Теперь копируем левый верхний квадрант первого битового образа во второй:

```
BitBlt(hdcMem2, 0, 0, bm.bmWidth / 2, bm.bmHeight / 2, hdcMem1, 0, 0, SRCCOPY);
```

Все сделано, кроме очистки:

```
DeleteDC(hdcMem1);
DeleteDC(hdcMem2);
DeleteObject(hBitmap);
```

Преобразования цветов

Если приемный и исходный контексты устройства в функции *BitBlt* имеют различные цветовые характеристики, то операционная система Windows должна преобразовать битовый образ из одного цветового формата в другой. Наилучший результат преобразования достигается в случае, если исходный битовый образ монохромный. Windows использует атрибуты цвета текста и фона приемного контекста устройства для преобразования:

Монохромный DC (Источник)	Цветной DC (Приемник)
0 (Черный)	Цвет текста (по умолчанию черный)
1 (Белый)	Цвет фона (по умолчанию белый)

Кроме того, атрибут цвета фона используется Windows для заполнения пробелов в точечных и штриховых линиях, а также между штрихами в штриховых кистях. Вы можете изменить цвет фона с помощью функции *SetBkColor*. Цвет фона, с которым мы еще встретимся позднее в этой главе, определяет цвет текста. Вы можете изменить его с помощью функции *SetTextColor*. По умолчанию, монохромный битовый образ просто преобразуется в черно-белый битовый образ в цветном контексте устройства.

Преобразование битового образа из цветного контекста устройства в монохромный приемный контекст устройства является менее удовлетворительным:

Цветной DC (Источник)	Монохромный DC (Приемник)
Пиксель != Цвет фона	0 (Черный)
Пиксель == Цвет фона	1 (Белый)

В этом случае Windows использует цвет фона исходного контекста устройства для определения того, какой цвет преобразовывать в белый. Любой другой цвет преобразуется в черный.

Здесь существует проблема, связанная с цветами: Windows необходимо сравнить конкретную комбинацию битов цвета битового образа (лежащих в разных цветовых плоскостях или в одной плоскости) с 24-битным значением цвета фона. Это значит, что цветной контекст устройства должен относиться к физическому устройству или быть контекстом памяти на базе реального устройства. Например, у вас имеется монохромный драйвер устройства. Вы строите контекст памяти на базе дисплейного контекста устройства и выбираете цветной битовый образ в контекст памяти, а затем делаете попытку перенести битовый образ в монохромный контекст устройства. Это не удается, потому что Windows не знает, как множество цветовых плоскостей или множество битов на пиксель битового образа контекста памяти соотносится с реальными цветами.

Преобразования режимов отображения

Вызов функции *BitBlt* требует задания различных начальных координат для исходного и приемного контекстов устройства, но при этом должна быть задана одна ширина и одна высота:

```
BitBlt(hdcDest, xDest, yDest, xWidth, yHeight, hdcSrc, xSrc, ySrc, dwROP);
```

Величины *xWidth* и *yHeight* задаются в логических единицах и относятся одновременно к прямоугольникам исходного и приемного контекста устройства. Функция *BitBlt* должна преобразовать все координаты и размеры в координаты устройства перед вызовом драйвера для выполнения операции. Поскольку значения *xWidth* и *yHeight* используются для обоих контекстов устройства, они должны преобразовываться в единицы устройства (пиксели) независимо для каждого контекста устройства.

Когда исходный и приемный контексты устройства равны или когда оба контекста устройства используют режим отображения MM_TEXT, размер этого прямоугольника в единицах устройства будет одинаковым для обоих контекстов устройства. Windows тогда может осуществить простой перенос пиксель-в-пиксель. Однако, когда размеры прямоугольника в единицах устройства различны в двух контекстах устройства, Windows перекладывает работу на более мощную функцию *StretchBlt*.

Растяжение битовых образов с помощью функции *StretchBlt*

Функция *StretchBlt* имеет два дополнительных параметра по сравнению с функцией *BitBlt*:

```
StretchBlt(
       hdcDest, xDest, yDest, xDestWidth, yDestHeight,
       hdcSrc, xSrc, ySrc, xSrcWidth, ySrcHeight, dwROP
);
```

Поскольку функция *StretchBlt* имеет различные параметры ширины и высоты исходного и приемного прямоугольников, становится возможным растяжение или сжатие битового образа исходного контекста устройства для того, чтобы занять большую или меньшую область в приемном контексте устройства.

Так же как функция *BitBlt* — есть расширение функции *PatBlt*, так и функция *StretchBlt* — есть расширение функции *BitBlt*, позволяющее задавать раздельно размеры исходного и приемного прямоугольника. Как и у функций *PatBlt* и *BitBlt* все координаты и значения в функции *StretchBlt* задаются в логических единицах. Функция *StretchBlt* также позволяет вам переворачивать изображение по горизонтали и вертикали. Если знаки *xSrcWidth* и *xDestWidth* (при преобразовании в единицы устройства) различны, то функция *StretchBlt* создает зеркальное изображение: левая часть становится правой, правая часть — левой. Если знаки *ySrcHeight* и *yDestHeight* (при преобразовании в единицы устройства) различны, то функция *StretchBlt* переворачивает изображение по вертикали.

Функция *StretchBlt* может работать достаточно медленно, особенно с большими битовыми образами. Кроме того, существуют проблемы, связанные с трудностями масштабирования битовых образов. При растяжении битового образа функция *StretchBlt* должна дублировать строки или столбцы пикселей. Если растяжение происходит не в целое число раз, то в результате может получиться искаженный образ.

При сжатии битового образа функция *StretchBlt* должна комбинировать две или более строки или столбца пикселей в одну строку или столбец. Она делает это одним из трех способов в зависимости от атрибута режима растяжения в контексте устройства. Вы можете использовать функцию *SetStretchBltMode* для изменения этого атрибута:

```
SetStretchBltMode(hdc, iMode);
```

Величина *iMode* может принимать следующие значения:

- BLACKONWHITE (по умолчанию) — Если два или более пикселей должны быть преобразованы в один пиксель, то функция *StretchBlt* выполняет логическую операцию AND над пикселями. Результирующий пиксель будет белым только в том случае, если все исходные пиксели были белыми, что на практике означает, что черные пиксели преобладают над белыми.
- WHITEONBLACK — Если два или более пикселей должны быть преобразованы в один пиксель, то функция *StretchBlt* выполняет логическую операцию OR над пикселями. Результирующий пиксель будет черным только в том случае, если все исходные пиксели были черными, что означает, что белые пиксели преобладают над черными.
- COLORONCOLOR — функция *StretchBlt* просто уничтожает строки или столбцы пикселей без выполнения логических операций. Иногда, это лучший подход для цветных битовых образов, поскольку два других режима могут вызвать искажения цветов.

Кисти и битовые образы

Когда вы используете функции *CreatePatternBrush* или *CreateBrushIndirect* с полем *lbStyle*, установленным в значение BS_PATTERN, вы должны сначала получить описатель битового образа. Битовый образ должен быть размером как минимум 8 на 8 пикселей. Если он больше, то Windows использует только левый верхний угол битового образа для кисти.

Поскольку кисти и битовые образы — объекты GDI, вы должны удалить все объекты, созданные вами, до того, как ваша программа завершится. Когда вы создаете кисть на базе битового образа, Windows делает копию данных битового образа для использования при рисовании кистью. Вы можете удалить битовый образ сразу же после вызова функций *CreatePatternBrush* или *CreateBrushIndirect* без какого-либо влияния на кисть. Аналогично, вы можете удалить кисть без влияния на битовый образ.

Скажем, вы хотите нарисовать прямоугольник, закрашенный кистью, в виде сетки, как показано на рис. 4.27.

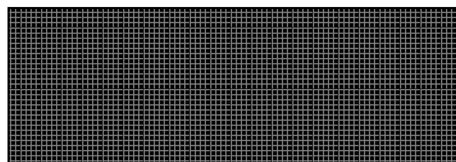
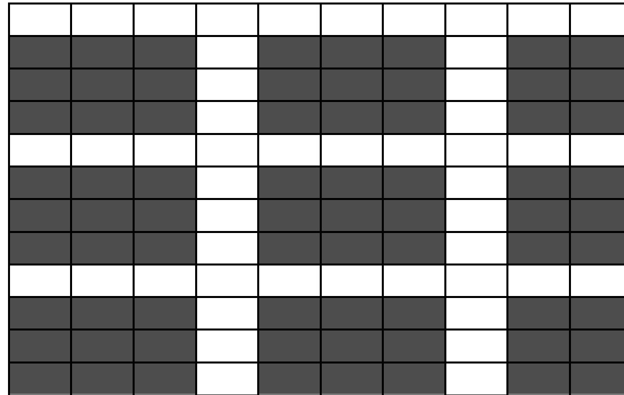


Рис. 4.27 Фигура, закрашенная пользовательской кистью

Битовый образ, который вам нужен, выглядит так:



Это монохромный битовый образ с высотой и шириной в 8 пикселей.

Как будет показано в главе 9, вы можете создать битовый образ в виде небольшого файла в программе Microsoft Developer Studio и определить его в вашей программе как ресурс. Загружая его, вы получаете описатель битового образа:

```
hBitmap = LoadBitmap(hInstance, "Brick");
hBrush = CreatePatternBrush(hBitmap);
```

Когда у вас будет действительный контекст устройства, выберите кисть в контекст устройства и отобразите прямоугольник:

```
SelectObject(hdc, hBrush);
Rectangle(hdc, xLeft, yTop, xRight, yBottom);
```

Когда вы освободите контекст устройства, удалите кисть и битовый образ:

```
DeleteObject(hBrush);
DeleteObject(hBitmap);
```

Для удаления битового образа не нужно ждать, пока освободится контекст устройства. Вы можете сделать это в любой момент после создания кисти на основе битового образа.

Можно также описать пиксели битового образа в вашей программе как массив восьми беззнаковых целых. Каждое целое соответствует скан-линии в шаблоне битового образа. Бит, равный 1, используется для задания белого цвета, бит, равный 0 — для черного:

```
HBITMAP hBitmap;
HBRUSH hBrush;
static WORD wBrickBits [] =
     { 0xFF, 0x0C, 0x0C, 0x0C, 0xFF, 0xC0, 0xC0, 0xC0 };
```

Битовый образ создается функцией *CreateBitmap* с параметром — ссылкой на массив целых:

```
hBitmap = CreateBitmap(8, 8, 1, 1,(LPVOID) &wBrickBits);
hBrush = CreatePatternBrush(hBitmap);
```

И далее продолжать, как указано выше.

Метафайлы

Метафайлы имеют такое же значение для векторной графики, как и битовые образы для растровой графики. Метафайлы конструируются человеком или компьютерной программой. Метафайл состоит из набора записей, соответствующих вызовам графических функций, например, рисующих прямые линии, кривые, закрашенные области и текст. Таким образом, метафайл — набор вызовов графических функций, закодированный в двоичной форме.

Программы рисования создают битовые образы, программы черчения — метафайлы. В программах черчения вы можете "захватить" конкретный графический объект (такой как линия) и переместить его в другое место. Это возможно потому, что все компоненты чертежа хранятся как отдельные записи. В программах рисования такие действия невозможны — вы можете только удалять и вставлять прямоугольные фрагменты битового образа.

Поскольку метафайл описывает изображение в терминах команд графического вывода, изображение может быть промасштабировано без потери разрешения. С битовыми образами это невозможно. Если вы отображаете битовый образ в увеличенном в два раза виде, то вы не получите двойного разрешения. Биты внутри битового образа будут просто повторяться горизонтально и вертикально.

Метафайл может быть преобразован в битовый образ без потери информации. Графические объекты, составляющие метафайл, теряют свою индивидуальность и соединяются вместе в одном большом образе. Преобразование битового образа в метафайл — гораздо более сложная задача, решаемая обычно только для простых битовых образов, и требующая высокой мощности процессора для выявления углов и контуров.

Метафайлы чаще всего используются для разделения изображений между программами посредством буфера обмена, хотя они могут также быть сохранены на диске в виде файлов. Поскольку метафайлы описывают изображение как набор вызовов GDI, для их хранения требуется значительно меньше места, и они более независимы от устройства, чем битовые образы.

Сначала мы рассмотрим функции работы с метафайлами, которые поддерживаются Windows, начиная с версии 1.0 и кончая Windows 95, а затем остановимся на "расширенных метафайлах", разработанных для Windows NT, но также поддерживаемых Windows 95. Расширенные метафайлы имеют серьезные преимущества в сравнении с метафайлами старого формата и значительно лучше приспособлены для дискового хранения.

Простое использование метафайлов памяти

Для создания метафайла сначала надо построить контекст устройства метафайла путем вызова функции *CreateMetaFile*. Затем вы можете использовать большинство функций рисования GDI в этом контексте метафайла. Эти вызовы функций GDI ничего не будут выводить на экран. Вместо этого они будут запоминаться в метафайле. Когда вы закроете контекст устройства метафайла, вы получите описатель метафайла. Затем, вы можете "проиграть" этот метафайл на реальном контексте устройства и выполнить функции GDI метафайла.

Функция *CreateMetaFile* имеет один параметр. Он может быть равен NULL или быть именем файла. Если NULL, то метафайл запоминается в оперативной памяти. Если указано имя файла (обычно с расширением .WMF — "Windows Metafile"), то метафайл запоминается как дисковый файл.

Программа METAFILE, приведенная на рис. 4.28, показывает, как создать метафайл в памяти во время обработки сообщения WM_CREATE и 100 раз вывести изображение во время обработки сообщения WM_PAINT.

METAFILE.MAK

```
#------------------------
# METAFILE.MAK make file
#------------------------

metafile.exe : metafile.obj
     $(LINKER) $(GUIFLAGS) -OUT:metafile.exe metafile.obj $(GUILIBS)

metafile.obj : metafile.c
     $(CC) $(CFLAGS) metafile.c
```

METAFILE.C

```
/*----------------------------------------------
   METAFILE.C -- Metafile Demonstration Program
                 (c) Charles Petzold, 1996
  ----------------------------------------------*/

#include <windows.h>

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName [] = "Metafile";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
```

```
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Metafile Demonstration",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HMETAFILE hmf;
     static int       cxClient, cyClient;
     HBRUSH           hBrush;
     HDC              hdc, hdcMeta;
     int              x, y;
     PAINTSTRUCT      ps;

     switch(iMsg)
          {
          case WM_CREATE:
               hdcMeta = CreateMetaFile(NULL);
               hBrush  = CreateSolidBrush(RGB(0, 0, 255));

               Rectangle(hdcMeta, 0, 0, 100, 100);

               MoveToEx(hdcMeta,   0,   0, NULL);
               LineTo  (hdcMeta, 100, 100);
               MoveToEx(hdcMeta,   0, 100, NULL);
               LineTo  (hdcMeta, 100,   0);

               SelectObject(hdcMeta, hBrush);
               Ellipse(hdcMeta, 20, 20, 80, 80);

               hmf = CloseMetaFile(hdcMeta);

               DeleteObject(hBrush);
               return 0;

          case WM_SIZE:
               cxClient = LOWORD(lParam);
               cyClient = HIWORD(lParam);
               return 0;

          case WM_PAINT:
               hdc = BeginPaint(hwnd, &ps);
```

```
            SetMapMode(hdc, MM_ANISOTROPIC);
            SetWindowExtEx(hdc, 1000, 1000, NULL);
            SetViewportExtEx(hdc, cxClient, cyClient, NULL);

            for(x = 0; x < 10; x++)
                 for(y = 0; y < 10; y++)
                      {
                      SetWindowOrgEx(hdc, -100 * x, -100 * y, NULL);
                      PlayMetaFile(hdc, hmf);
                      }
            EndPaint(hwnd, &ps);
            return 0;

       case WM_DESTROY:
            DeleteMetaFile(hmf);
            PostQuitMessage(0);
            return 0;
       }
  return DefWindowProc(hwnd, iMsg, wParam, lParam);
  }
```

Рис. 4.28 Программа METAFILE

Эта программа демонстрирует использование четырех функций метафайлов, работающих с метафайлом памяти. Первая функция — *CreateMetaFile*, вызванная с параметром NULL в теле обработчика сообщения WM_CREATE. Эта функция возвращает описатель контекста устройства метафайла. Затем программа METAFILE рисует прямоугольник, две прямые и один голубой эллипс, используя данный контекст устройства метафайла. Вызовы этих 4-х функций запоминаются в двоичной форме в метафайле. Функция *CloseMetaFile* возвращает описатель метафайла. Обратите внимание, что описатель метафайла запоминается в статической переменной, поскольку он используется позднее.

Метафайл содержит двоичное представление вызовов функций GDI — одного вызова *Rectangle*, двух вызовов *MoveToEx*, двух вызовов *LineTo*, вызова *SelectObject* (задающего голубую кисть) и вызова *Ellipse*. Режим отображения или преобразования не описывается координатами. Они просто запоминаются в метафайле в числовом виде.

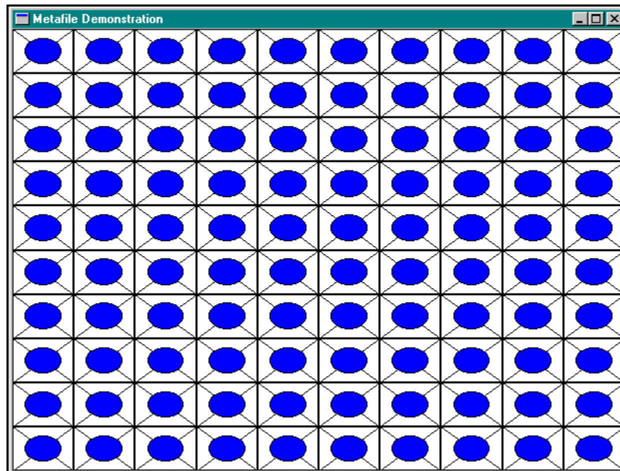


Рис. 4.29 Вывод программы METAFILE

Во время обработки сообщения WM_PAINT программа METAFILE устанавливает режим отображения и вызывает функцию *PlayMetaFile* для рисования объекта в окне 100 раз. Координаты вызовов функций в метафайле интерпретируются в соответствии с режимом отображения приемного контекста устройства. Вызывая *PlayMetaFile*, вы повторяете все вызовы функций, сделанные между вызовами функций *CreateMetaFile* и *CloseMetaFile* при первоначальном создании метафайла во время обработки сообщения WM_CREATE.

Как и другие объекты GDI, метафайлы должны быть тоже удалены до завершения программы. Процедура удаления реализуется в теле обработчика сообщения WM_DESTROY с помощью функции *DeleteMetaFile*.

Результат работы программы METAFILE приведен на рис. 4.29.

Сохранение метафайлов на диске

В приведенном выше примере использование NULL в качестве параметра функции *CreateMetaFile* означало, что мы хотим создать метафайл в памяти. Мы можем также создать метафайл, сохраняемый на диске как обычный файл. Этот метод предпочтителен для больших метафайлов, поскольку он требует меньше памяти. Windows необходимо выделить относительно небольшой фрагмент памяти для хранения имени файла, содержащего метафайл. Однако, каждый раз, когда вы проигрываете метафайл, сохраненный на диске, будет осуществляться доступ к диску.

Для преобразования программы METAFILE для работы с дисковым метафайлом вам необходимо заменить параметр NULL в функции *CreateMetaFile* именем файла. По завершении обработки сообщения функцией WM_CREATE вы можете удалить описатель метафайла с помощью функции *DeleteMetaFile*. Описатель удаляется, но файл на диске остается.

Во время обработки сообщения WM_PAINT вы можете получить описатель метафайла, соответствующего этому дисковому файлу, вызывая функцию *GetMetaFile*:

```
hmf = GetMetaFile(szFileName);
```

Теперь вы можете проиграть метафайл также, как раньше. Когда обработка сообщения WM_PAINT закончится, вы можете удалить описатель метафайла:

```
DeleteMetaFile(hmf);
```

Когда придет время обработки сообщения WM_DESTROY, вам не нужно удалять метафайл, поскольку он был удален по завершении обработки сообщения WM_CREATE и в конце обработки каждого сообщения WM_PAINT. Но вам следует удалить дисковый файл (конечно, в том случае, если он вам более не нужен):

```
remove(szFileName);
```

В главе 9 будут рассмотрены ресурсы, определяемые пользователем. Ресурсы — это обычные двоичные данные, хранящиеся в EXE-файле программы, но отдельно от обычных областей кода и данных. Метафайл тоже может быть ресурсом. Если у вас есть блок данных, содержащий метафайл, вы можете создать метафайл, используя функцию:

```
hmf = SetMetaFileBitsEx(iSize, pData);
```

Функция *SetMetaFileBitsEx* имеет парную функцию *GetMetaFileBitsEx*, которая копирует содержимое метафайла в блок памяти.

Расширенные метафайлы

С устаревшими (но еще поддерживаемыми) метафайлами, рассмотренными выше, связаны некоторые проблемы. В частности, программа, использующая метафайл, созданный другой программой, не может легко определить размер отображаемого образа, представляемого метафайлом. Ей нужно просмотреть метафайл и проанализировать все команды рисования. Это большая проблема.

Ранее Microsoft рекомендовала создавать метафайлы, не содержащие вызовов функции *SetMapMode* и других функций, изменяющих преобразование окно/область вывода. Соблюдение данной рекомендации делает метафайлы зависимыми от устройства и не дает возможности приложению изменить размер выводимого изображения. Таким образом, все координаты в метафайле — просто числа, не связанные с какой-либо системой координат.

Как мы увидим в главе 16, метафайлы устаревшего типа не передаются через буфер обмена непосредственно. Вместо этого, буфер обмена работает с неким объектом под названием "картина метафайла" (metafile picture). Это структура типа METAFILEPICT. Описатель метафайла является полем этой структуры. Кроме того, в этой структуре также содержатся идентификатор режима отображения (отражающий единицы измерения по осям координат для всех функций GDI, содержащихся в метафайле) и размеры изображения. Эта информация помогает программе, импортирующей метафайл, установить соответствующую среду GDI для отображения образа.

Структура картины метафайла — настоящая находка. Ее, очевидно, добавили в GDI для устранения недостатков формата метафайла. Следует отметить, что аналогичная возможность не была разработана для дисковых метафайлов, и именно поэтому вам не удавалось увидеть большинство из этих файлов, используемых для обмена картинками. Они просто очень сложны для использования.

Делаем это лучше

Также как и Windows NT, Windows 95 поддерживает новый формат "расширенного метафайла". Кроме того добавляются несколько новых функций, несколько новых структур данных, новый формат буфера обмена и новое расширение файла — EMF.

Главное усовершенствование состоит в том, что новый формат метафайла содержит более обширный информационный заголовок, доступный посредством вызова функций. Эта информация призвана помочь приложениям выводить изображения, содержащиеся в метафайлах.

Некоторые из расширенных функций работы с метафайлами позволяют вам преобразовывать метафайлы из нового расширенного формата .EMF в устаревший формат .WMF и обратно. Конечно, это преобразование не может быть произведено без потерь. Устаревший формат метафайлов не поддерживает некоторые из новых графических возможностей (например, пути).

Базовая процедура

На рис. 4.30 приведена программа EMF1, которая создает и выводит на экран расширенный метафайл.

EMF1.MAK

```
#--------------------
# EMF1.MAK make file
#--------------------

emf1.exe : emf1.obj
     $(LINKER) $(GUIFLAGS) -OUT:emf1.exe emf1.obj $(GUILIBS)

emf1.obj : emf1.c
     $(CC) $(CFLAGS) emf1.c
```

EMF1.C

```
/*-------------------------------------
   EMF1.C -- Enhanced Metafile Demo #1
            (c) Charles Petzold, 1996
  -------------------------------------*/

#include <windows.h>

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "EMF1";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Enhanced Metafile Demo #1",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
```

```
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HENHMETAFILE hemf;
     HDC                 hdc, hdcEMF;
     PAINTSTRUCT         ps;
     RECT                rect;

     switch(iMsg)
          {
          case WM_CREATE:
               hdcEMF = CreateEnhMetaFile(NULL, NULL, NULL, NULL);

               Rectangle(hdcEMF, 100, 100, 200, 200);

               MoveToEx (hdcEMF, 100, 100, NULL);
               LineTo   (hdcEMF, 200, 200);

               MoveToEx (hdcEMF, 200, 100, NULL);
               LineTo   (hdcEMF, 100, 200);

               hemf = CloseEnhMetaFile(hdcEMF);

               return 0;
          case WM_PAINT:
               hdc = BeginPaint(hwnd, &ps);

               GetClientRect(hwnd, &rect);

               rect.left   =     rect.right  / 4;
               rect.right  = 3 * rect.right  / 4;
               rect.top    =     rect.bottom / 4;
               rect.bottom = 3 * rect.bottom / 4;

               PlayEnhMetaFile(hdc, hemf, &rect);

               EndPaint(hwnd, &ps);
               return 0;

          case WM_DESTROY:
               DeleteEnhMetaFile(hemf);

               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

Рис. 4.30 Программа EMF1

В процессе обработки сообщения WM_CREATE в оконной процедуре программы EMF1, с помощью функции *CreateEnhMetaFile* создается расширенный метафайл. Эта функция требует задания четырех параметров, но вы можете задать их все равными NULL. (Как удобно!) Позднее будет рассказано, как использовать эту функцию с параметрами, отличными от NULL.

Так же, как и функция *CreateMetaFile*, функция *CreateEnhMetaFile* возвращает описатель специального контекста устройства. Программа использует этот описатель для рисования прямоугольника и двух прямых, соединяющих противоположные углы прямоугольника. Эти вызовы функций конвертируются в двоичный вид и запоминаются в метафайле.

Наконец, вызов функции *CloseEnhMetaFile* завершает формирование расширенного метафайла и возвращает его описатель. Он запоминается в статической переменной типа HENHMETAFILE.

В процессе обработки сообщения WM_PAINT программа EMF1 получает размеры рабочей области окна и записывает их в структуру типа RECT. Четыре поля этой структуры пересчитываются таким образом, что прямоугольник приобретает ширину, равную половине ширины рабочей области, и высоту, равную половине высоты рабочей области, а весь прямоугольник располагается в центре рабочей области. Затем программа EMF1 вызывает функцию *PlayEnhMetaFile*. Первый параметр этой функции — описатель контекста устройства окна, второй — описатель расширенного метафайла, третий параметр — указатель на структуру типа RECT.

Здесь произойдет то, что происходило при создании метафайла — GDI вычислит размеры изображения, хранящегося в метафайле. В нашем случае изображение имеет ширину и высоту 100 единиц. При отображении метафайла GDI растягивает изображение так, чтобы полностью занять указанный в вызове функции *PlayEnhMetaFile* прямоугольник. Три примера работы программы EMF1 в среде Windows 95 показаны на рис. 4.31.

Наконец, при обработке сообщения WM_DESTROY программа EMF1 удаляет метафайл, вызывая функцию *DeleteEnhMetaFile*.

Давайте обратим внимание на некоторые вещи, которым мы могли бы научиться из программы EMF1.

Во-первых, в этой конкретной программе координаты, используемые функциями рисования прямоугольника и линий для создания расширенного метафайла, на самом деле определяют далеко не все. Вы можете удвоить их значения или вычесть константу из них — результат останется таким же. Таким образом, можно сделать вывод, что координаты имеют взаимосвязи друг с другом в определении образа.

Во-вторых, изображение растягивается так, чтобы занять весь прямоугольник, передаваемый как параметр в функцию *PlayEnhMetaFile*. Следовательно, как показано на рис. 4.31, изображение может быть искажено. При задании координат в метафайле предполагается, что изображение — квадрат, но в общем случае мы его не получим.

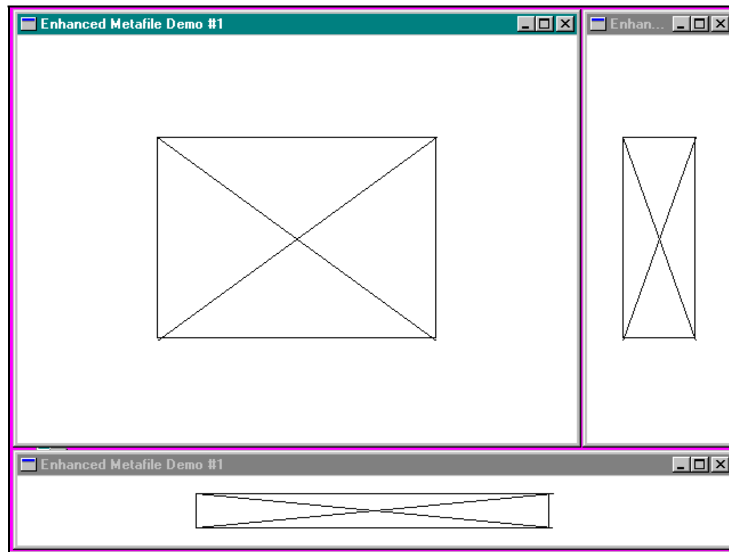


Рис. 4.31 Окна программ EMF1.

Иногда это как раз то, что требуется. В программах обработки текстов пользователю может быть предоставлена возможность задавать прямоугольник для изображения так, чтобы все изображение целиком занимало указанную область без потерь места. Пусть пользователь устанавливает правильный относительный размер, изменяя соответствующим образом размеры прямоугольника.

Однако, существуют случаи, когда нужно другое. Можно установить относительный размер исходного изображения, поскольку это бывает крайне важно для визуального представления информации. Например, эскиз лица подозреваемого в преступлении (полицейский фоторобот) не должен быть искажен относительно оригинала. Или можно сохранить метрические размеры исходного изображения, так как важно то, что образ имеет высоту два дюйма, и не должен изображаться иначе.

Заглянем внутрь

Программа EMF2, приведенная на рис. 4.32, строит дисковый метафайл.

EMF2.MAK

```
#--------------------
# EMF2.MAK make file
#--------------------

emf2.exe : emf2.obj
     $(LINKER) $(GUIFLAGS) -OUT:emf2.exe emf2.obj $(GUILIBS)

emf2.obj : emf2.c
     $(CC) $(CFLAGS) emf2.c
```

EMF2.C

```
/*-------------------------------------
   EMF2.C -- Enhanced Metafile Demo #2
            (c) Charles Petzold, 1996
  -------------------------------------*/

#include <windows.h>

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "EMF2";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Enhanced Metafile Demo #2",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     HDC          hdc, hdcEMF;
     HENHMETAFILE hemf;
     PAINTSTRUCT  ps;
     RECT         rect;

     switch(iMsg)
          {
          case WM_CREATE:
               hdcEMF = CreateEnhMetaFile(NULL, "emf2.emf", NULL,
                                          "EMF2\0EMF Demo #2\0");

               Rectangle(hdcEMF, 100, 100, 200, 200);

               MoveToEx (hdcEMF, 100, 100, NULL);
               LineTo   (hdcEMF, 200, 200);

               MoveToEx (hdcEMF, 200, 100, NULL);
               LineTo   (hdcEMF, 100, 200);

               hemf = CloseEnhMetaFile(hdcEMF);
               DeleteEnhMetaFile(hemf);
               return 0;

          case WM_PAINT:
               hdc = BeginPaint(hwnd, &ps);

               GetClientRect(hwnd, &rect);

               rect.left   =     rect.right  / 4;
               rect.right  = 3 * rect.right  / 4;
               rect.top    =     rect.bottom / 4;
               rect.bottom = 3 * rect.bottom / 4;

               hemf = GetEnhMetaFile("emf2.emf");

               PlayEnhMetaFile(hdc, hemf, &rect);

               DeleteEnhMetaFile(hemf);

               EndPaint(hwnd, &ps);
               return 0;

          case WM_DESTROY:
               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

Рис. 4.32 Программа EMF2

Обратите внимание, что первый параметр функции *CreateEnhMetaFile* — это описатель контекста устройства. GDI использует этот параметр для вставки метрической информации в заголовок метафайла. Если этот параметр установлен в NULL, то GDI берет эту метрическую информацию из контекста устройства дисплея.

Второй параметр функции *CreateEnhMetaFile* — это имя файла. Если вы установите этот параметр в NULL (как в программе EMF1), то функция построит метафайл в памяти. Программа EMF2 строит дисковый метафайл с именем EMF2.EMF.

Третий параметр функции — адрес структуры типа RECT, описывающей общие размеры метафайла. Эта часть важнейшей информации (та, что отсутствовала в предыдущем формате метафайлов Windows) заносится в заголовок метафайла. Если вы установите этот параметр в NULL, то GDI определит размеры за вас. Приятно, что

операционная система делает это для нас, поэтому имеет смысл устанавливать этот параметр в NULL. Если производительность вашего приложения критична, то вы можете использовать этот параметр для того, чтобы избежать лишней работы GDI.

Наконец, последний параметр — текстовая строка, описывающая метафайл. Эта строка делится на две части: Первая часть — имя приложения (не обязательно имя программы), за которым следует NULL-символ. Вторая часть описывает визуальный образ. Эта часть завершается двумя NULL-символами. Например, используя нотацию языка C '\0' для NULL-символа, можно получить строку описания следующего вида "HemiDemiSemiCad V6.4\0Flying Frogs\0\0". Поскольку C обычно помещает NULL-символ в конец строки, заданной в кавычках, вам нужен только один символ '\0' в конце строки, как показано в программе EMF2.

После создания метафайла программа EMF2 работает также, как программа EMF1, вызывая несколько функций GDI и используя описатель контекста устройства, возвращенный функцией *CreateEnhMetaFile*. Эти функции рисуют прямоугольник и две линии, соединяющие его противоположные вершины. Затем программа вызывает функцию *CloseEnhMetaFile* для уничтожения описателя контекста устройства и получения описателя сформированного метафайла.

Затем, еще при обработке сообщения WM_CREATE, программа EMF2 делает то, чего не делала программа EMF1: сразу после получения описателя метафайла программа вызывает функцию *DeleteEnhMetaFile*. Этот вызов освобождает все ресурсы памяти, необходимые для построения метафайла. Однако дисковый метафайл сохраняется. (Если вы когда-нибудь захотите избавиться от этого файла, то используйте обычные функции удаления файлов.) Обратите внимание, что описатель метафайла не запоминается в статической переменной, как в программе EMF1, поскольку не требуется его сохранение в промежутке времени между обработкой различных сообщений.

Теперь, для использования созданного метафайла программе EMF2 необходимо получить доступ к дисковому файлу. Она делает это при обработке сообщения WM_PAINT путем вызова функции *GetEnhMetaFile*. Единственный параметр этой функции — имя метафайла. Функция возвращает описатель метафайла. Программа EMF2 передает этот описатель в функцию *PlayEnhMetaFile*, так же как программа EMF1. Изображение из метафайла выводится в прямоугольник, заданный последним параметром функции. Но в отличие от программы EMF1, программа EMF2 удаляет метафайл перед завершением обработки сообщения WM_PAINT. При обработке последующих сообщений WM_PAINT программа EMF2 опять получает метафайл, проигрывает его и потом удаляет.

Запомните, что удаление метафайла влечет за собой удаление только ресурсов памяти, требуемых для построения метафайла. Дисковый метафайл сохраняется даже после завершения программы.

Поскольку программа EMF2 оставляет не удаленным дисковый метафайл, вы можете взглянуть на него. Он состоит из записей переменной длины, описываемых структурой ENHMETARECORD, определенной в заголовочных файлах Windows. Расширенный метафайл всегда начинается с заголовка типа структуры ENHMETAHEADER.

Вам не нужен доступ к физическому дисковому метафайлу для получения заголовочной информации. Если у вас есть описатель метафайла, то вы можете использовать функцию *GetEnhMetaFileHeader*:

```
GetEnhMetaFileHeader(hemf, cbSize, &emh);
```

Первый параметр — описатель метафайла. Последний — указатель на структуру типа ENHMETAHEADER, а второй — размер этой структуры. Вы можете использовать функцию *GetEnhMetaFileDescription* для получения строки описания. Поле *rclBounds* структуры ENHMETAHEADER — структура прямоугольника, хранящая размеры изображения в пикселях. Поле *rclFrame* структуры ENHMETAHEADER — структура прямоугольника, хранящая размеры изображения в других единицах (0.01 мм).

Заголовочная запись завершается двумя структурами типа SIZE, содержащими два 32-разрядных поля, *szlDevice* и *szlMillimeters*. Поле *szlDevice* хранит размеры устройства вывода в пикселях, а поле *szlMillimeters* хранит размеры устройства вывода в миллиметрах. Эти данные основываются на контексте устройства, описатель которого передается в функцию *CreateEnhMetaFile* первым параметром. Если этот параметр равен NULL, то GDI использует экран дисплея. GDI получает метрические данные с помощью функции *GetDeviceCaps*.

Вывод точных изображений

Большой плюс изображений, хранящихся в виде метафайлов, состоит в том, что они могут растягиваться до любого размера и при этом оставаться правильными. Увеличение или сжатие изображения — это просто масштабирование всех координатных точек, определяющих примитивы. С другой стороны, битовые образы могут терять правильность изображения при сжатии в результате отбрасывания целых строк или столбцов пикселей.

Иногда, однако, произвольное масштабирование метафайла — не очень хорошая идея. Может оказаться смешным растягивание содержащегося в метафайле образа, в виде лица человека, в толстое или тонкое лицо. В этом случае

надо знать правильный относительный размер образа. Некоторые изображения метафайлов могут иметь смысл только при конкретных физических размерах.

Как мы уже видели, последний параметр функции *PlayEnhMetaFile* — структура типа RECT, сообщающая GDI, где вы хотите нарисовать изображение в приемном контексте устройства. GDI растягивает образ так, чтобы он занял полностью указанный прямоугольник. Точный вывод изображений метафайлов — в заданных метрических размерах или с соответствующим относительным размером, требует использования информации о размерах из заголовка метафайла и точной установки размеров прямоугольника.

Это кажется достаточно сложным, но еще неизвестно, что может получиться при выводе изображения на принтер. Поэтому эту работу отложим до главы 15.

Текст и шрифты

Первая задача, которую мы решали в этой книге, используя программирование под Windows, была задача отображения простой строки текста в центре окна. В предыдущей главе мы пошли дальше и рассмотрели отображение на экране текста, состоящего из нескольких строк, и его прокрутку в окне. Теперь наступило время рассмотреть механизм отображения текста на экране более детально. Обсуждение задач, связанных с текстом, будет продолжено также в следующих главах. В главе 11 мы увидим, как использование библиотеки диалоговых окон общего пользования (Common Dialog Box library) значительно упрощает программы, давая пользователю возможность выбирать шрифты. В главе 15 мы исследуем проблемы отображения текста на экране в таком же виде, как на бумаге.

Вывод простого текста

Давайте сначала рассмотрим существующие в Windows для вывода текста различные функции, атрибуты контекста устройства, которые определяют вид текста, а также использование стандартных (stock) шрифтов.

Наиболее часто используемой функцией вывода текста является функция, которая использовалась в программе SYSMETS в главе 3:

```
TextOut(hdc, xStart, yStart, pString, iCount);
```

Параметры *xStart* и *yStart* определяют начальную позицию строки в логических координатах. Обычно это точка, в которую Windows помещает верхний левый угол первого символа. *TextOut* требует также в качестве параметра дальний указатель на символьную строку и длину строки. Эта функция не распознает текстовые строки по NULL символу.

Смысл параметров *xStart* и *yStart* функции *TextOut* может быть изменен с помощью функции *SetTextAlign*. Флаги TA_LEFT, TA_RIGHT и TA_CENTER влияют на использование *xStart* при позиционировании строки по горизонтали. По умолчанию установлен флаг TA_LEFT. Если вы установите флаг TA_RIGHT при вызове функции *SetTextAlign*, то последующие вызовы функции *TextOut* устанавливают правую границу последнего символа строки в *xStart*. При заданном флаге TA_CENTER в *xStart* устанавливается середина строки.

Аналогично, флаги TA_TOP, TA_BOTTOM и TA_BASELINE влияют на вертикальное позиционирование строки. По умолчанию установлен флаг TA_TOP, который означает, что строка позиционируется таким образом, что *yStart* определяет вершину символов в строке. Использование флага TA_BOTTOM означает, что строка позиционируется над *yStart*. Вы можете использовать флаг TA_BASELINE для размещения строки таким образом, чтобы положение базовой линии определялось значением *yStart*. Базовая линия — это линия, ниже которой располагаются "хвостики" некоторых строчных букв (например, р, q, у).

Если вы вызываете *SetTextAlign* с флагом TA_UPDATECP, Windows игнорирует параметры *xStart* и *yStart* функции *TextOut* и вместо них использует текущее положение пера, ранее установленное функциями *MoveToEx*, *LineTo* или какой-либо другой функцией, изменяющей текущее положение пера. Флаг TA_UPDATECP также заставляет функцию *TextOut* изменить значение текущего положения пера на конец строки (при установленном флаге TA_LEFT) или на начало строки (при установленном флаге TA_RIGHT). Это используется для отображения строки текста с помощью последовательных вызовов функции *TextOut*. Когда горизонтальное позиционирование осуществляется при установленном флаге TA_CENTER, текущее положение пера не меняется после вызова функции *TextOut*.

Теперь давайте вспомним, как осуществлялся вывод на экран текста в виде столбцов в ряде программ SYSMETS в главе 3. Тогда каждый новый вызов функции *TextOut* использовался для отображения на экране одного столбца. В качестве альтернативы можно использовать функцию *TabbedTextOut*:

```
TabbedTextOut(hdc, xStart, yStart, pString, iCount,
              iNumTabs, piTabStops, xTabOrigin);
```

Если строка символов содержит символы табуляции ('\t' или 0x09), то функция *TabbedTextOut* будет при выводе заменять символы табуляции числом пробелов, соответствующих списку целых параметров, которые вы передаете в функцию.

Первые пять параметров функции *TabbedTextOut* такие же, как у функции *TextOut*. Шестой параметр — число позиций табуляции, седьмой параметр — массив позиций табуляции, заданных в пикселях. Например, если средняя ширина символа 8 пикселей, и вы хотите установить позиции табуляции через каждые 5 символов, то этот список будет содержать числа 40, 80, 120 и т. д., в порядке возрастания.

Если шестой и седьмой параметры имеют значения 0 или NULL, то позиции табуляции устанавливаются через равные промежутки, равные восьмикратной средней ширине символов. Если шестой параметр равен 1, то седьмой параметр указывает на простое целое, которое каждый раз прибавляется для определения следующей позиции табуляции. (Например, если шестой параметр равен 1, а седьмой параметр является указателем на переменную, содержащую число 30, то позиции табуляции будут установлены так: 30, 60, 90, ... пикселей.) Последний параметр задает логическую координату по горизонтали точки отсчета позиций табуляции. Точка отсчета может совпадать с начальной позицией строки или отличаться от нее.

Примером другой расширенной функции вывода текста является функция *ExtTextOut* (приставка *Ext* означает расширенная):

```
ExtTextOut(hdc, xStart, yStart, iOptions, &rect, pString, iCount, pxDistance);
```

Пятый параметр этой функции является указателем на прямоугольную структуру. Эта структура является прямоугольником отсечения (если параметр *iOptions* имеет значение ETO_CLIPPED) или прямоугольником фона, который должен быть закрашен текущим цветом фона (если параметр *iOptions* имеет значение ETO_OPAQUE). Вы можете задавать обе опции или ни одной.

Последний параметр является массивом целых величин, задающих интервалы между соседними символами строки. Это позволяет программно сжимать или растягивать межсимвольный интервал, что иногда требуется для того, чтобы разместить отдельное слово в узком столбце. Если этот параметр имеет значение NULL, то устанавливается значение межсимвольного интервала по умолчанию.

Одной из функций вывода текста более высокого уровня является функция *DrawText*, которую мы использовали в программе HELLOWIN в главе 2. Вместо указания координат начальной позиции вы задаете структуру типа RECT, определяющую прямоугольник, в котором вы хотите разместить текст:

```
DrawText(hdc, pString, iCount, &rect, iFormat);
```

Так же, как и другие функции вывода текста, функция *DrawText* требует задания в качестве параметров дальнего указателя на символьную строку и длину строки. Однако, при использовании функции *DrawText* для вывода строки, оканчивающейся символом NULL, вы можете задать значение параметра *iCount* равным —1. В этом случае Windows вычислит длину строки.

Если параметр *iFormat* имеет значение 0, 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 интерпретируется как символ "новая строка" (newline). В соответствии с этим Windows прерывает вывод текущей строки и начинает новую строку. Вывод новой строки начинается под предыдущей строкой от левого края прямоугольника с интервалом равным высоте символа в строке (без учета величины пространства, заданного в шрифте в качестве межстрочного интервала (external leading)). Любой текст, в том числе и части букв, которые при отображении на экране попадают правее или ниже границ прямоугольника, отсекаются.

Вы можете изменить действие функции *DrawText* по умолчанию, задав значение параметра *iFormat*, как комбинацию одного или нескольких флагов. Флаг DT_LEFT (установлен по умолчанию) задает выравнивание выводимого текста влево, флаг DT_RIGHT — выравнивание вправо, флаг DT_CENTER — выравнивание по центру относительно левой и правой сторон прямоугольника. Поскольку флаг DT_LEFT имеет значение 0, вы можете не задавать его значение в явном виде, если хотите, чтобы весь выводимый текст был выровнен влево.

Если вы не хотите, чтобы символы возврата каретки и символы конца строки интерпретировались как символы начала новой строки, вы можете включить идентификатор DT_SINGLELINE. В этом случае Windows интерпретирует символы возврата каретки и конца строки как отображаемые символы, а не как управляющие символы. Если вы используете идентификатор DT_SINGLELINE, вам необходимо также задать положение строки по вертикали: вверху прямоугольника (флаг DT_TOP, включен по умолчанию), внизу прямоугольника (флаг DT_BOTTOM) или посередине между верхней и нижней границами прямоугольника вывода (флаг DT_VCENTER).

Когда на экран выводится текст, состоящий из нескольких строк, Windows в обычном режиме заканчивает строки, только встретив символ возврата каретки или символ конца строки. Однако, если строка оказывается длиннее, чем ширина прямоугольника вывода, вы можете использовать флаг DT_WORDBREAK. В этом случае Windows будет

обрывать строки в конце слов внутри строки. При выводе любого текста (состоящего из одной строки или многострочного) Windows отсекает ту часть текста, которая попадает за пределы прямоугольника вывода. Вы можете избежать этой ситуации, включив флаг DT_NOCLIP, который также ускоряет выполнение функции. Когда Windows осуществляет вывод на экран текста, состоящего из нескольких строк, то межстрочный интервал обычно выбирается равным высоте символа без учета величины пространства, заданного в шрифте, как межстрочный интервал. Если вы хотите, чтобы величина этого пространства была включена в межстрочный интервал, то вы можете использовать флаг DT_EXTERNALLEADING.

Если текст содержит символы табуляции ('\t' или 0x09), вам необходимо включить флаг DT_EXPANDTABS. По умолчанию позиции табуляции установлены через каждые восемь символьных позиций. Вы можете задать разные позиции табуляции, используя флаг DT_TABSTOP. В этом случае старший байт параметра *iFormat* содержит число символьных позиций для каждой новой позиции табуляции. Однако, здесь рекомендуется избегать использования флага DT_TABSTOP, поскольку старший байт параметра *iFormat* используется также для некоторых других флагов.

Атрибуты контекста устройства и текст

Некоторые атрибуты контекста устройства влияют на вид выводимого текста. В контексте устройства по умолчанию для текста задан черный цвет, но вы можете изменить его с помощью функции:

```
SetTextColor(hdc, rgbColor);
```

Так же, как для цвета пера и цвета штриховой кисти, Windows преобразует значение параметра *rgbColor* в чистый цвет. Вы можете определить цвет выводимого текста, вызывая функцию *GetTextColor*.

Пространство между строками текста закрашивается на основании установленных режима фона и цвета фона. Вы можете изменить режим фона, используя функцию:

```
SetBkMode(hdc, iMode);
```

где параметр *iMode* имеет значение OPAQUE или TRANSPARENT. По умолчанию режим фона установлен равным OPAQUE. Это означает, что Windows закрашивает пространство между строками текста цветом фона. Вы можете изменить цвет фона с помощью функции:

```
SetBkColor(hdc, rgbColor);
```

Значение параметра *rgbColor* преобразуется в значение чистого цвета. По умолчанию задан белый цвет фона. Если режим фона имеет значение TRANSPARENT, то Windows игнорирует цвет фона и не закрашивает пространство между строками. Windows также использует режим фона и цвет фона для закрашивания пространства между точечными и штриховыми линиями, а также пространства между штрихами штриховых кистей, как уже обсуждалось ранее.

Многие программы под Windows задают кисть WHITE_BRUSH, которую Windows использует для закрашивания фона окна. Кисть определяется в структуре класса окна. Кроме того, можно сделать цвет окна вашей программы совпадающим с системными цветами, которые пользователь может установить в программе Control Panel (панель управления). В этом случае вы задаете цвет фона в структуре WNDCLASS таким образом:

```
wndclass.hbrBackground =(HBRUSH)(COLOR_WINDOW + 1);
```

Когда вы хотите вывести текст в рабочей области, вы можете установить цвет текста и цвет фона, используя текущие системные цвета:

```
SetTextColor(hdc, GetSysColor(COLOR_WINDOWTEXT));
SetBkColor(hdc, GetSysColor(COLOR_WINDOW));
```

Можно сделать так, чтобы ваша программа отслеживала изменение системных цветов:

```
case WM_SYSCOLORCHANGE :
        InvalidateRect(hwnd, NULL, TRUE);
        break;
```

Другим атрибутом контекста устройства, определяющим вид текста, является межсимвольный интервал. По умолчанию он установлен в 0, что означает отсутствие интервала между соседними символами. Вы можете задать межсимвольный интервал, используя функцию:

```
SetTextCharacterExtra(hdc, iExtra);
```

Значение параметра *iExtra* задается в логических единицах. Windows преобразует его до ближайшего значения в пикселях, которое может быть и 0. Если вы зададите значение параметра *iExtra* отрицательным (например, чтобы прижать символы ближе друг к другу), Windows использует абсолютное значение заданного числа: вы не можете сделать эту величину меньше 0. Вы можете получить текущее значение межсимвольного интервала, используя

функцию *GetTextCharacterExtra*. Windows предварительно преобразует возвращаемое значение межсимвольного интервала из пикселей в логические единицы.

Использование стандартных шрифтов

Когда вы вызываете одну из функций вывода текста *TextOut*, *TabbedTextOut*, *ExtTextOut* или *DrawText*, Windows использует шрифт, выбранный в момент вызова функции в контексте устройства. Шрифт определяет особенности изображения символов и размер. Простейший путь выводить текст различными шрифтами состоит в использовании стандартных шрифтов, поддерживаемых Windows. Однако, их список достаточно ограничен.

Вы можете получить описатель стандартного шрифта, вызывая функцию:

```
hFont = GetStockObject(iFont);
```

где параметр *iFont* — один из нескольких идентификаторов, только два из которых обычно используются. Вы можете затем выбрать шрифт в контекст устройства:

```
SelectObject(hdc, hFont);
```

Вы можете осуществить это и за один шаг:

```
SelectObject(hdc, GetStockObject(iFont));
```

Функция *GetStockObject* — это та же самая функция, которую мы использовали ранее для получения стандартных перьев и кистей; функцию *SelectObject* мы использовали при выборе перьев, кистей, битовых образов и регионов в контекст устройства.

Шрифт, выбранный в контекст устройства по умолчанию, называется системным шрифтом и определяется параметром SYSTEM_FONT функции *GetStockObject*. Это пропорциональный шрифт, состоящий из ANSI символов, который Windows использует для вывода текста в меню, диалоговых окнах, окнах подсказок и в строках заголовков окон. Задание параметра SYSTEM_FIXED_FONT функции *GetStockObject* (которое было сделано в программе WHATSIZE ранее в этой главе) дает вам описатель шрифта фиксированной ширины (fixed-pitch), совместимого с системным шрифтом, который использовался в более ранних (до 3.0) версиях Windows. Это очень удобно, когда все символы шрифта имеют одинаковую ширину.

Когда вы выбираете новый шрифт в контекст устройства, вы должны вычислить высоту символа шрифта и среднюю ширину символа, используя функцию *GetTextMetrics*. Если вы выбираете пропорциональный шрифт, то вы должны помнить, что средняя ширина символа — это действительно среднее значение, т. е. некоторые символы более широкие, а некоторые более узкие. Далее в этой главе мы рассмотрим, как определить общую ширину строки, состоящую из символов различной ширины.

Несмотря на то, что функция *GetStockObject* несомненно предлагает простейший доступ к различным шрифтам, вы не имеете возможности управлять шрифтами, которые предлагает вам Windows. Вскоре вы увидите, каким образом вы можете точно задать желаемый вид и размер символов.

Типы шрифтов

Windows поддерживает две больших категории шрифтов — " шрифты GDI " и "шрифты устройства" (device fonts). Шрифты GDI хранятся в файлах на вашем жестком диске. Шрифты устройства соответствуют конкретному устройству вывода. Например, большинство принтеров имеет набор встроенных шрифтов устройства.

Шрифты GDI могут быть одного из трех типов — растровые шрифты, векторные шрифты и шрифты типа True Type.

Растровый шрифт иногда называют шрифтом битовых шаблонов, т. к. в файле растрового шрифта каждый символ хранится в виде битового шаблона. Каждый растровый шрифт разработан для определенного относительного размера пикселя дисплея и размера символа. Windows может создавать символы больших размеров из растровых шрифтов GDI, просто дублируя строки или колонки пикселей. Однако, это может быть сделано только целое количество раз и до определенного предела. По этой причине растровые шрифты GDI называют "немасштабируемыми" (nonscalable) шрифтами. Они не могут быть растянуты или сжаты до произвольного размера. Основными преимуществами растровых шрифтов являются их быстрое отображение на экране и четкость (т. к. они были разработаны вручную, чтобы текст выглядел максимально разборчиво).

В более ранних версиях (до 3.1) Windows поддерживала, кроме шрифтов GDI, еще и векторные шрифты. Векторные шрифты определены как набор соединенных друг с другом отрезков прямых (connect-the-dots). Векторные шрифты легко масштабируются в широких пределах, т. е. один и тот же шрифт может быть использован в графических устройствах вывода с любой разрешающей способностью, и эти шрифты могут быть увеличены или уменьшены до любого размера. Однако, эти шрифты имеют более низкую скорость отображения, плохую четкость при маленьких размерах, а при использовании больших размеров символы выглядят очень

бледными, потому что их контуры — тонкие линии. Векторные шрифты сейчас иногда называют "плоттерными" (plotter fonts), поскольку они подходят только для плоттеров.

Для обоих типов шрифтов GDI, растровых и векторных, Windows может синтезировать полужирный курсив, подчеркнутый и зачеркнутый шрифты без хранения отдельно шрифтов каждого из этих типов. Например, чтобы получить курсив, Windows просто сдвигает верхнюю часть символа вправо.

Далее рассмотрим шрифты TrueType, которым посвятим остаток главы.

Шрифты TrueType

С введением шрифтов 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; она поддерживается многими производителями шрифтов. Отдельные символы шрифтов TrueType определяются контурами, состоящими из прямых линий и кривых. Таким образом, Windows может масштабировать эти шрифты, изменяя определяющие контур координаты. Шрифты TrueType могут быть использованы как для вывода на экран, так и для вывода на принтер, делая реально возможным режим отображения текста WYSIWYG (what-you-see-is-what-you-get, что видите — то и получаете).

Когда вашей программе необходимо использовать шрифт TrueType определенного размера, Windows формирует растровое представление символов этого шрифта. Это означает, что Windows масштабирует координаты точек соединения прямых и кривых каждого символа, используя дополнительные данные, которые включены в файл шрифта TrueType. Эти дополнительные данные позволяют компенсировать ошибки округления, которые могли бы привести в результате к искажению символа. (Например, в некоторых шрифтах обе ножки заглавной буквы Н должны иметь одинаковую ширину. Простое масштабирование может привести к результату, когда одна ножка будет на пиксель шире, чем другая. Учет дополнительных данных шрифта предотвращает эту ситуацию.) Полученный таким образом контур каждого символа используется затем для создания битового образа символа. Эти битовые образы кэшируются в памяти для последующего применения.

Windows 95 оснащена 13 шрифтами TrueType. Они имеют следующие имена в соответствии с их видом:

- Courier New
- Courier New Bold
- Courier New Italic
- Courier New Bold Italic
- Times New Roman
- Times New Roman Bold
- Times New Roman Italic
- Times New Roman Bold Italic
- Arial
- Arial Bold
- Arial Italic
- Arial Bold Italic
- Symbol

Шрифт Courier New — это фиксированный шрифт (т. е. все символы имеют одинаковую ширину). Он разработан похожим на выводимые данные такого устаревшего устройства, как пишущая машинка. Группа шрифтов Times New Roman — это производные от шрифта Times, впервые разработанного специально для *Times of London*, и используемого во многих печатных материалах. Они рассчитаны на то, чтобы обеспечить максимальное удобство чтения. Группа шрифтов Arial — это производные от шрифта Helvetica, являющегося рубленым (sans serif) шрифтом. Это означает, что символы не имеют засечек на концах. Шрифт Symbol содержит ряд часто используемых специальных символов.

Обычно вы задаете шрифт путем указания его типового имени и типового размера. Типовой размер выражается в единицах, называемых пунктами. Пункт — это приблизительно 1/72 дюйма, разница настолько несущественна, что в компьютерной полиграфии пункт часто определяют как 1/72 дюйма без всяких оговорок. Текст этой книги напечатан шрифтом типового размера 12 пунктов. Типовой размер иногда определяют как высоту символа от верхней границы букв, имеющих выступающую верхнюю часть, до нижней границы букв, имеющих выступающую нижнюю часть. Это достаточно убедительный способ оценки типового размера, но он не является точным для всех шрифтов. Иногда разработчики шрифтов действуют по-другому.

Система EZFONT

Введение шрифтов TrueType — и их основополагающее использование в традиционной полиграфии — обеспечило Windows серьезную базу для многовариантного отображения текста. Однако, некоторые из функций выбора

шрифта Windows основаны на более старой технологии, при которой растровые шрифты на экране должны были аппроксимировать шрифты печатающего устройства.

В предыдущих изданиях этой книги подробно рассматривалась функция *EnumFonts*, которая предоставляет любой программе структуры LOGFONT (logical font, логический шрифт) и TEXTMETRIC для каждого шрифта GDI, установленного для использования под Windows, и для всех шрифтов устройств. Диалоговое окно *ChooseFont* (которое подробнее будет рассмотрено в главе 11) делает ненужным использование этой функции и всего, что с ней связано. Также подробно обсуждалась функция *CreateFontIndirect*, которая позволяет программе описывать создаваемый шрифт без его немедленного именования. Это объясняет то, каким образом шрифты печатающего устройства аппроксимировались на экране монитора.

В этой главе вам будет предложено нечто совсем другое. Здесь будет показано, как можно использовать различные стандартные шрифты TrueType в ваших программах наиболее эффективно и одновременно с учетом требований традиционной полиграфии. Для этого надо задать название шрифта (одного из 13, перечисленных выше) и его размер (который будет рассмотрен вскоре). Шрифт назван EZFONT (easy font, простой шрифт). На рис. 4.33 приведены два файла, тексты которых вам потребуются.

EZFONT.H

```
/*----------------------
   EZFONT.H header file
  ----------------------*/

HFONT EzCreateFont(HDC hdc, char * szFaceName, int iDeciPtHeight,
                    int iDeciPtWidth, int iAttributes, BOOL fLogRes);

#define EZ_ATTR_BOLD          1
#define EZ_ATTR_ITALIC        2
#define EZ_ATTR_UNDERLINE     4
#define EZ_ATTR_STRIKEOUT     8
```

EZFONT.C

```
/*---------------------------------------
   EZFONT.C -- Easy Font Creation
              (c) Charles Petzold, 1996
  ---------------------------------------*/

#include <windows.h>
#include <string.h>
#include <math.h>
#include "ezfont.h"

HFONT EzCreateFont(HDC hdc, char * szFaceName, int iDeciPtHeight,
                    int iDeciPtWidth, int iAttributes, BOOL fLogRes)
     {
     FLOAT      cxDpi, cyDpi;
     HFONT      hFont;
     LOGFONT    lf;
     POINT      pt;
     TEXTMETRIC tm;

     SaveDC(hdc);

     SetGraphicsMode(hdc, GM_ADVANCED);
     ModifyWorldTransform(hdc, NULL, MWT_IDENTITY);
     SetViewportOrgEx(hdc, 0, 0, NULL);
     SetWindowOrgEx  (hdc, 0, 0, NULL);

     if(fLogRes)
          {
          cxDpi =(FLOAT) GetDeviceCaps(hdc, LOGPIXELSX);
          cyDpi =(FLOAT) GetDeviceCaps(hdc, LOGPIXELSY);
          }
     else
          {
          cxDpi =(FLOAT)(25.4 * GetDeviceCaps(hdc, HORZRES) /
```

```
                              GetDeviceCaps(hdc, HORZSIZE));

          cyDpi =(FLOAT)(25.4 * GetDeviceCaps(hdc, VERTRES) /
                              GetDeviceCaps(hdc, VERTSIZE));
          }

     pt.x =(int)(iDeciPtWidth  * cxDpi / 72);
     pt.y =(int)(iDeciPtHeight * cyDpi / 72);

     DPtoLP(hdc, &pt, 1);

     lf.lfHeight         = -(int)(fabs(pt.y) / 10.0 + 0.5);
     lf.lfWidth          = 0;
     lf.lfEscapement     = 0;
     lf.lfOrientation    = 0;
     lf.lfWeight         = iAttributes & EZ_ATTR_BOLD      ? 700 : 0;
     lf.lfItalic         = iAttributes & EZ_ATTR_ITALIC    ?   1 : 0;
     lf.lfUnderline      = iAttributes & EZ_ATTR_UNDERLINE ?   1 : 0;
     lf.lfStrikeOut      = iAttributes & EZ_ATTR_STRIKEOUT ?   1 : 0;
     lf.lfCharSet        = 0;
     lf.lfOutPrecision   = 0;
     lf.lfClipPrecision  = 0;
     lf.lfQuality        = 0;
     lf.lfPitchAndFamily = 0;

     strcpy(lf.lfFaceName, szFaceName);

     hFont = CreateFontIndirect(&lf);

     if(iDeciPtWidth != 0)
          {
          hFont =(HFONT) SelectObject(hdc, hFont);

          GetTextMetrics(hdc, &tm);

          DeleteObject(SelectObject(hdc, hFont));

          lf.lfWidth =(int)(tm.tmAveCharWidth *
                              fabs(pt.x) / fabs(pt.y) + 0.5);

          hFont = CreateFontIndirect(&lf);
          }

     RestoreDC(hdc, -1);

     return hFont;
     }
```

Рис. 4.33 Файлы EZFONT

Программа EZFONT.C содержит только одну функцию — *EzCreateFont* — которую вы можете использовать, например, таким образом:

```
hFont = EzCreateFont(hdc, szFaceName, iDeciPtHeight, iDeciPtWidth, iAttributes, fLogRes);
```

Функция возвращает описатель шрифта. Шрифт может быть выбран в контекст устройства посредством вызова функции *SelectObject*. Затем вы можете вызывать функции *GetTextMetrics* или *GetOutlineTextMetrics* для определения действительного размера шрифта в логических координатах. Перед окончанием вашей программы необходимо удалить все созданные шрифты, вызвав функцию *DeleteObject*.

Параметр *szFaceName* — одно из 13 типовых имен шрифта TrueType из приведенного ранее списка. Если в вашей системе есть другие шрифты TrueType, вы можете также использовать их названия, но только 13 шрифтов, перечисленных ранее, обязательно присутствуют во всех системах Windows 95.

Третий параметр определяет желаемый размер шрифта в пунктах, но его особенность состоит в том, что он задается в деципунктах (каждый деципункт равен 1/10 пункта). Следовательно, если вы хотите задать размер в пунктах, равный $12^1/_2$, используйте значение 125.

Обычно четвертый параметр должен быть установлен в ноль или быть таким же, как третий параметр. Однако, вы можете создать более широкий или более узкий шрифт TrueType, установив другое значение этого параметра. Иногда этот размер называют "эм-шириной" (em-width) шрифта, и он описывает ширину шрифта в пунктах. Не путайте эту величину со средней шириной символов шрифта или с чем-нибудь похожим. На раннем этапе развития типографского дела заглавная буква 'M' имела одинаковые ширину и высоту. Так возникла концепция "эм-квадрат", а впоследствии и такая мера как "эм-ширина". Когда эм-ширина равна эм-высоте (размеру шрифта в пунктах), то ширины символов установлены такими, какими изначально задумывались разработчиком шрифта. Задание большей или меньшей эм-ширины позволяет вам создавать более широкие или более узкие символы.

Параметр *iAttributes* может быть установлен в одно из следующих значений, определенных в EZFONT.H:

- EZ_ATTR_BOLD
- EZ_ATTR_ITALIC
- EZ_ATTR_UNDERLINE
- EZ_ATTR_STRIKEOUT

Может быть вам и не потребуется использовать значения EZ_ATTR_BOLD или EZ_ATTR_ITALIC, потому что эти атрибуты являются частью полного типового имени шрифта TrueType. Если вы их используете, то Windows синтезирует соответствующие эффекты.

Наконец, вы должны установить в TRUE значение последнего параметра для того, чтобы видимый размер шрифта основывался на логическом разрешении (logical resolution), возвращаемом функцией *GetDeviceCaps*. В противном случае он базируется на действительном разрешении.

Внутренняя работа

Функция *EzCreateFont* разработана для использования в системах Windows 95 или Windows NT. Для совместимости с NT используются функции *SetGraphicsMode* и *ModifyWorldTransform*, которые никак не влияют на работу Windows 95. (Другими словами, *ModifyWorldTransform* в Windows NT могла бы оказать влияние на видимый размер шрифта. Поэтому эта функция устанавливается в режим по умолчанию — без преобразования — перед тем, как вычисляется размер шрифта.)

Функция *EzCreateFont* сначала устанавливает поля структуры LOGFONT и вызывает функцию *CreateFont*, которая возвращает описатель шрифта. Для выбранных шрифтов TrueType большинство полей может быть установлено в ноль. Вам необходимо установить значения следующих полей:

- *lfHeight* — это желаемая высота символов (включая поле, отведенное для специальных знаков над символами, но не включая поле, установленное для межстрочного интервала) в логических единицах. Поскольку размер шрифта в пунктах — это и есть высота шрифта без величины поля, отведенного для специальных знаков над символами, то здесь вы на самом деле определяете значение межстрочного интервала. Вы можете установить значение *lfHeight* равным 0 для задания размера по умолчанию. Если вы зададите значение *lfHeight* отрицательным числом, Windows использует абсолютное значение этого числа в качестве желаемого размера высоты шрифта, а не как межстрочный интервал. Если вы хотите задать конкретное значение размера в пунктах, его надо преобразовать в логические единицы, и поле *lfHeight* установить в отрицательное значение результата.
- *lfWidth* — это желаемая ширина символов в логических единицах. В большинстве случаев его устанавливают в 0 и позволяют Windows выбирать шрифт, основываясь исключительно на высоте.
- *lfWeight* — это поле позволяет вам задать полужирный шрифт путем установки значения, равного 700.
- *lfItalic* — ненулевое значение поля определяет курсив.
- *lfUnderline* — ненулевое значение поля задает подчеркивание.
- *lfStrikeOut* — ненулевое значение поля определяет шрифт с зачеркиванием символов.
- *lfFaceName* (массив типа BYTE) — это имя шрифта (например, Courier New, Arial или Times New Roman).

Одним из действий, выполняемых функцией *EzCreateFont*, является преобразование размера в пунктах в логические единицы для последующей установки этого значения в структуре LOGFONT. Сначала размер в пунктах должен быть преобразован в единицы устройства (пиксели), а затем в логические единицы. Чтобы выполнить этот первый шаг, мы используем информацию, получаемую от функции *GetDeviceCaps*.

Вызов функции *GetDeviceCaps* с параметрами HORZRES и VERTRES дает нам ширину и высоту экрана (или области печати выводимой на принтере страницы) в пикселях. Вызов функции *GetDeviceCaps* с параметрами HORZSIZE и VERTSIZE дает нам физическую ширину и высоту экрана (или области печати выводимой на

принтере страницы) в миллиметрах. Если последний параметр имеет значение FALSE, то функция *EzCreateFont* использует эти значения для получения разрешающей способности устройств в точках на дюйм.

Здесь возможны два варианта. Вы можете получить разрешение устройства в точках на дюйм непосредственно, используя параметры LOGPIXELSX и LOGPIXELSY в функции *GetDeviceCaps*. Это называется "логическим разрешением" (logical resolution) устройства. Для печатающих устройств нормальное разрешение и логическое разрешение одинаковы (отбрасывая ошибки округления). Для дисплеев, однако, логическое разрешение лучше, чем нормальное разрешение. Например, для VGA наилучшим значением нормального разрешения является примерно 68 точек на дюйм. В то время, как логическое разрешение — 96 точек на дюйм.

Это различие, можно сказать, драматическое. Предположим, что мы работаем со шрифтом размера 12 пунктов, который имеет высоту 1/6 дюйма. Предположив, что нормальное разрешение равно 68 точкам на дюйм, полная высота символов будет около 11 пикселей. При логическом разрешении 96 точек на дюйм эта величина будет 16 пикселей. То есть разница составляет около 45%.

Чем объясняется такая разница? Если немного задуматься над этим, то на самом деле не существует истинного разрешения VGA. Стандартный VGA показывает 640 пикселей по горизонтали и 480 пикселей по вертикали, но размер экрана реального компьютера может быть различным — от маленького в компьютерах notebook до большого в VGA — проекторах. Windows не имеет способа самостоятельно определить действительный размер экрана. Значения HORZSIZE и VERTSIZE основаны на стандартных размерах настольных (desktop) дисплеев VGA, которые могли быть установлены для каких-нибудь ранних моделей IBM (году в 1987) путем простого измерения линейкой экрана каким-то программистом Microsoft.

Если же рассмотреть этот вопрос еще глубже, то в действительности вам и не нужно, чтобы шрифты были отображены на экране в их истинном размере. Предположим, что вы используете VGA проектор на презентации перед сотнями людей, и вы используете 12-пунктовый шрифт, реальный размер которого составляет 1/6 дюйма на проекционном экране. Нет сомнений, что ваша аудитория будет в замешательстве.

Люди, постоянно работающие с текстами, часто используют текстовые процессоры и настольные издательские системы. Довольно часто вывод осуществляется на бумаге размером $8^1/_2$x11 дюймов (или 8x10 дюймов с учетом отступов). Многие дисплеи VGA шире, чем 8 дюймов. Отображение на экране более крупных символов предпочтительнее, чем изображение в реальных размерах на экране.

Однако, если вы используете логическое разрешение шрифта, то могут возникнуть проблемы при совмещении текста и другой графики. Если вы используете функцию *SetMapMode* для рисования графики в дюймах или миллиметрах и одновременно логическое разрешение устройства для установки размера шрифта, то вы придете к несоответствию — не при выводе на принтере (т. к. здесь нормальное разрешение совпадает с логическим), а при выводе на экран, где существует разница в 45%. Решение этой проблемы будет продемонстрировано далее в этой главе в программе JUSTIFY1.

Структура LOGFONT, которую вы передаете функции *CreateFontIndirect*, требует задания высоты шрифта в логических единицах. Однажды получив это значение в пикселях, вы легко преобразуете его в логические единицы посредством вызова функции *DPtoLP* (device point to logical point, точка устройства в логическую точку). Но для того чтобы преобразование *DPtoLP* выполнялось правильно, должен быть установлен тот же режим отображения (mapping mode), с каким вы далее будете работать при отображении текста на экране, используя созданный шрифт. Это значит, что вы должны установить режим отображения до того, как будете вызывать функцию *EzCreateFont*. В большинстве случаев вы используете только один режим отображения для рисования в конкретной области окна, так что выполнение этого требования не является проблемой.

Форматирование простого текста

Разобравшись с файлами EZFONT, наступило время потренироваться в форматировании текста. Процесс форматирования заключается в расположении каждой строки текста в пределах установленных полей одним из четырех способов: с выравниванием влево, с выравниванием вправо, с выравниванием по центру или с выравниванием по всей ширине (когда строка растягивается от левого края до правого края с формированием одинаковых интервалов между словами). Первые три задачи можно решить, используя функцию *DrawText* с параметром DT_WORDBREAK, но ее использование ограничено. Например, вы не можете определить, какую часть текста функция *DrawText* сможет разместить в прямоугольнике вывода. Функция *DrawText* удобна для некоторых простых задач, но для более сложных задач форматирования вы, вероятно, захотите применить функцию *TextOut*.

Одной из наиболее часто используемых функций работы с текстом является функция *GetTextExtentPoint32*. (Это функция, имя которой претерпело некоторые изменения со времени более ранних версий Windows.) Эта функция дает значения ширины и высоты строки символов, основываясь на текущем шрифте, выбранном в контексте устройства:

```
GetTextExtentPoint32(hdc, pString, iCount, &size);
```

Ширина и высота текста в логических единицах возвращается в поля *cx* и *cy* структуры SIZE. Начнем с примера, использующего одну строку текста. Предположим, что вы выбрали шрифт в контексте устройства и теперь хотите вывести текст:

```
char *szText [ ] = "Hello, how are you?";
```

Вам нужно, чтобы текст начинался в позиции с вертикальной координатой *yStart* и находился между границами, установленными координатами *xLeft* и *xRight*. Ваша задача заключается в том, чтобы вычислить значение *xStart* горизонтальной координаты начала текста. Эта задача была бы значительно проще, если бы текст отображался с использованием шрифта фиксированной ширины, но это не является общим случаем. Сначала определим длину строки текста:

```
GetTextExtentPoint32(hdc, szText, strlen(szText), &size);
```

Если значение *size.cx* больше, чем (*xRight* — *xLeft*), то срока слишком длинна, чтобы поместиться в указанных границах. Предположим, что строка все же помещается.

Для того, чтобы выровнять текст влево, нужно просто установить значение *xStart* равным *xLeft* и затем вывести текст:

```
TextOut(hdc, xStart, yStart, szText, strlen(szText));
```

Это просто. Теперь вы можете прибавить значение *size.cy* к *yStart* и затем выводить следующую строку текста.

Чтобы выровнять текст вправо, воспользуемся для вычисления *xStart* следующей формулой:

```
xStart = xRight — size.cx;
```

Чтобы выровнять текст по центру между левой и правой границами используем другую формулу:

```
xStart =(xLeft + xRight — size.cx) / 2;
```

Теперь перед нами самая трудная задача — выровнять текст по всей ширине между левой и правой границами. Расстояние между ними вычисляется как (*xRight* — *xLeft*). Без растягивания по ширине текст имеет ширину *size.cx*. Разница между этими двумя величинами:

```
xRight — xLeft — size.cx
```

должна быть равномерно распределена между тремя символами пробелов в символьной строке. На первый взгляд это кажется труднейшей задачей, но на самом деле это не так уж и трудно. Чтобы сделать это, вы вызываете функцию:

```
SetTextJustification(hdc, xRight — xLeft — size.cx, 3)
```

Второй параметр — это величина пробела, который должен быть распределен поровну между тремя символами пробела в символьной строке. Третий параметр — число символов пробела в строке, в нашем примере — 3.

Теперь установим значение *xStart* равным *xLeft* и выведем текст с помощью функции *TextOut*:

```
TextOut(hdc, xStart, yStart, szText, strlen(szText));
```

Текст будет выровнен по всей ширине между границами *xLeft* и *xRight*.

При каждом вызове функции *SetTextJustification* накапливается погрешность, если величина добавляемого пропуска не делится нацело на число символов пробела. Эта ошибка будет влиять на последующие вызовы функции *GetTextExtent*. Каждый раз перед тем, как начинать вывод новой строки, вы должны сбросить эту погрешность с помощью вызова функции:

```
SetTextJustification(hdc, 0, 0);
```

Работа с абзацами

Если вы работаете с целым абзацем, вы должны просматривать всю строку от начала, отыскивая символы пробела. Каждый раз, как вы встречаете символ пробела, вы вызываете функцию *GettextExtentPoint32*, чтобы определить, умещается ли текст по ширине между левой и правой границами. Когда текст не укладывается в отведенное для него пространство, вы возвращаетесь на один шаг назад к предыдущему найденному символу пробела. Теперь у вас есть законченная строка символов. Если вы хотите выровнять текст по ширине, то нужно вызвать функции *SetTextJustification* и *TextOut*, избавиться от ошибки и перейти к следующей строке.

Программа JUSTIFY1, приведенная на рис. 4.34, проделывает эту процедуру с первым параграфом книги автора Herman Melville, которая называется "Moby Dick". Программа использует встроенный шрифт Times New Roman размера 15 пунктов, но вы можете изменить его в функции *MyCreateFont*, описанной в начале программы, и перекомпилировать саму программу. Вы также можете изменить тип выравнивания, используя определение в начале программы. На рис. 4.35 показан вид текста, выведенного программой JUSTIFY1 на экран.

JUSTIFY1.MAK

```
#------------------------
# JUSTIFY1.MAK make file
#------------------------

justify1.exe : justify1.obj ezfont.obj
     $(LINKER) $(GUIFLAGS) -OUT:justify1.exe justify1.obj ezfont.obj $(GUILIBS)

justify1.obj : justify1.c
     $(CC) $(CFLAGS) justify1.c

ezfont.obj : ezfont.c
     $(CC) $(CFLAGS) ezfont.c
```

JUSTIFY1.C

```
/*-----------------------------------------
   JUSTIFY1.C -- Justified Type Program
                 (c) Charles Petzold, 1996
  -----------------------------------------*/

#include <windows.h>
#include "ezfont.h"

#define LEFT        0
#define RIGHT       1
#define CENTER      2
#define JUSTIFIED   3

#define ALIGN       JUSTIFIED

#define MyCreateFont EzCreateFont(hdc, "Times New Roman", 150, 0, 0, TRUE)

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static     char szAppName[] = "Justify1";
     HWND       hwnd;
     MSG        msg;
     WNDCLASSEX wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = szAppName;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);
     hwnd = CreateWindow(szAppName, "Justified Type",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
```

```
     UpdateWindow(hwnd);

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

void DrawRuler(HDC hdc, RECT *prc)
     {
     static int iRuleSize [16] = { 360, 72, 144, 72, 216, 72, 144, 72,
                                   288, 72, 144, 72, 216, 72, 144, 72 };
     int        i, j;
     POINT      ptClient;

     SaveDC(hdc);
                    // Set Logical Twips mapping mode

     SetMapMode(hdc, MM_ANISOTROPIC);
     SetWindowExtEx(hdc, 1440, 1440, NULL);
     SetViewportExtEx(hdc, GetDeviceCaps(hdc, LOGPIXELSX),
                           GetDeviceCaps(hdc, LOGPIXELSY), NULL);

                    // Move the origin to a half inch from upper left

     SetWindowOrgEx(hdc, -720, -720, NULL);

                    // Find the right margin(quarter inch from right)

     ptClient.x = prc->right;
     ptClient.y = prc->bottom;
     DPtoLP(hdc, &ptClient, 1);
     ptClient.x -= 360;

                    // Draw the rulers

     MoveToEx(hdc, 0,          -360, NULL);
     LineTo  (hdc, ptClient.x, -360);
     MoveToEx(hdc, -360,          0, NULL);
     LineTo  (hdc, -360, ptClient.y);
     for(i = 0, j = 0; i <= ptClient.x; i += 1440 / 16, j++)
          {
          MoveToEx(hdc, i, -360, NULL);
          LineTo  (hdc, i, -360 - iRuleSize [j % 16]);
          }

     for(i = 0, j = 0; i <= ptClient.y; i += 1440 / 16, j++)
          {
          MoveToEx(hdc, -360, i, NULL);
          LineTo  (hdc, -360 - iRuleSize [j % 16], i);
          }

     RestoreDC(hdc, -1);
     }

void Justify(HDC hdc, PSTR pText, RECT *prc, int iAlign)
     {
     int  xStart, yStart, iBreakCount;
     PSTR pBegin, pEnd;
     SIZE size;

     yStart = prc->top;
```

```
do                           // for each text line
     {
     iBreakCount = 0;
     while(*pText == ' ')    // skip over leading blanks

          pText++;
     pBegin = pText;

     do                      // until the line is known
          {
          pEnd = pText;

          while(*pText != '\0' && *pText++ != ' ');
          if(*pText == '\0')
               break;
                             // for each space, calculate extents
          iBreakCount++;
          SetTextJustification(hdc, 0, 0);
          GetTextExtentPoint32(hdc, pBegin, pText - pBegin - 1, &size);
          }
     while((int) size.cx <(prc->right - prc->left));

     iBreakCount--;
     while(*(pEnd - 1) == ' ')   // eliminate trailing blanks
          {
          pEnd--;
          iBreakCount--;
          }
     if(*pText == '\0' || iBreakCount <= 0)
          pEnd = pText;

     SetTextJustification(hdc, 0, 0);
     GetTextExtentPoint32(hdc, pBegin, pEnd - pBegin, &size);


     switch(iAlign)               // use alignment for xStart
          {
          case LEFT:
               xStart = prc->left;
               break;

          case RIGHT:
               xStart = prc->right - size.cx;
               break;

          case CENTER:
               xStart =(prc->right + prc->left - size.cx) / 2;
               break;

          case JUSTIFIED:
               if(*pText != '\0' && iBreakCount > 0)
                    SetTextJustification(hdc,
                         prc->right - prc->left - size.cx,
                         iBreakCount);
               xStart = prc->left;
               break;
          }

     TextOut(hdc, xStart, yStart, pBegin, pEnd - pBegin);
     yStart += size.cy;
     pText = pEnd;
     }
while(*pText && yStart < prc->bottom);
```

```
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char szText[] = "Call me Ishmael. Some years ago -- never mind "
                            "how long precisely -- having little or no money "
                            "in my purse, and nothing particular to interest "
                            "me on shore, I thought I would sail about a "
                            "little and see the watery part of the world. It "
                            "is a way I have of driving off the spleen, and "
                            "regulating the circulation.  Whenever I find "
                            "myself growing grim about the mouth; whenever "
                            "it is a damp, drizzly November in my soul; "
                            "whenever I find myself involuntarily pausing "
                            "before coffin warehouses, and bringing up the "
                            "rear of every funeral I meet; and especially "
                            "whenever my hypos get such an upper hand of me, "
                            "that it requires a strong moral principle to "
                            "prevent me from deliberately stepping into the "
                            "street, and methodically knocking people's hats "
                            "off -- then, I account it high time to get to sea "
                            "as soon as I can. This is my substitute for "
                            "pistol and ball.  With a philosophical flourish "
                            "Cato throws himself upon his sword; I quietly "
                            "take to the ship. There is nothing surprising "
                            "in this. If they but knew it, almost all men in "
                            "their degree, some time or other, cherish very "
                            "nearly the same feelings towards the ocean with "
                            "me.";

     HDC         hdc;
     PAINTSTRUCT ps;
     RECT        rcClient;

     switch(iMsg)
          {
          case WM_PAINT:
               hdc = BeginPaint(hwnd, &ps);

               GetClientRect(hwnd, &rcClient);
               DrawRuler(hdc, &rcClient);

               rcClient.left  += GetDeviceCaps(hdc, LOGPIXELSX) / 2;
               rcClient.top   += GetDeviceCaps(hdc, LOGPIXELSY) / 2;
               rcClient.right -= GetDeviceCaps(hdc, LOGPIXELSX) / 4;

               SelectObject(hdc, MyCreateFont);

               Justify(hdc, szText, &rcClient, ALIGN);

               DeleteObject(SelectObject(hdc, GetStockObject(SYSTEM_FONT)));
               EndPaint(hwnd, &ps);
               return 0;

          case WM_DESTROY:
               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

Рис. 4.34 Файлы JUSTIFY1

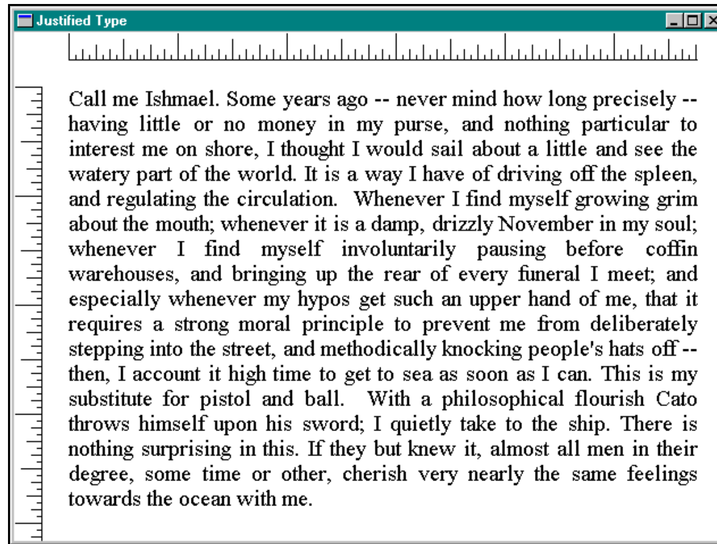


Рис. 4.35 Вывод программы JUSTIFY1

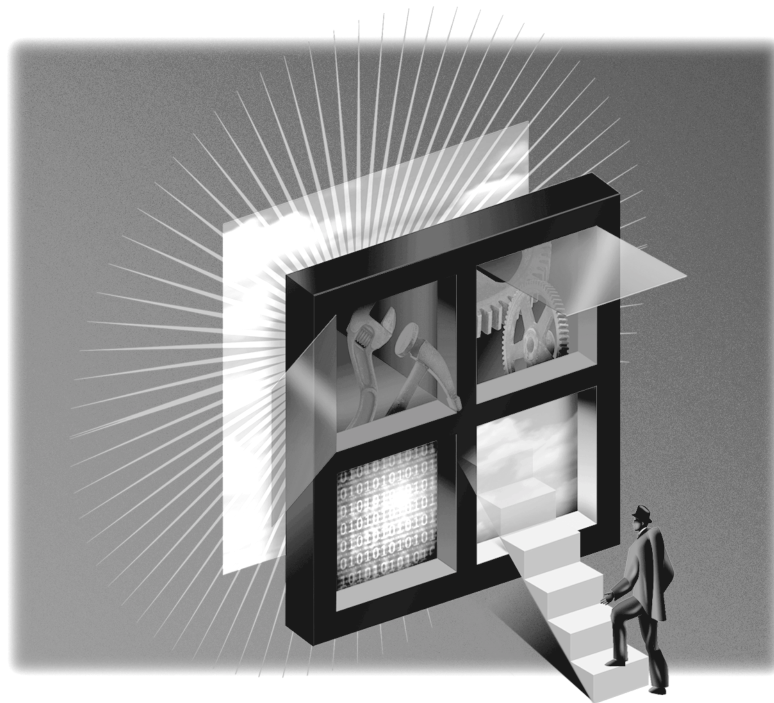
Программа JUSTIFY1 выводит на экран линейки (в логических дюймах, конечно) вдоль верхнего края и вдоль левого края рабочей области. Функция *DrawRuler* рисует линейки. Структура прямоугольника определяет область, в которой может быть расположен текст.

Основная работа заключается в форматировании заданного в программе JUSTIFY1 текста. Выполнение программы JUSTIFY1 начинается с поиска всех символов пробела от начала текста. При помощи функции *GetTextExtentPoint32* измеряется длина каждой строки. Когда длина строки превышает ширину области вывода, программа JUSTIFY1 возвращается к предыдущему символу пробела и заканчивает строку в этой позиции. В зависимости от значения константы ALIGN строка выравнивается влево, выравнивается вправо, выравнивается по центру или выравнивается по ширине.

Эта программа не является совершенной. В частности, выравнивание по ширине логически бессмысленно, когда в каждой строке меньше, чем два слова. Но даже если мы решим этот вопрос (что не особенно сложно), программа все еще не будет работать как следует, если одно слово будет слишком длинным, чтобы уместиться между заданными полями. Конечно, ситуация может стать более сложной, если вы работаете с программами, использующими несколько шрифтов в одной строке (как с кажущейся легкостью делают это текстовые процессоры в Windows). Но никто никогда и не требовал, чтобы это было легко. Это проще только по сравнению с самостоятельным выполнением той же работы.

Часть II

Средства ввода



Глава 5 Клавиатура

5

Как и большинство интерактивных программ, работающих на персональных компьютерах, приложения Windows 95 активно используют пользовательский ввод с клавиатуры. Хотя Windows поддерживает в качестве устройства ввода также и мышь, работа с клавиатурой по-прежнему превалирует. Действительно, поскольку некоторые пользователи персональных компьютеров предпочитают пользоваться не мышью, а клавиатурой, рекомендуется, чтобы создатели программ пытались реализовать все функциональные возможности программы с помощью клавиатуры. (Конечно, в некоторых случаях, таких как программы рисования или издательские системы, это просто не практикуется, и в этом случае необходима мышь.)

Клавиатура не может рассматриваться исключительно как устройство для ввода информации, изолированно от других функций программы. Например, программы часто повторяют ввод с клавиатуры путем отображения печатаемых символов в рабочей области окна. Таким образом, обработка ввода с клавиатуры и вывод текста должны рассматриваться совместно. Если вам важно адаптировать ваши программы к иностранным языкам и рынкам, вам также нужно знать о том, как Windows 95 поддерживает расширенный набор символов ASCII (коды от 128 и выше), двухбайтные наборы символов (DBCS), и поддерживаемую Windows NT 16-разрядную кодировку клавиатуры, известную как Unicode.

Клавиатура. Основные понятия

Как вы, наверное, догадались, основанная на сообщениях архитектура Windows идеальна для работы с клавиатурой. Ваша программа узнает о нажатиях клавиш посредством сообщений, которые посылаются оконной процедуре.

На самом деле все происходит не столь просто: когда пользователь нажимает и отпускает клавиши, драйвер клавиатуры передает информацию о нажатии клавиш в Windows. Windows сохраняет эту информацию (в виде сообщений) в системной очереди сообщений. Затем она передает сообщения клавиатуры, по одному за раз, в очередь сообщений программы, содержащей окно, имеющее "фокус ввода" (input focus) (о котором вскоре будет рассказано). Затем программа отправляет сообщения соответствующей оконной процедуре.

Смысл этого двухступенчатого процесса — сохранение сообщений в системной очереди сообщений, и дальнейшая их передача в очередь сообщений приложения — в синхронизации. Если пользователь печатает на клавиатуре быстрее, чем программа может обрабатывать поступающую информацию, Windows сохраняет информацию о дополнительных нажатиях клавиш в системной очереди сообщений, поскольку одно из этих дополнительных нажатий может быть переключением фокуса ввода на другую программу. Информацию о последующих нажатиях следует затем направлять в другую программу. Таким образом Windows корректно синхронизирует такие сообщения клавиатуры.

Для отражения различных событий клавиатуры, Windows посылает программам восемь различных сообщений. Это может показаться излишним, и ваша программа вполне может игнорировать многие из них. Кроме того, в большинстве случаев, в этих сообщениях от клавиатуры содержится значительно больше закодированной информации, чем нужно вашей программе. Залог успешной работы с клавиатурой — это знание того, какие сообщения важны, а какие нет.

Игнорирование клавиатуры

Хотя клавиатура является основным источником пользовательского ввода программ для Windows, вашей программе не нужно реагировать на каждое получаемое от клавиатуры сообщение. Windows сама обрабатывает многие функции клавиатуры. Например, вы можете игнорировать нажатие системных клавиш. В таких нажатиях обычно используется клавиша Alt.

Программе не нужно отслеживать нажатие этих клавиш, поскольку Windows извещает программу об эффекте, вызванном их нажатием. (Однако, при желании программа сама может их отслеживать.) Например, когда пользователь Windows выбирает пункт меню с помощью клавиатуры, Windows посылает программе сообщение, что выбран пункт меню, независимо от того, был ли он выбран с помощью мыши, или с помощью клавиатуры. (О работе с меню рассказывается в главе 10.)

В некоторых программах для Windows используются "быстрые клавиши" (keyboard accelerators) для быстрого доступа к часто употребляемым пунктам меню. В качестве быстрых клавиш обычно используются функциональные клавиши или комбинация символьной клавиши и клавиши <Ctrl>. Такие быстрые клавиши определяются в описании ресурсов программы. (В главе 10 показано, как Windows преобразует быстрые клавиши в сообщения команд меню. Вам не нужно самим делать это преобразование.)

Окна диалога (о которых рассказывается в главе 11) также имеют интерфейс клавиатуры, но обычно программам не нужно отслеживать клавиатурные события, когда активно окно диалога. Интерфейс клавиатуры обслуживается самой Windows, и Windows посылает сообщения вашей программе о действиях, соответствующих нажимаемым клавишам. В окнах диалога могут содержаться "окна редактирования" (edit) для ввода текста. Это обычно небольшие окна, в которых пользователь набирает строки символов. Windows управляет всей логикой окон редактирования и дает вашей программе окончательное содержимое этих окон, после того, как пользователь завершит ввод строки.

Даже внутри вашего главного окна вы можете определить дочерние окна как окна редактирования. Особым примером этого является программа POPPAD, представленная в главе 8. Эта программа практически представляет собой просто большое окно редактирования, причем вся черновая работа в ней возложена на Windows.

Фокус ввода

Клавиатура должна разделяться между всеми приложениями, работающими под Windows. Некоторые приложения могут иметь больше одного окна, и клавиатура должна разделяться между этими окнами в рамках одного и того же приложения. Когда на клавиатуре нажата клавиша, только одна оконная процедура может получить сообщение об этом. Окно, которое получает это сообщение клавиатуры, является окном, имеющем "фокус ввода" (input focus).

Концепция фокуса ввода тесно связана с концепцией "активного окна". Окно, имеющее фокус ввода — это либо активное окно, либо дочернее окно активного окна. Определить активное окно обычно достаточно просто. Если у активного окна имеется панель заголовка, то Windows выделяет ее. Если у активного окна вместо панели заголовка имеется рамка диалога (это наиболее часто встречается в окнах диалога), то Windows выделяет ее. Если активное окно минимизировано, то Windows выделяет текст заголовка в панели задач.

Наиболее часто дочерними окнами являются кнопки, переключатели, флажки, полосы прокрутки и списки, которые обычно присутствуют в окне диалога. Сами по себе дочерние окна никогда не могут быть активными. Если фокус ввода находится в дочернем окне, то активным является родительское окно этого дочернего окна. То, что фокус ввода находится в дочерних окнах, обычно показывается посредством мигающего курсора или каретки (caret).

Если активное окно минимизировано, то окна с фокусом ввода нет. Windows продолжает слать программе сообщения клавиатуры, но эти сообщения выглядят иначе, чем сообщения, направленные активным и еще не минимизированным окнам.

Обрабатывая сообщения WM_SETFOCUS и WM_KILLFOCUS, оконная процедура может определить, когда окно имеет фокус ввода. WM_SETFOCUS показывает, что окно получило фокус ввода, а WM_KILLFOCUS, что окно потеряло его.

Аппаратные и символьные сообщения

Сообщения, которые приложение получает от Windows о событиях, относящихся к клавиатуре, различаются на "аппаратные" (keystrokes) и "символьные" (characters). Такое положение соответствует двум представлениям о клавиатуре. Во-первых, вы можете считать клавиатуру набором клавиш. В клавиатуре имеется только одна клавиша <A>. Нажатие на эту клавишу является аппаратным событием. Отпускание этой клавиши является аппаратным событием. Но клавиатура также является устройством ввода, генерирующем отображаемые символы. Клавиша <A>, в зависимости от состояния клавиш <Ctrl>, <Shift> и <CapsLock>, может стать источником нескольких символов. Обычно, этим символом является строчное ‘a’. Если нажата клавиша <Shift> или установлен режим Caps Lock, то этим символом является прописное <A>. Если нажата клавиша <Ctrl>, этим символом является <Ctrl>+<A>. На клавиатуре, поддерживающей иностранные языки, аппаратному событию ‘A’ может предшествовать либо специальная клавиша, либо <Shift>, либо <Ctrl>, либо <Alt>, либо их различные сочетания. Эти сочетания могут стать источником вывода строчного ‘a’ или прописного ‘A’ с символом ударения.

Для сочетаний аппаратных событий, которые генерируют отображаемые символы, Windows посылает программе и оба аппаратных и символьное сообщения. Некоторые клавиши не генерируют символов. Это такие клавиши, как

клавиши переключения, функциональные клавиши, клавиши управления курсором и специальные клавиши, такие как Insert и Delete. Для таких клавиш Windows вырабатывает только аппаратные сообщения.

Аппаратные сообщения

Когда вы нажимаете клавишу, Windows помещает либо сообщение WM_KEYDOWN, либо сообщение WM_SYSKEYDOWN в очередь сообщений окна, имеющего фокус ввода. Когда вы отпускаете клавишу, Windows помещает либо сообщение WM_KEYUP, либо сообщение WM_SYSKEYUP в очередь сообщений.

	Клавиша нажата	Клавиша отпущена
Несистемные аппаратные сообщения	WM_KEYDOWN	WM_KEYUP
Системные аппаратные сообщения	WM_SYSKEYDOWN	WM_SYSKEYUP

Обычно сообщения о "нажатии" (down) и "отпускании" (up) появляются парами. Однако, если вы оставите клавишу нажатой так, чтобы включился автоповтор, то Windows посылает оконной процедуре серию сообщений WM_KEYDOWN (или WM_SYSKEYDOWN) и одно сообщение WM_KEYUP (или WM_SYSKEYUP), когда в конце концов клавиша будет отпущена. Также как и все синхронные сообщения, аппаратные сообщения клавиатуры также становятся в очередь. Вы можете с помощью функции *GetMessageTime* получить время нажатия и отпускания клавиши относительно старта системы.

Системные и несистемные аппаратные сообщения клавиатуры

Префикс "SYS" в WM_SYSKEYDOWN и WM_SYSKEYUP означает "системное" (system) и относится к аппаратным сообщениям клавиатуры, которые больше важны для Windows, чем для приложений Windows. Сообщения WM_SYSKEYDOWN и WM_SYSKEYUP обычно вырабатываются при нажатии клавиш в сочетании с клавишей <Alt>. Эти сообщения вызывают опции меню программы или системного меню, или используются для системных функций, таких как смена активного окна (<Alt>+<Tab> или <Alt>+<Esc>), или как быстрые клавиши системного меню (<Alt> в сочетании с функциональной клавишей). Программы обычно игнорируют сообщения WM_SYSKEYDOWN и WM_SYSKEYUP и передают их *DefWindowProc*. Поскольку Windows отрабатывает всю логику Alt-клавиш, то вам фактически не нужно обрабатывать эти сообщения. Ваша оконная процедура в конце концов получит другие сообщения, являющиеся результатом этих аппаратных сообщений клавиатуры (например, выбор меню). Если вы хотите включить в код вашей оконной процедуры инструкции для обработки аппаратных сообщений клавиатуры (как мы это сделаем в программе KEYLOOK, представленной далее в этой главе), то после обработки этих сообщений передайте их в *DefWindowProc*, чтобы Windows могла по-прежнему их использовать в обычных целях.

Но подумайте немного об этом. Почти все, что влияет на окно вашей программы, в первую очередь проходит через вашу оконную процедуру. Windows каким-то образом обрабатывает сообщения только в том случае, если они передаются в *DefWindowProc*. Например, если вы добавите строки:

```
case WM_SYSKEYDOWN;
case WM_SYSKEYUP;
case WM_SYSCHAR;
     return 0;
```

в оконную процедуру, то запретите все операции с клавишей <Alt> (команды меню, <Alt>+<Tab>, <Alt>+<Esc> и т. д.), когда ваша программа имеет фокус ввода. Маловероятно, что вы захотели бы так сделать, тем не менее есть надежда, что вы начинаете понимать мощь вашей оконной процедуры.

Сообщения WM_KEYDOWN и WM_KEYUP обычно вырабатываются для клавиш, которые нажимаются и отпускаются без участия клавиши <Alt>. Ваша программа может использовать или не использовать эти сообщения клавиатуры. Сама Windows их игнорирует.

Переменная *lParam*

Для всех аппаратных сообщений клавиатуры, 32-разрядная переменная *lParam*, передаваемая в оконную процедуру, состоит из шести полей: счетчика повторений (Repeat Count), скан-кода OEM (Original Equipment Manufacturer Scan Code), флага расширенной клавиатуры (Extended Key Flag), кода контекста (Context Code), флага предыдущего состояния клавиши (Previous Key State) и флага состояния клавиши (Transition State). (См. рис. 5.1)

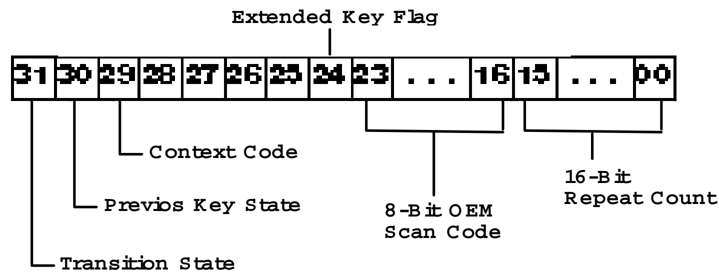


Рис. 5.1 Шесть полей переменной *lParam* аппаратных сообщений клавиатуры

Счетчик повторений

Счетчик повторений равен числу нажатий клавиши, которое отражено в сообщении. В большинстве случаев он устанавливается в 1. Однако, если клавиша остается нажатой, а ваша оконная процедура недостаточно быстра, чтобы обрабатывать эти сообщения в темпе автоповтора (по умолчанию это приблизительно 10 символов в секунду), то Windows объединяет несколько сообщений WM_KEYDOWN или WM_SYSKEYDOWN в одно сообщение и соответственно увеличивает счетчик повторений. Для сообщения WM_KEYUP или WM_SYSKEYUP счетчик повторений всегда равен 1.

Поскольку значение счетчика большее 1 показывает, что имеющийся темп автоповтора превышает возможности вашей программы по обработке, вы можете игнорировать счетчик повторений при обработке сообщений клавиатуры. Почти каждый из вас имел опыт работы с документами в программах текстовых редакторов или электронных таблиц, и видел, как строки или страницы продолжают перелистываться и после отпускания соответствующей клавиши. Это происходит из-за дополнительных сообщений в буфере клавиатуры, которые накапливаются там, если некоторое время удерживать клавишу. Игнорирование счетчика повторений в вашей программе существенно снизит вероятность проявления этого эффекта. Тем не менее, счетчик повторений может быть использован. Вероятно, вам следует протестировать в ваших программах обе эти возможности и выбрать наиболее подходящий вариант.

Скан-код OEM

Скан-код OEM является кодом клавиатуры, генерируемым аппаратурой компьютера. (Если вы хорошо знакомы с программированием на языке ассемблера, то скан-код — это код, передаваемый программе в регистре AH при вызове прерывания 16H BIOS.) Приложения Windows обычно игнорируют скан-код OEM, поскольку имеют более совершенные способы расшифровки информации от клавиатуры.

Флаг расширенной клавиатуры

Флаг расширенной клавиатуры устанавливается в 1, если сообщение клавиатуры появилось в результате работы с дополнительными клавишами расширенной клавиатуры IBM. (Расширенная клавиатура IBM имеет функциональные клавиши сверху и отдельную комбинированную область клавиш управления курсором и цифр.) Этот флаг устанавливается в 1 для клавишей <Alt> и <Ctrl> на правой стороне клавиатуры, клавиш управления курсором (включая <Insert> и <Delete>), которые не являются частью числовой клавиатуры, клавиш наклонной черты (</>) и <Enter> на числовой клавиатуре и клавиши <NumLock>. Программы для Windows обычно игнорируют флаг расширенной клавиатуры.

Код контекста

Код контекста устанавливается в 1, если нажата клавиша <Alt>. Этот разряд всегда равен 1 для сообщений WM_SYSKEYDOWN и WM_SYSKEYUP и 0 для сообщений WM_KEYDOWN и WM_KEYUP с двумя исключениями:

- Если активное окно минимизировано, оно не имеет фокус ввода. Все нажатия клавиш вырабатывают сообщения WM_SYSKEYDOWN и WM_SYSKEYUP. Если не нажата клавиша <Alt>, поле кода контекста устанавливается в 0. (Windows использует SYS сообщения клавиатуры так, чтобы активное окно, которое минимизировано, не обрабатывало эти сообщения.)

- На некоторых иноязычных клавиатурах некоторые символы генерируются комбинацией клавиш <Shift>, <Ctrl> или <Alt> с другой клавишей. В этих случаях, у переменной *lParam*, которая сопровождает сообщения WM_KEYDOWN и WM_KEYUP, в поле кода контекста ставится 1, но эти сообщения не являются системными сообщениями клавиатуры.

Флаг предыдущего состояния клавиши

Флаг предыдущего состояния клавиши равен 0, если в предыдущем состоянии клавиша была отпущена, и 1, если в предыдущем состоянии она была нажата. Он всегда устанавливается в 1 для сообщения WM_KEYUP или WM_SYSKEYUP, но для сообщения WM_KEYDOWN или WM_SYSKEYDOWN он может устанавливаться как в 1, так и в 0. В этом случае 1 показывает наличие второго и последующих сообщений от клавиш в результате автоповтора.

Флаг состояния клавиши

Флаг состояния клавиши равен 0, если клавиша нажимается, и 1, если клавиша отпускается. Это поле устанавливается в 0 для сообщения WM_KEYDOWN или WM_SYSKEYDOWN и в 1 для сообщения WM_KEYUP или WM_SYSKEYUP.

Виртуальные коды клавиш

Хотя некоторая информация в *lParam* при обработке сообщений WM_KEYUP, WM_KEYDOWN, WM_SYSKEYUP и WM_SYSKEYDOWN может оказаться полезной, гораздо более важен параметр *wParam*. В этом параметре содержится "виртуальный код клавиши" (virtual key code), идентифицирующий нажатую или отпущенную клавишу. Разработчики Windows попытались определить виртуальные клавиши независимым от аппаратуры способом. По этой причине, некоторые виртуальные коды клавиш не могут вырабатываться на персональных компьютерах IBM и совместимых с ними, но их можно встретить на клавиатурах других производителей.

Виртуальные коды клавиш, которые вы используете, наиболее часто имеют имена, определенные в заголовочных файлах Windows. В показанной ниже таблице представлены эти имена, числовые коды клавиш, а также клавиши персональных компьютеров IBM, соответствующие виртуальным клавишам. Хотя все клавиши вызывают аппаратные сообщения клавиатуры, в таблицу не включены клавиши символов (например, клавиши символов / или ?). Эти клавиши имеют виртуальные коды от 128 и выше, и они в международных клавиатурах часто определяются по-другому. Вы можете определить значения этих виртуальных кодов клавиш с помощью программы KEYLOOK, представленной далее в этой главе, но обычно вам не нужно обрабатывать аппаратные сообщения клавиатуры для этих клавиш.

Виртуальные коды клавиш

Десятичное	Шестнадцатеричное	WINDOWS.H Идентификатор	Требуется	Клавиатура IBM
1	01	VK_LBUTTON		
2	02	VK_RBUTTON		
3	03	VK_CANCEL	∨	Ctrl-Break
4	04	VK_MBUTTON		
8	08	VK_BACK	∨	Backspace
9	09	VK_TAB	∨	Tab
12	0C	VK_CLEAR		5 на дополнительной цифровой клавиатуре при отключенном Num Lock
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 (Spacebar)
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
41	29	VK_SELECT		
42	2A	VK_PRINT		
43	2B	VK_EXECUTE		
44	2C	VK_SNAPSHOT		Print Screen
45	2D	VK_INSERT	∨	Insert

Деся-тичное	Шестнадца-теричное	WINDOWS.H Идентификатор	Требу-ется	Клавиатура IBM
46	2E	VK_DELETE	˅	Delete
47	2F	VK_HELP		
48-57	30-39		˅	От 0 до 9 на основной клавиатуре
65-90	41-5A		˅	От A до Z
96	60	VK_NUMPAD0		0 на дополнительной цифровой клавиатуре при включенном Num Lock
97	61	VK_NUMPAD1		1 на дополнительной цифровой клавиатуре при включенном Num Lock
98	62	VK_NUMPAD2		2 на дополнительной цифровой клавиатуре при включенном Num Lock
99	63	VK_NUMPAD3		3 на дополнительной цифровой клавиатуре при включенном Num Lock
100	64	VK_NUMPAD4		4 на дополнительной цифровой клавиатуре при включенном Num Lock
101	65	VK_NUMPAD5		5 на дополнительной цифровой клавиатуре при включенном Num Lock
102	66	VK_NUMPAD6		6 на дополнительной цифровой клавиатуре при включенном Num Lock
103	67	VK_NUMPAD7		7 на дополнительной цифровой клавиатуре при включенном Num Lock
104	68	VK_NUMPAD8		8 на дополнительной цифровой клавиатуре при включенном Num Lock
105	69	VK_NUMPAD9		9 на дополнительной цифровой клавиатуре при включенном Num Lock
106	6A	VK_MULTIPLY		* на дополнительной цифровой клавиатуре
107	6B	VK_ADD		+ на дополнительной цифровой клавиатуре
108	6C	VK_SEPARATOR		
109	6D	VK_SUBTRACT		- на дополнительной цифровой клавиатуре
110	6E	VK_DECIMAL		. на дополнительной цифровой клавиатуре
111	6F	VK_DIVIDE		/ на дополнительной цифровой клавиатуре
112	70	VK_F1	˅	Функциональная клавиша F1
113	71	VK_F2	˅	Функциональная клавиша F2
114	72	VK_F3	˅	Функциональная клавиша F3
115	73	VK_F4	˅	Функциональная клавиша F4
116	74	VK_F5	˅	Функциональная клавиша F5
117	75	VK_F6	˅	Функциональная

Десятичное	Шестнадцатеричное	WINDOWS.H Идентификатор	Требуется	Клавиатура IBM
				клавиша F6
118	76	VK_F7	∨	Функциональная клавиша F7
119	77	VK_F8	∨	Функциональная клавиша F8
120	78	VK_F9	∨	Функциональная клавиша F9
121	79	VK_F10	∨	Функциональная клавиша F10
122	7A	VK_F11		Функциональная клавиша F11 (расширенная клавиатура)
123	7B	VK_F12		Функциональная клавиша F12 (расширенная клавиатура)
124	7C	VK_F13		
125	7D	VK_F14		
126	7E	VK_F15		
127	7F	VK_F16		
144	90	VK_NUMLOCK		Num Lock
145	91	VK_SCROLL		Scroll Lock

Пометка (∨) в столбце "Требуется" показывает, что клавиша предопределена для любой реализации Windows. Windows также требует, чтобы клавиатура и драйвер клавиатуры позволяли комбинировать клавиши <Shift>, <Ctrl>, а также <Shift> и <Ctrl> вместе, со всеми буквенными клавишами, всеми клавишами управления курсором и всеми функциональными клавишами. Виртуальные коды клавиш VK_LBUTTON, VK_MBUTTON и VK_RBUTTON относятся к левой, центральной и правой кнопкам мыши. Однако, вам никогда не удастся получить сообщения клавиатуры с параметром *wParam*, в котором установлены эти значения. Мышь, как мы увидим в следующей главе, вырабатывает свои собственные сообщения.

Положения клавиш сдвига и клавиш-переключателей

Параметры *wParam* и *lParam* сообщений WM_KEYUP, WM_KEYDOWN, WM_SYSKEYUP и WM_SYSKEYDOWN ничего не сообщают вашей программе о положении клавиш сдвига и клавиш-переключателей. Вы можете получить текущее состояние любой виртуальной клавиши с помощью функции *GetKeyState*. Эта функция в основном используется для получения информации о состоянии клавиш сдвига (<Shift>, <Ctrl> и <Alt>) и клавиш-переключателей (<CapsLock>, <NumLock> и <ScrollLock>). Например:

```
GetKeyState(VK_SHIFT);
```

возвращает отрицательное значение (т. е., установлен старший разряд), если клавиша <Shift> нажата. В возвращаемом функцией:

```
GetKeyState(VK_CAPITAL);
```

значении установлен младший разряд, если переключатель <CapsLock> включен. Вы также можете получить положение кнопок мыши с помощью виртуальных кодов клавиш VK_LBUTTON, VK_MBUTTON и VK_RBUTTON. Однако, большая часть программ для Windows, которым надо отслеживать сочетание состояний кнопок мыши и клавиш клавиатуры, делают это другим способом — проверяя состояние клавиш клавиатуры при получении сообщения от мыши. Фактически, информация о положении клавиш сдвига и клавиш-переключателей включается в сообщения от мыши (как вы увидите в следующей главе).

Будьте осторожны с функцией *GetKeyState*. Она не отражает положение клавиатуры в реальном времени. Она отражает состояние клавиатуры на момент, когда последнее сообщение от клавиатуры было выбрано из очереди. Функция *GetKeyState* не позволяет вам получать информацию о клавиатуре, независимо от обычных сообщений клавиатуры. Например, вы захотите, чтобы процесс обработки сообщения в оконной процедуре продолжался до тех пор, пока пользователь не нажмет функциональную клавишу <F1>:

```
while(GetKeyState(VK_F1) >= 0); // ОШИБКА!!!
```

Не делайте так! Вашей программе необходимо получить сообщение клавиатуры из очереди сообщений до того, как *GetKeyState* сможет определить состояние клавиши. Эта синхронизация, в самом деле, дает вам преимущество, потому что, если вам нужно узнать положение переключателя для конкретного сообщения клавиатуры, *GetKeyState* обеспечивает возможность получения точной информации, даже если вы обработаете сообщение уже

после того, как состояние переключателя было изменено. Если вам действительно нужна информация о текущем положении клавиши, вы можете использовать функцию *GetAsyncKeyState*.

Использование сообщений клавиатуры

Идея программы, получающей информацию о нажатии любой клавиши несомненно привлекательна; однако, большинство программ для Windows игнорируют все, кроме нескольких сообщений о нажатии и отпускании клавиш. Сообщения WM_SYSKEYUP и WM_SYSKEYDOWN адресованы системным функциям Windows, и вам не нужно их отслеживать. Если вы обрабатываете сообщения WM_KEYDOWN, то сообщения WM_KEYUP вам обычно также можно игнорировать.

Программы для Windows обычно используют сообщения WM_KEYDOWN для нажатия и отпускания клавиш, которые не генерируют символьные сообщения. Хотя вы можете подумать, что есть возможность использовать сообщения о нажатии клавиш в сочетании с информацией о состоянии клавиш сдвига для преобразования сообщений о нажатии клавиш в символьные сообщения, не делайте так. У вас будут проблемы из-за отличий международных клавиатур. Например, если вы получаете сообщение WM_KEYDOWN с *wParam* равным 33H, вы знаете, что пользователь нажал клавишу <3>. Так, хорошо. Если вы используете *GetKeyState* и обнаруживаете, что клавиша <Shift> нажата, то можно было бы предположить, что пользователь печатает знак фунта стерлингов <£>. Вовсе необязательно. Британский пользователь печатает знак <&>. Поэтому сообщения WM_KEYDOWN более удобны для клавиш управления курсором, функциональных клавиш и специальных клавиш, таких как <Insert> и <Delete>. Однако, иногда клавиши <Insert> и <Delete>, а также функциональные клавиши используются в качестве быстрых клавиш меню. Поскольку Windows преобразует быстрые клавиши меню в сообщения команд меню, вы не должны сами обрабатывать эти сообщения. Некоторые программы, написанные не для Windows, широко используют функциональные клавиши в сочетании с клавишами <Shift>, <Ctrl> и <Alt>. Вы можете сделать что-то похожее в ваших программах для Windows, но это не рекомендуется. Если вы хотите использовать функциональные клавиши, то лучше, чтобы они дублировали команды меню. Одна из задач Windows — обеспечить такой пользовательский интерфейс, для которого не требуется заучивание или использование сложного набора команд.

Мы собираемся отказаться от всего, за исключением последнего пункта: большую часть времени вы будете обрабатывать сообщения WM_KEYDOWN только для клавиш управления курсором. Если вы используете клавиши управления курсором, то можете контролировать состояние клавиш <Shift> и <Ctrl> с помощью функции *GetKeyState*. Функции Windows часто используют клавишу <Shift> в сочетании с клавишами управления курсором для расширения выбора, например, в программах текстовых редакторов. Клавиша <Ctrl> часто используется для изменения значения клавиш управления курсором. (Например, <Ctrl> в сочетании с клавишей стрелки вправо могло бы означать перемещение курсора на одно слово вправо.)

Одним из лучших способов выяснить то, как использовать клавиатуру должно быть изучение использования клавиатуры в существующих популярных программах для Windows. Если вам это не подходит, можете действовать как-то иначе. Но запомните, что в этом случае вы можете помешать пользователю быстро изучить вашу программу.

Модернизация SYSMETS: добавление интерфейса клавиатуры

Когда в главе 3 мы написали три версии программы SYSMETS, мы ничего не знали о клавиатуре. Мы могли прокрутить текст только с помощью мыши на полосе прокрутки. Теперь мы знаем, как обрабатывать сообщения клавиатуры, давайте добавим интерфейс клавиатуры в программу SYSMETS. Это очевидно будет работа для клавиш управления курсором. Мы используем большинство клавиш управления курсором <Home>, <End>, <PageUp>, <PageDown>, <↑> и <↓> для вертикальной прокрутки. Клавиши <→> и <←> можно оставить на менее важную горизонтальную прокрутку.

Логика обработки сообщений WM_KEYDOWN

Один из простейших способов создать интерфейс клавиатуры — это использовать логику обработки сообщений WM_KEYDOWN в оконной процедуре, которая будет работать параллельно с логикой обработки сообщений WM_VSCROLL и WM_HSCROLL:

```
case WM_KEYDOWN
       iVscrollInc = iHscrollInc = 0;
       switch(wParam)
       {
       case VK_HOME: // аналогично WM_VSCROLL, SB_TOP
              iVscrollInc = — iVscrollPos;
              break;
```

```
        case VK_END:  // аналогично WM_VSCROLL, SB_BOTTOM
              iVscrollInc = iVscrollMax — iVscrollPos;
              break;

        case VK_UP:           // аналогично WM_VSCROLL, SB_LINEUP
              iVscrollInc = — 1;
              break;

        case VK_DOWN: // аналогично WM_VSCROLL, SB_LINEDOWN
              iVscrollInc = 1;
              break;

        case VK_PRIOR:        // аналогично WM_VSCROLL, SB_PAGEUP
              iVscrollInc = min(-1, — cyClient / cyChar);
              break;

        case VK_NEXT: // аналогично WM_VSCROLL, SB_PAGEDOWN
              iVscrollInc = max(1, cyClient / cyChar);
              break;

        case VK_LEFT: // аналогично WM_HSCROLL, SB_PAGEUP
              iHscrollInc = — 8;
              break;

        case VK_RIGHT:// аналогично WM_HSCROLL, SB_PAGEDOWN
              iHscrollInc = 8;
              break;

        default:
              break;
        }
if (iVscrollInc = max(- iVscrollPos, min(iVscrollInc, iVscrollMax — iVscrollPos)))
{
        iVscrollPos += iVscrollInc;
        ScrollWindow(hwnd, 0, — cyChar * iVscrollInc, NULL, NULL);
        SetScrollPos(hwnd, SB_VERT, iVscrollPos, TRUE);
        UpdateWindow(hwnd);
}

if (iHscrollInc = max(- iHscrollPos, min(iHscrollInc, iHscrollMax — iHscrollPos)))
{
        iHscrollPos += iHscrollInc;
        ScrollWindow(hwnd, — cxChar * iHscrollInc, 0, NULL, NULL);
        SetScrollPos(hwnd, SB_HORZ, iHscrollPos, TRUE);
}

return 0;
```

Вам тоже не нравится эта программа? Простое дублирование всех инструкций предыдущей программы занятие глупое, поскольку, если когда-нибудь захочется изменить логику работы полос прокрутки, то придется делать параллельные изменения в логике WM_KEYDOWN. Должен быть лучший способ. И он есть.

Посылка асинхронных сообщений

Не лучше было бы просто преобразовать каждое из этих сообщений WM_KEYDOWN в эквивалентное сообщение WM_VSCROLL и WM_HSCROLL и, таким образом, быть может, обмануть оконную процедуру *WndProc*, чтобы ей казалось, что она получает сообщения полосы прокрутки WM_VSCROLL или WM_HSCROLL? Windows позволяет это сделать. Функция называется *SendMessage*, и имеет те же параметры, что и параметры, передаваемые в оконную процедуру:

```
SendMessage(hwnd, message, wParam, lParam);
```

Когда вы вызываете *SendMessage*, то Windows вызывает оконную процедуру с описателем окна *hwnd*, передавая ей эти четыре параметра. После того, как оконная процедура заканчивает обработку сообщения, Windows передает управление следующей за вызовом *SendMessage* инструкции. Оконная процедура, которой вы отправляете синхронное сообщение, может быть той же самой оконной процедурой, другой оконной процедурой той же программы, или даже оконной процедурой другого приложения.

Далее показано, как можно было бы использовать *SendMessage* для обработки WM_KEYDOWN в программе SYSMETS:

```
case WM_KEYDOWN:
       switch(wParam)
       {
       case VK_HOME:
              SendMessage(hwnd, WM_VSCROLL, SB_TOP, 0L);
              break;

       case VK_END:
              SendMessage(hwnd, WM_VSCROLL, SB_BOTTOM, 0L);
              break;

       case VK_PRIOR:
              SendMessage(hwnd, WM_VSCROLL, SB_PAGEUP, 0L);
              break;
```

[остальные строки программы]

Теперь вы поняли основную идею. Нашей целью было добавить интерфейс клавиатуры к полосам прокрутки, и мы это сделали. Мы продублировали логику обработки сообщений полос прокрутки обработкой сообщений о нажатии клавиш управления курсором, фактически передавая оконной процедуре синхронное сообщение полос прокрутки. Теперь вы понимаете, почему в программу SYSMETS3 включена обработка SB_TOP и SB_BOTTOM для сообщений WM_VSCROLL. Тогда это не использовалось, а сейчас используется для обработки клавиш Home и End. Последняя программа SYSMETS, показанная на рис. 5.2, включает в себя все эти изменения. Вам для компиляции этой программы понадобится также файл SYSMETS.H из главы 3 (рис. 3.4).

SYSMETS.MAK

```
#-----------------------
# SYSMETS.MAK make file
#-----------------------

sysmets.exe : sysmets.obj
     $(LINKER) $(GUIFLAGS) -OUT:sysmets.exe sysmets.obj $(GUILIBS)

sysmets.obj : sysmets.c sysmets.h
     $(CC) $(CFLAGS) sysmets.c
```

SYSMETS.C

```
/*-----------------------------------------------------
   SYSMETS.C -- System Metrics Display Program(Final)
               (c) Charles Petzold, 1996
  -----------------------------------------------------*/

#include <windows.h>
#include <string.h>
#include "sysmets.h"

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "SysMets";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
```

```
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "System Metrics",
                         WS_OVERLAPPEDWINDOW | WS_VSCROLL | WS_HSCROLL,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int  cxChar, cxCaps, cyChar, cxClient, cyClient, iMaxWidth,
                 iVscrollPos, iVscrollMax, iHscrollPos, iHscrollMax;
     char        szBuffer[10];
     HDC         hdc;
     int         i, x, y, iPaintBeg, iPaintEnd, iVscrollInc, iHscrollInc;
     PAINTSTRUCT ps;
     TEXTMETRIC  tm;

     switch(iMsg)
          {
          case WM_CREATE :
               hdc = GetDC(hwnd);

               GetTextMetrics(hdc, &tm);
               cxChar = tm.tmAveCharWidth;
               cxCaps =(tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2;
               cyChar = tm.tmHeight + tm.tmExternalLeading;

               ReleaseDC(hwnd, hdc);

               iMaxWidth = 40 * cxChar + 22 * cxCaps;
               return 0;

          case WM_SIZE :
               cxClient = LOWORD(lParam);
               cyClient = HIWORD(lParam);

               iVscrollMax = max(0, NUMLINES + 2 - cyClient / cyChar);
               iVscrollPos = min(iVscrollPos, iVscrollMax);

               SetScrollRange(hwnd, SB_VERT, 0, iVscrollMax, FALSE);
               SetScrollPos  (hwnd, SB_VERT, iVscrollPos, TRUE);

               iHscrollMax = max(0, 2 +(iMaxWidth - cxClient) / cxChar);
               iHscrollPos = min(iHscrollPos, iHscrollMax);

               SetScrollRange(hwnd, SB_HORZ, 0, iHscrollMax, FALSE);
               SetScrollPos  (hwnd, SB_HORZ, iHscrollPos, TRUE);
               return 0;
```

```
case WM_VSCROLL :
     switch(LOWORD(wParam))
          {
          case SB_TOP :
               iVscrollInc = -iVscrollPos;
               break;
          case SB_BOTTOM :
               iVscrollInc = iVscrollMax - iVscrollPos;
               break;

          case SB_LINEUP :
               iVscrollInc = -1;
               break;

          case SB_LINEDOWN :
               iVscrollInc = 1;
               break;

          case SB_PAGEUP :
               iVscrollInc = min(-1, -cyClient / cyChar);
               break;

          case SB_PAGEDOWN :
               iVscrollInc = max(1, cyClient / cyChar);
               break;

          case SB_THUMBTRACK :
               iVscrollInc = HIWORD(wParam) - iVscrollPos;
               break;

          default :
               iVscrollInc = 0;
          }
     iVscrollInc = max(-iVscrollPos,
              min(iVscrollInc, iVscrollMax - iVscrollPos));

     if(iVscrollInc != 0)
          {
          iVscrollPos += iVscrollInc;
          ScrollWindow(hwnd, 0, -cyChar * iVscrollInc, NULL, NULL);
          SetScrollPos(hwnd, SB_VERT, iVscrollPos, TRUE);
          UpdateWindow(hwnd);
          }
     return 0;

case WM_HSCROLL :
     switch(LOWORD(wParam))
          {
          case SB_LINEUP :
               iHscrollInc = -1;
               break;

          case SB_LINEDOWN :
               iHscrollInc = 1;
               break;
          case SB_PAGEUP :
               iHscrollInc = -8;
               break;

          case SB_PAGEDOWN :
               iHscrollInc = 8;
               break;

          case SB_THUMBPOSITION :
```

```
                    iHscrollInc = HIWORD(wParam) - iHscrollPos;
                    break;

               default :
                    iHscrollInc = 0;
               }
          iHscrollInc = max(-iHscrollPos,
                        min(iHscrollInc, iHscrollMax - iHscrollPos));

          if(iHscrollInc != 0)
               {
               iHscrollPos += iHscrollInc;
               ScrollWindow(hwnd, -cxChar * iHscrollInc, 0, NULL, NULL);
               SetScrollPos(hwnd, SB_HORZ, iHscrollPos, TRUE);
               }
          return 0;

     case WM_KEYDOWN :
          switch(wParam)
               {
               case VK_HOME :
                    SendMessage(hwnd, WM_VSCROLL, SB_TOP, 0L);
                    break;

               case VK_END :
                    SendMessage(hwnd, WM_VSCROLL, SB_BOTTOM, 0L);
                    break;

               case VK_PRIOR :
                    SendMessage(hwnd, WM_VSCROLL, SB_PAGEUP, 0L);
                    break;

               case VK_NEXT :
                    SendMessage(hwnd, WM_VSCROLL, SB_PAGEDOWN, 0L);
                    break;

               case VK_UP :
                    SendMessage(hwnd, WM_VSCROLL, SB_LINEUP, 0L);
                    break;
               case VK_DOWN :
                    SendMessage(hwnd, WM_VSCROLL, SB_LINEDOWN, 0L);
                    break;
               case VK_LEFT :
                    SendMessage(hwnd, WM_HSCROLL, SB_PAGEUP, 0L);
                    break;

               case VK_RIGHT :
                    SendMessage(hwnd, WM_HSCROLL, SB_PAGEDOWN, 0L);
                    break;
               }
          return 0;

     case WM_PAINT :
          hdc = BeginPaint(hwnd, &ps);

          iPaintBeg = max(0, iVscrollPos + ps.rcPaint.top / cyChar - 1);
          iPaintEnd = min(NUMLINES,
                          iVscrollPos + ps.rcPaint.bottom / cyChar);

          for(i = iPaintBeg; i < iPaintEnd; i++)
               {
               x = cxChar *(1 - iHscrollPos);
               y = cyChar *(1 - iVscrollPos + i);
```

```
                TextOut(hdc, x, y,
                         sysmetrics[i].szLabel,
                         strlen(sysmetrics[i].szLabel));

                TextOut(hdc, x + 22 * cxCaps, y,
                         sysmetrics[i].szDesc,
                         strlen(sysmetrics[i].szDesc));

                SetTextAlign(hdc, TA_RIGHT | TA_TOP);

                TextOut(hdc, x + 22 * cxCaps + 40 * cxChar, y,
                         szBuffer,
                         wsprintf(szBuffer, "%5d",
                             GetSystemMetrics(sysmetrics[i].iIndex)));

                SetTextAlign(hdc, TA_LEFT | TA_TOP);
                }

           EndPaint(hwnd, &ps);
           return 0;

      case WM_DESTROY :
           PostQuitMessage(0);
           return 0;
      }

 return DefWindowProc(hwnd, iMsg, wParam, lParam);
 }
```

Рис. 5.2 Программа SYSMETS

Символьные сообщения

Ранее рассматривалась идея преобразования аппаратных сообщений клавиатуры в символьные сообщения путем учета информации о положении клавиш сдвига и предупреждалось, что информации о положении этих клавиш недостаточно: вам необходимо также знать об особенностях реализации вашей национальной клавиатуры. По этой причине вам не следует пытаться самостоятельно преобразовывать аппаратные сообщения клавиатуры в символьные коды.

Для вас это делает Windows. Вы уже встречали такой код раньше:

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

Это типичный цикл обработки сообщений, содержащийся в *WinMain*. Функция *GetMessage* заполняет поля структуры *msg* данными следующего сообщения из очереди. Вызов *DispatchMessage* вызывает соответствующую оконную процедуру.

Между двумя этими функциями находится функция *TranslateMessage*, преобразующая аппаратные сообщения клавиатуры в символьные сообщения. Если этим сообщением является WM_KEYDOWN или WM_SYSKEYDOWN и, если нажатие клавиши в сочетании с положением клавиши сдвига генерирует символ, тогда *TranslateMessage* помещает символьное сообщение в очередь сообщений. Это символьное сообщение будет следующим, после сообщения о нажатии клавиши, которое функция *GetMessage* извлечет из очереди сообщений.

Существует четыре символьных сообщения:

	Символы	Немые символы
Несистемные символы:	WM_CHAR	WM_DEADCHAR
Системные символы:	WM_SYSCHAR	WM_SYSDEADCHAR

Сообщения WM_CHAR и WM_DEADCHAR являются следствием сообщений WM_KEYDOWN. Сообщения WM_SYSCHAR и WM_SYSDEADCHAR являются следствием сообщений WM_SYSKEYDOWN. В большинстве случаев ваши программы для Windows могут игнорировать все сообщения, за исключением WM_CHAR. Параметр

lParam, передаваемый в оконную процедуру как часть символьного сообщения, является таким же, как параметр *lParam* аппаратного сообщения клавиатуры, из которого сгенерировано символьное сообщение. Параметр *wParam* — это код символа ASCII.

Символьные сообщения доставляются в вашу оконную процедуру в промежутке между аппаратными сообщениями клавиатуры. Например, если переключатель <CapsLock> не включен и вы нажимаете и отпускаете клавишу <A>, оконная процедура получит три следующих сообщения:

Сообщение	Клавиша или код
WM_KEYDOWN	Виртуальная клавиша A
WM_CHAR	ASCII код a
WM_KEYUP	Виртуальная клавиша A

Если вы набираете прописное ‘A’, удерживая клавишу <Shift>, нажимая клавишу <A>, отпуская клавишу <A>, и затем отпуская клавишу <Shift>, оконная процедура получит пять сообщений:

Сообщение	Клавиша или код
WM_KEYDOWN	Виртуальная клавиша VK_SHIFT
WM_KEYDOWN	Виртуальная клавиша A
WM_CHAR	ASCII код A
WM_KEYUP	Виртуальная клавиша A
WM_KEYUP	Виртуальная клавиша VK_SHIFT

Сама по себе клавиша <Shift> не вырабатывает символьного сообщения.

Если вы удерживаете клавишу <A> нажатой так, что автоповтор генерирует аппаратные сообщения клавиатуры, то на каждое сообщение WM_KEYDOWN, вы получите символьное сообщение:

Сообщение	Клавиша или код
WM_KEYDOWN	Виртуальная клавиша A
WM_CHAR	ASCII код A
WM_KEYDOWN	Виртуальная клавиша A
WM_CHAR	ASCII код A
WM_KEYDOWN	Виртуальная клавиша A
WM_CHAR	ASCII код A
WM_KEYDOWN	Виртуальная клавиша A
WM_CHAR	ASCII код A
WM_KEYUP	Виртуальная клавиша A

Если у некоторых сообщений WM_KEYDOWN счетчик повторений больше 1, то у соответствующих сообщений WM_CHAR счетчик повторений будет иметь то же значение.

Клавиша <Ctrl> в сочетании с буквенной клавишей генерирует управляющие коды ASCII от 01H (<Ctrl>+<A>) до 1AH (<Ctrl>+<Z>). Для выработки таких управляющих кодов вы можете также использовать и другие клавиши. В следующей таблице показано значение *wParam* в сообщении WM_CHAR для клавиш, вырабатывающих управляющие коды.

Клавиша	ASCII код	Дублирующая комбинация клавиш
Backspace	08H	Ctrl-H
Tab	09H	Ctrl-I
Ctrl-Enter	0AH	Ctrl-J
Enter	0DH	Ctrl-M
Esc	1BH	Ctrl-[

В программах для Windows клавиша <Ctrl> иногда используется с клавишами букв в качестве быстрых клавиш, в таком случае сообщения от буквенных клавиш не преобразуются в символьные сообщения.

Сообщения WM_CHAR

Если вашей Windows-программе необходимо обрабатывать символы клавиатуры (например, в программах обработки текстов или коммуникационных программах), то она будет обрабатывать сообщения WM_CHAR. Вероятно, вы захотите как-то по особому обрабатывать клавиши <Backspace>, <Tab> и <Enter> (и может быть клавишу <Linefeed>), но все остальные символы вы будете обрабатывать похожим образом:

```
case WM_CHAR:
       switch(wParam)
       {
       case '\b':           // забой(Backspace)
```

```
            [другие строки программы]
            break;

        case '\t':           // табуляция(Tab)
            [другие строки программы]
            break;

        case '\n':           // перевод строки(Linefeed)
            [другие строки программы]
            break;

        case '\r':           // возврат каретки(Enter)
            [другие строки программы]
            break;

        default:             // символьный код
            [другие строки программы]
            break;
        }

return 0;
```

Этот фрагмент программы фактически идентичен обработке символов клавиатуры в обычных программах MS_DOS.

Сообщения немых символов

Программы для Windows обычно могут игнорировать сообщения WM_DEADCHAR и WM_SYSDEADCHAR. На некоторых, не американских клавиатурах, некоторые клавиши определяются добавлением диакритического знака к букве. Они называются "немыми клавишами" (dead keys), поскольку эти клавиши сами по себе не определяют символов. Например, при инсталляции немецкой клавиатуры, клавиша, находящаяся там же, где в американской клавиатуре находится клавиша <+/=>, становится немой клавишей для указания высокого звука (ʹ), при ненажатой клавише сдвига, и низкого звука (`), при нажатой клавише сдвига.

Если пользователь нажимает немую клавишу, оконная процедура получает сообщение WM_DEADCHAR с параметром *wParam* равным коду ASCII самого диакритического знака. Когда затем пользователь нажимает клавишу буквы (например, клавишу <A>), оконная процедура получает сообщение WM_CHAR, где параметр *wParam* равен коду ASCII буквы с диакритическим знаком. Таким образом, ваша программа не должна обрабатывать сообщение WM_DEADCHAR, поскольку сообщение WM_CHAR и так дает программе всю необходимую информацию. Windows имеет даже встроенную систему отслеживания ошибок: если за немой клавишей следует буква, у которой не может быть диакритического знака (например буква s), то оконная процедура получает два сообщения WM_CHAR подряд — первое с *wParam* равным коду ASCII самого диакритического знака (такое же значение *wParam*, как было передано с сообщением WM_DEADCHAR) и второе *wParam* равным коду ASCII буквы s.

Взгляд на сообщения от клавиатуры

Если вы захотели бы узнать, как Windows посылает сообщения клавиатуры в программу, то программа KEYLOOK (показанная на рис. 5.3) вам поможет. Эта программа выводит в рабочую область всю информацию, которую Windows посылает оконной процедуре для восьми различных сообщений клавиатуры.

KEYLOOK.MAK

```
#-----------------------
# KEYLOOK.MAK make file
#-----------------------

keylook.exe : keylook.obj
     $(LINKER) $(GUIFLAGS) -OUT:keylook.exe keylook.obj $(GUILIBS)
keylook.obj : keylook.c
     $(CC) $(CFLAGS) keylook.c
```

KEYLOOK.C

```
/*-----------------------------------------------------
   KEYLOOK.C -- Displays Keyboard and Character Messages
               (c) Charles Petzold, 1996
  -----------------------------------------------------*/
```

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

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

RECT rect;
int  cxChar, cyChar;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "KeyLook";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Keyboard Message Looker",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

void ShowKey(HWND hwnd, int iType, char *szMessage,
              WPARAM wParam, LPARAM lParam)
     {
     static char *szFormat[2] = { "%-14s %3d    %c %6u %4d %3s %3s %4s %4s",
                                  "%-14s    %3d %c %6u %4d %3s %3s %4s %4s" };
     char        szBuffer[80];
     HDC         hdc;

     ScrollWindow(hwnd, 0, -cyChar, &rect, &rect);
     hdc = GetDC(hwnd);

     SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));

     TextOut(hdc, cxChar, rect.bottom - cyChar, szBuffer,
              wsprintf(szBuffer, szFormat [iType],
```

```
                        szMessage, wParam,
                       (BYTE)(iType ? wParam : ' '),
                        LOWORD(lParam),
                        HIWORD(lParam) & 0xFF,
                       (PSTR)(0x01000000 & lParam ? "Yes"  : "No"),
                       (PSTR)(0x20000000 & lParam ? "Yes"  : "No"),
                       (PSTR)(0x40000000 & lParam ? "Down" : "Up"),
                       (PSTR)(0x80000000 & lParam ? "Up"   : "Down")));

     ReleaseDC(hwnd, hdc);
     ValidateRect(hwnd, NULL);
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char szTop[] =
               "Message        Key Char Repeat Scan Ext ALT Prev Tran";
     static char szUnd[] =
               "_______        ___ ____ ______ ____ ___ ___ ____ ____";
     HDC         hdc;
     PAINTSTRUCT ps;
     TEXTMETRIC  tm;

     switch(iMsg)
          {
          case WM_CREATE :
               hdc = GetDC(hwnd);

               SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));

               GetTextMetrics(hdc, &tm);
               cxChar = tm.tmAveCharWidth;
               cyChar = tm.tmHeight;

               ReleaseDC(hwnd, hdc);

               rect.top = 3 * cyChar / 2;
               return 0;

          case WM_SIZE :
               rect.right  = LOWORD(lParam);
               rect.bottom = HIWORD(lParam);
               UpdateWindow(hwnd);
               return 0;

          case WM_PAINT :
               InvalidateRect(hwnd, NULL, TRUE);
               hdc = BeginPaint(hwnd, &ps);

               SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));
               SetBkMode(hdc, TRANSPARENT);
               TextOut(hdc, cxChar, cyChar / 2, szTop,(sizeof szTop) - 1);
               TextOut(hdc, cxChar, cyChar / 2, szUnd,(sizeof szUnd) - 1);
               EndPaint(hwnd, &ps);
               return 0;

          case WM_KEYDOWN :
               ShowKey(hwnd, 0, "WM_KEYDOWN", wParam, lParam);
               return 0;

          case WM_KEYUP :
               ShowKey(hwnd, 0, "WM_KEYUP", wParam, lParam);
               return 0;
```

```
        case WM_CHAR :
             ShowKey(hwnd, 1, "WM_CHAR", wParam, lParam);
             return 0;

        case WM_DEADCHAR :
             ShowKey(hwnd, 1, "WM_DEADCHAR", wParam, lParam);
             return 0;

        case WM_SYSKEYDOWN :
             ShowKey(hwnd, 0, "WM_SYSKEYDOWN", wParam, lParam);
             break;         // ie, call DefWindowProc

        case WM_SYSKEYUP :
             ShowKey(hwnd, 0, "WM_SYSKEYUP", wParam, lParam);
             break;         // ie, call DefWindowProc

        case WM_SYSCHAR :
             ShowKey(hwnd, 1, "WM_SYSCHAR", wParam, lParam);
             break;         // ie, call DefWindowProc

        case WM_SYSDEADCHAR :
             ShowKey(hwnd, 1, "WM_SYSDEADCHAR", wParam, lParam);
             break;         // ie, call DefWindowProc

        case WM_DESTROY :
             PostQuitMessage(0);
             return 0;
        }
   return DefWindowProc(hwnd, iMsg, wParam, lParam);
   }
```

Рис. 5.3 Программа KEYLOOK

Дисплей в программе KEYLOOK используется также, как устаревшее устройство вывода информации — телетайп. Когда KEYLOOK получает аппаратные сообщения клавиатуры, вызывается функция *ScrollWindow* для прокрутки содержимого всей рабочей области окна так, чтобы это содержимое сместилось вверх на высоту одного символа. Функция *TextOut* используется для вывода строки новой информации на экран, начиная с высоты одного символа от нижнего края рабочей области. Это почти также просто, как может происходить вывод информации на телетайпе. На рис. 5.4 показано, как выглядит окно программы KEYLOOK, когда вы печатаете слово "Windows". В первом столбце показаны сообщения клавиатуры, во втором — коды виртуальных клавиш для аппаратных сообщений клавиатуры, в третьем — коды символов (и сами символы) для символьных сообщений и, наконец, в шести оставшихся столбцах показано состояние шести полей параметра сообщения *lParam*.

Keyboard Message Looker

Message	Key	Char	Repeat	Scan	Ext	ALT	Prev	Tran
WM_KEYDOWN	16		1	54	No	No	Up	Down
WM_KEYDOWN	87		1	17	No	No	Up	Down
WM_CHAR		87 W	1	17	No	No	Up	Down
WM_KEYUP	87		1	17	No	No	Down	Up
WM_KEYUP	16		1	54	No	No	Down	Up
WM_KEYDOWN	73		1	23	No	No	Up	Down
WM_CHAR		105 i	1	23	No	No	Up	Down
WM_KEYUP	73		1	23	No	No	Down	Up
WM_KEYDOWN	78		1	49	No	No	Up	Down
WM_CHAR		110 n	1	49	No	No	Up	Down
WM_KEYUP	78		1	49	No	No	Down	Up
WM_KEYDOWN	68		1	32	No	No	Up	Down
WM_CHAR		100 d	1	32	No	No	Up	Down
WM_KEYUP	68		1	32	No	No	Down	Up
WM_KEYDOWN	79		1	24	No	No	Up	Down
WM_CHAR		111 o	1	24	No	No	Up	Down
WM_KEYUP	79		1	24	No	No	Down	Up
WM_KEYDOWN	87		1	17	No	No	Up	Down
WM_CHAR		119 w	1	17	No	No	Up	Down
WM_KEYUP	87		1	17	No	No	Down	Up
WM_KEYDOWN	83		1	31	No	No	Up	Down
WM_CHAR		115 s	1	31	No	No	Up	Down
WM_KEYUP	83		1	31	No	No	Down	Up

Start | MS-DOS Prompt | Keyboard Message ... | DIB Manager | 1:29 PM

Рис. 5.4 Вывод на экран программы KEYLOOK

В большей части KEYLOOK.C используются те возможности Windows, которые уже были рассмотрены в различных программах SYSMETS, но также здесь используются несколько новых функций. Отметьте однако, что форматирование выводимого текста программы KEYLOOK по столбцам было бы затруднительно при

использовании задаваемого по умолчанию пропорционального шрифта. Для того, чтобы получить выравненное изображение, код для вывода каждой строки пришлось бы разбить на девять секций. Чтобы избежать всех этих трудностей, гораздо легче просто использовать фиксированный шрифт. Как уже говорилось в последней главе, для этого нужны две функции, которые здесь объединены в одну инструкцию:

```
SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));
```

Программа KEYLOOK вызывает эти две функции везде, где она получает контекст устройства. Это происходит в трех местах: в функции *ShowKey*, при обработке сообщения WM_CREATE в *WndProc* и при обработке сообщения WM_PAINT. Функция *GetStockObject* получает описатель стандартного графического объекта, каковым является фиксированный шрифт, используемый в ранних версиях Windows, предшествовавших Windows 3.0. Вызов функции *SelectObject* выбирает этот объект в контекст устройства. Благодаря этому вызову, весь текст будет выводиться на экран фиксированным шрифтом. Вернуться обратно к пропорциональному шрифту можно с помощью функции:

```
SelectObject(hdc, GetStockObject(SYSTEM_FONT));
```

Функция *ShowKey* вызывает *ScrollWindow* для прокрутки вверх предыдущих строк перед выводом новой строки. Обычно это приводит к тому, что часть окна становится недействительной, и следовательно генерируется сообщение WM_PAINT. Для того, чтобы этого избежать, в функцию *ShowKey* включен вызов функции *ValidateRect*.

Программа KEYLOOK не хранит полученные аппаратные сообщения клавиатуры, поэтому по получении сообщения WM_PAINT она не может перерисовать окно. По этой причине KEYLOOK просто выводит заголовок таблицы в верхней части рабочей области при обработке сообщения WM_PAINT. Перед вызовом функции *BeginPaint*, при обработке сообщения WM_PAINT, KEYLOOK делает недействительным все окно. Это позволяет стереть все окно, вместо того, чтобы стирать недействительный прямоугольник.

(То, что программа KEYLOOK не хранит полученные аппаратные сообщения клавиатуры, и следовательно не может перерисовать окно, пока обрабатывается сообщение WM_PAINT, несомненно является недостатком. В программе TYPER, показанной далее в этой главе, эта недостаток устранен.)

Вверху рабочей области программа KEYLOOK рисует заголовок таблицы и, таким образом, идентифицирует девять столбцов. Хотя можно создать шрифт, в котором символы будут подчеркнуты, здесь применяется немного другой подход. Определены две переменные типа строка символов, которые называются *szTop* (в ней содержится текст) и *szUnd* (в ней содержатся символы подчеркивания) и при обработке сообщения WM_PAINT они выводятся в одну и ту же позицию в верхней части окна. Обычно Windows выводит текст в режиме "opaque", означающим, что Windows обновляет область фона символа при его выводе на экран. Использование этого режима фона может привести к тому, что вторая символьная строка (*szUnd*) сотрет первую (*szTop*). Чтобы предотвратить это, переключите контекст устройства в режим "transparent" (режим без заполнения фона символов):

```
SetBkMode(hdc, TRANSPARENT);
```

Каретка (не курсор)

Когда вы в программе набираете текст, обычно маленький символ подчеркивания или маленький четырехугольник показывает вам место, где следующий набираемый вами символ появится на экране. Вы можете считать, что это курсор (cursor), но, если вы программируете для Windows, необходимо отказаться от такого представления. В Windows это называется "каретка" (caret). Слово же "курсор" относится к битовому образу, отражающему положение мыши на экране.

Функции работы с кареткой

Здесь перечислены пять основных функций работы с кареткой:

- *CreateCaret* — создает связанную с окном каретку.
- *SetCaretPos* — устанавливает положение каретки в окне.
- *ShowCaret* — показывает каретку.
- *HideCaret* — прячет каретку.
- *DestroyCaret* — удаляет каретку.

Кроме этих, еще имеется функция получения положения каретки (*GetCaretPos*) и функции установки и получения частоты мигания каретки (*GetCaretBlinkTime*) и (*SetCaretBlinkTime*).

Каретка — это обычно горизонтальная черточка или прямоугольник, имеющие размер символа, или вертикальная черточка. Вертикальная черточка рекомендуется при использовании пропорционального шрифта, такого как задаваемый по умолчанию системный шрифт Windows. Поскольку размер символов пропорционального шрифта не фиксирован, горизонтальной черточке и прямоугольнику невозможно задать размер символа.

Вы не можете просто создать каретку при обработке сообщения WM_CREATE и удалить ее при обработке сообщения WM_DESTROY. Каретка — это то, что называется "общесистемным ресурсом" (systemwide resource). Это означает, что в системе имеется только одна каретка. И, как результат, программа при необходимости вывода каретки на экран своего окна "заимствует" ее у системы.

Серьезно ли это необычное ограничение? Конечно нет. Подумайте: появление каретки в окне имеет смысл только в том случае, если окно имеет фокус ввода. Каретка показывает пользователю, что он может вводить в программу текст. В каждый конкретный момент времени только одно окно имеет фокус ввода, поэтому для всей системы и нужна только одна каретка.

Обрабатывая сообщения WM_SETFOCUS и WM_KILLFOCUS, программа может определить, имеет ли она фокус ввода. Оконная процедура получает сообщение WM_SETFOCUS, когда получает фокус ввода, а сообщение WM_KILLFOCUS, когда теряет фокус ввода. Эти сообщения приходят попарно: оконная процедура получает сообщение WM_SETFOCUS всегда до того, как получит сообщение WM_KILLFOCUS, и она всегда получает одинаковое количество сообщений WM_SETFOCUS и WM_KILLFOCUS за время существования окна.

Основное правило использования каретки выглядит просто: оконная процедура вызывает функцию *CreateCaret* при обработке сообщения WM_SETFOCUS и функцию *DestroyCaret* при обработке сообщения WM_KILLFOCUS.

Имеется несколько других правил: каретка создается скрытой. После вызова функции *CreateCaret*, оконная процедура должна вызвать функцию *ShowCaret*, чтобы сделать каретку видимой. В дополнение к этому, оконная процедура, когда она рисует в своем окне при обработке сообщения, отличного от WM_PAINT, должна скрыть каретку, вызвав функцию *HideCaret*. После того как оконная процедура закончит рисовать в своем окне, она вызывает функцию *ShowCaret*, чтобы снова вывести каретку на экран. Функция *HideCaret* имеет дополнительный эффект: если вы несколько раз вызываете *HideCaret*, не вызывая при этом *ShowCaret*, то чтобы каретка снова стала видимой, вам придется такое же количество раз вызвать функцию *ShowCaret*.

Программа TYPER

Программа TYPER, представленная на рис. 5.5, объединяет вместе многое из того, о чем мы узнали в этой главе. Программу TYPER можете считать сильно устаревшим текстовым редактором. Вы можете набрать в окне текст, перемещая курсор (имеется ввиду каретка) по экрану с помощью клавиш управления курсором (или клавишами управления кареткой?), и стереть содержимое окна, нажав <Esc>. Окно также очистится при изменении размеров окна. Здесь нет ни прокрутки, ни поиска и замены, ни возможности сохранять файлы, ни контроля правописания, но это только начало.

Для простоты в программе TYPER используется фиксированный шрифт. Создание текстового редактора для пропорционального шрифта является, как вы могли бы сообразить, гораздо более трудным делом. Программа получает контекст устройства в нескольких местах: при обработке сообщений WM_CREATE, WM_KEYDOWN, WM_CHAR и WM_PAINT. Каждый раз при этом для выбора фиксированного шрифта вызываются функции *GetStockObject* и *SelectObject*.

При обработке сообщения WM_SIZE программа TYPER рассчитывает ширину и высоту окна в символах и хранит эти значения в переменных *cxBuffer* и *cyBuffer*. Затем она использует функцию *malloc*, чтобы выделить буфер для хранения всех символов, которые могут быть напечатаны в окне. В переменных *xCaret* и *yCaret* сохраняется положение каретки в символах.

При обработке сообщения WM_SETFOCUS программа TYPER вызывает функцию *CreateCaret* для создания каретки, имеющей ширину и высоту символа, функцию *SetCaretPos* для установки положения каретки и функцию *ShowCaret*, чтобы сделать каретку видимой. При обработке сообщения WM_KILLFOCUS программа вызывает функции *HideCaret* и *DestroyCaret*.

TYPER.MAK

```
#---------------------
# TYPER.MAK make file
#---------------------

typer.exe : typer.obj
     $(LINKER) $(GUIFLAGS) -OUT:typer.exe typer.obj $(GUILIBS)

typer.obj : typer.c
     $(CC) $(CFLAGS) typer.c
```

TYPER.C

```
/*--------------------------------------
   TYPER.C -- Typing Program
```

```
            (c) Charles Petzold, 1996
  --------------------------------------*/

#include <windows.h>
#include <stdlib.h>

#define BUFFER(x,y) *(pBuffer + y * cxBuffer + x)

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Typer";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Typing Program",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char *pBuffer = NULL;
     static int  cxChar, cyChar, cxClient, cyClient, cxBuffer, cyBuffer,
                 xCaret, yCaret;
     HDC         hdc;
     int         x, y, i;
     PAINTSTRUCT ps;
     TEXTMETRIC  tm;

     switch(iMsg)
          {
          case WM_CREATE :
               hdc = GetDC(hwnd);
```

```
        SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));
        GetTextMetrics(hdc, &tm);
        cxChar = tm.tmAveCharWidth;
        cyChar = tm.tmHeight;

        ReleaseDC(hwnd, hdc);
        return 0;
    case WM_SIZE :
                        // obtain window size in pixels

        cxClient = LOWORD(lParam);
        cyClient = HIWORD(lParam);

                        // calculate window size in characters

        cxBuffer = max(1, cxClient / cxChar);
        cyBuffer = max(1, cyClient / cyChar);

                        // allocate memory for buffer and clear it

        if(pBuffer != NULL)
            free(pBuffer);

        if((pBuffer =(char *) malloc(cxBuffer * cyBuffer)) == NULL)
            MessageBox(hwnd, "Window too large.  Cannot "
                             "allocate enough memory.", "Typer",
                             MB_ICONEXCLAMATION | MB_OK);
        else
            for(y = 0; y < cyBuffer; y++)
                for(x = 0; x < cxBuffer; x++)
                    BUFFER(x,y) = ' ';

                        // set caret to upper left corner
        xCaret = 0;
        yCaret = 0;

        if(hwnd == GetFocus())
            SetCaretPos(xCaret * cxChar, yCaret * cyChar);

        return 0;

    case WM_SETFOCUS :
                        // create and show the caret

        CreateCaret(hwnd, NULL, cxChar, cyChar);
        SetCaretPos(xCaret * cxChar, yCaret * cyChar);
        ShowCaret(hwnd);
        return 0;

    case WM_KILLFOCUS :
                        // hide and destroy the caret
        HideCaret(hwnd);
        DestroyCaret();
        return 0;

    case WM_KEYDOWN :
        switch(wParam)
            {
            case VK_HOME :
                xCaret = 0;
                break;

            case VK_END :
```

```
                    xCaret = cxBuffer - 1;
                    break;

               case VK_PRIOR :
                    yCaret = 0;
                    break;

               case VK_NEXT :
                    yCaret = cyBuffer - 1;
                    break;

               case VK_LEFT :
                    xCaret = max(xCaret - 1, 0);
                    break;

               case VK_RIGHT :
                    xCaret = min(xCaret + 1, cxBuffer - 1);
                    break;

               case VK_UP :
                    yCaret = max(yCaret - 1, 0);
                    break;

               case VK_DOWN :
                    yCaret = min(yCaret + 1, cyBuffer - 1);
                    break;

               case VK_DELETE :
                    for(x = xCaret; x < cxBuffer - 1; x++)
                         BUFFER(x, yCaret) = BUFFER(x + 1, yCaret);

                    BUFFER(cxBuffer - 1, yCaret) = ' ';

                    HideCaret(hwnd);
                    hdc = GetDC(hwnd);

                    SelectObject(hdc,
                         GetStockObject(SYSTEM_FIXED_FONT));

                    TextOut(hdc, xCaret * cxChar, yCaret * cyChar,
                             & BUFFER(xCaret, yCaret),
                             cxBuffer - xCaret);

                    ShowCaret(hwnd);
                    ReleaseDC(hwnd, hdc);
                    break;
               }

          SetCaretPos(xCaret * cxChar, yCaret * cyChar);
          return 0;

     case WM_CHAR :
          for(i = 0; i <(int) LOWORD(lParam); i++)
               {
               switch(wParam)
                    {
                    case '\b' :                    // backspace
                         if(xCaret > 0)
                              {
                              xCaret--;
                              SendMessage(hwnd, WM_KEYDOWN,
                                          VK_DELETE, 1L);
                              }
                         break;
```

```
                case '\t' :                    // tab
                     do
                          {
                          SendMessage(hwnd, WM_CHAR, ' ', 1L);
                          }
                     while(xCaret % 8 != 0);
                     break;

                case '\n' :                    // line feed
                     if(++yCaret == cyBuffer)
                          yCaret = 0;
                     break;

                case '\r' :                    // carriage return
                     xCaret = 0;

                     if(++yCaret == cyBuffer)
                          yCaret = 0;
                     break;

                case '\x1B' :                  // escape
                     for(y = 0; y < cyBuffer; y++)
                          for(x = 0; x < cxBuffer; x++)
                               BUFFER(x, y) = ' ';

                     xCaret = 0;
                     yCaret = 0;

                     InvalidateRect(hwnd, NULL, FALSE);
                     break;
                default :                       // character codes
                     BUFFER(xCaret, yCaret) =(char) wParam;

                     HideCaret(hwnd);
                     hdc = GetDC(hwnd);

                     SelectObject(hdc,
                          GetStockObject(SYSTEM_FIXED_FONT));

                     TextOut(hdc, xCaret * cxChar, yCaret * cyChar,
                             & BUFFER(xCaret, yCaret), 1);

                     ShowCaret(hwnd);
                     ReleaseDC(hwnd, hdc);

                     if(++xCaret == cxBuffer)
                          {
                          xCaret = 0;

                          if(++yCaret == cyBuffer)
                               yCaret = 0;
                          }
                     break;
                }
           }

      SetCaretPos(xCaret * cxChar, yCaret * cyChar);
      return 0;

 case WM_PAINT :
      hdc = BeginPaint(hwnd, &ps);
      SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));
```

```
          for(y = 0; y < cyBuffer; y++)
               TextOut(hdc, 0, y * cyChar, & BUFFER(0,y), cxBuffer);

          EndPaint(hwnd, &ps);
          return 0;

     case WM_DESTROY :
          PostQuitMessage(0);
          return 0;
     }
return DefWindowProc(hwnd, iMsg, wParam, lParam);
}
```

Рис. 5.5 Программа TYPER

Сообщения WM_KEYDOWN и WM_KEYUP здесь обрабатываются более полно. Обработка сообщения WM_KEYDOWN в основном включает в себя обработку клавиш управления курсором. Клавиши <Home> и <End> заставляют каретку переместиться в начало или конец строки, клавиши <PageUp> и <PageDown> — к верхней или нижней границе окна. Клавиши стрелок работают так, как вы и ожидаете. При нажатии клавиши <Delete> программа TYPER должна переместить все, что находится в буфере, начиная от следующей позиции каретки и до конца строки, а затем вывести на экран в конец строки символ пробела.

Обработчик сообщения WM_CHAR включает в себя обработку клавиш <Backspace>, <Tab>, <Linefeed> (<Ctrl>+<Enter>), <Enter>, <Escape> и символьных клавиш. Отметьте, что поле Repeat Count параметра *lParam* использовано при обработке сообщения WM_CHAR (здесь предполагается, что важен каждый вводимый пользователем символ), а не при обработке сообщения WM_KEYDOWN (чтобы предотвратить нечаянное двойное нажатие). Обработка нажатий <Backspace> и <Tab> отчасти упрощена путем использования функции *SendMessage*. Логика обработки клавиши <Backspace> заменяется логикой обработки <Delete>, а клавише <Tab> ставится в соответствие несколько пробелов.

Как уже упоминалось, во время рисования в окне при обработке отличного от WM_PAINT сообщения, вы должны сделать каретку невидимой. В программе это делается при обработке сообщения WM_KEYDOWN для клавиши <Delete> и сообщения WM_CHAR для символьных клавиш. В обоих этих случаях, в программе TYPER меняется содержимое буфера, а затем в окне рисуется новый символ или символы.

Программу TYPER можно использовать при работе над выступлениями, что и показано на рис. 5.6.

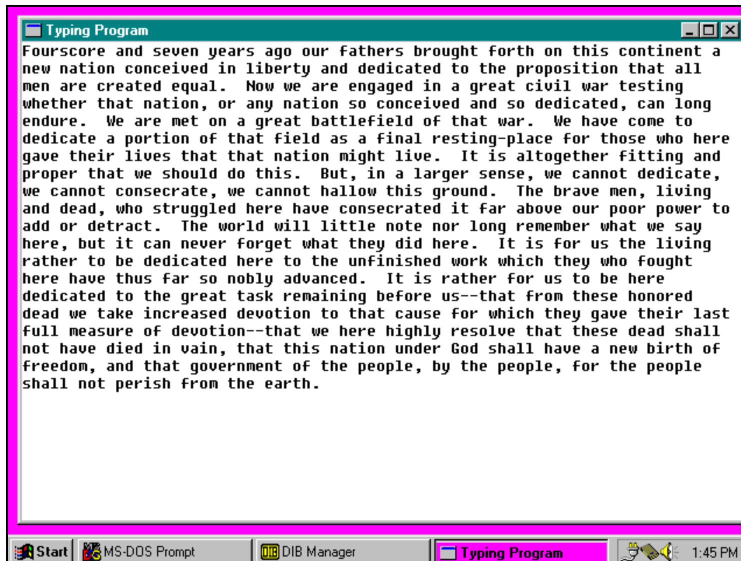


Рис. 5.6 Вид экрана программы TYPER

Наборы символов Windows

Уже упоминалось, что буквенные клавиши, если их нажатию предшествует нажатие клавиши немого символа, вырабатывают сообщения WM_CHAR, где параметр *wParam* является кодом ASCII для символа с диакритическим знаком. Это может вызвать легкое замешательство, поскольку в наборе кодов ASCII отсутствуют какие бы то ни было символы с диакритическими знаками. Так что же на самом деле представляет собой параметр *wParam*? Ответ

на этот вопрос требует, чтобы мы разобрались с наборами символов, что, как вначале может показаться, больше относится к шрифтам. Однако, эта тема также жизненно важна и для обработки клавиатуры.

Стандартный 7-битный набор символов ASCII определяет коды от 0 до 31 (0x1F) и 127 (0x7F) как управляющие символы, а также коды от 32 (0x20) до 126 (0x7E) как символы, которые могут быть выведены на экран. Здесь нет ни одного символа с диакритическим знаком. Поскольку в персональных компьютерах используются байты, состоящие из 8 битов, то производители компьютеров часто определяют наборы символов, использующие 256 кодов вместо 128 кодов ASCII. Дополнительные коды могут назначаться символам с диакритическими знаками. В итоге получается "расширенный набор символов" (extended character set), который включает в себя набор символов ASCII и до 128 других символов.

Если бы в Windows поддерживался такой расширенный набор символов, выводить на экран символы с диакритическими знаками было бы просто. Но в Windows обычный расширенный набор символов не поддерживается. В Windows поддерживается два расширенных набора символов. К несчастью, наличие двух наборов символов не делает их использование вдвое проще.

Набор символов OEM

Для начала давайте обратимся к аппаратуре, на которой работает Windows — к персональным компьютерам IBM и совместимым с ними. В начале 80-х годов производители IBM PC решили расширить набор символов ASCII так, как показано на рис. 5.7. Коды от 0x20 до 0x7E — это выводимые на дисплей символы из набора символов ASCII. Оставшиеся являются нестандартными или, по крайней мере, тогда являлись нестандартными.

Этот набор символов не может игнорироваться. Он закодирован в миллионах микросхем ПЗУ в видеоадаптерах, принтерах и микросхемах BIOS. Он был растиражирован в аппаратуре многочисленных производителей IBM-совместимых компьютеров и периферии. Этот набор символов стал частью того, что обозначается фразой "стандарт IBM". Для множества программ, работающих в текстовом режиме и написанных не для Windows, требуется этот расширенный набор символов, поскольку в них для вывода информации на экран используются символы псевдографики — символы блоков и линий (коды от B0H до DFH).

Здесь есть только одна проблема: расширенный набор символов IBM не предназначен для Windows. Во-первых, символы псевдографики, которые обычно используются в программах персональных компьютеров для приложений, работающих в текстовом режиме, в Windows не нужны, поскольку Windows работает с настоящей графикой. Если вы хотите нарисовать в Windows горизонтальную линию, то гораздо легче нарисовать эту линию, а не выводить на экран строку символов с кодом 0xC4. Во-вторых, греческий алфавит и математические символы менее важны для Windows, чем буквы с символами ударения, которые используются в большинстве европейских языков. Программы, для которых нужен вывод на экран математических символов, гораздо лучше их рисуют с помощью графических функций.

Короче говоря, Windows поддерживает набор символов IBM, но им придается второстепенное значение — в основном он используется в старых приложениях, работающих в окне. Приложения Windows обычно не используют набор символов IBM. В документации по Windows набор символов IBM упоминается как "набор символов OEM" (OEM character set). Набор символов OEM более точно определяется как набор символов национального алфавита для машины, работающей под Windows.

Поддержка языков различных стран в DOS

Имеется несколько вариантов набора символов IBM, которые называются "кодовые страницы" (code pages). Вариант, используемый в Соединенных Штатах и большинстве европейских стран, называется Code Page 437. В системах, продаваемых в Норвегии, Дании, Португалии и некоторых других странах Европы, используются другие специальные кодовые страницы, в которых содержится больше специальных символов, необходимых для языков этих стран. Недавно некоторые из этих стран начали использовать Code Page 850, в которой содержится меньше графических символов, а больше букв со значками типа знаков ударения и других специальных символов.

Windows поддерживает кодовые страницы, устанавливая шрифты OEM (которые используются работающими в окне приложениями DOS и в программе просмотра буфера обмена), которые соответствуют кодовой странице системы, и устанавливая соответствующие таблицы преобразования для функций *CharToOem* и *OemToChar* (о которых будет рассказано позднее).

Программа установки Windows Setup выберет нужные кодовые страницы на основе региональных установок текущей конфигурации системы.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00:		☺	☻	♥	♦	♣	♠	•	◘	○	◙	♂	♀	♪	♫	☼
10:	►	◄	↕	‼	¶	§	▬	↨	↑	↓	→	←	∟	↔	▲	▼
20:		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30:	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40:	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50:	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60:	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70:	p	q	r	s	t	u	v	w	x	y	z	{	¦	}	~	⌂
80:	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
90:	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	¢	£	¥	₧	ƒ
A0:	á	í	ó	ú	ñ	Ñ	ª	º	¿	⌐	¬	½	¼	¡	«	»
B0:	░	▒	▓	│	┤	╡	╢	╖	╕	╣	║	╗	╝	╜	╛	┐
C0:	└	┴	┬	├	─	┼	╞	╟	╚	╔	╩	╦	╠	═	╬	╧
D0:	╨	╤	╥	╙	╘	╒	╓	╫	╪	┘	┌	█	▄	▌	▐	▀
E0:	α	β	Γ	π	Σ	σ	µ	τ	Φ	Θ	Ω	δ	∞	ø	∈	∩
F0:	≡	±	≥	≤	⌠	⌡	÷	≈	°	∙	·	√	^{n}	^{2}	■	

Рис. 5.7 Расширенный набор символов IBM, упорядоченный по возрастанию значений кода символов

Набор символов ANSI

Расширенный набор символов, который Windows и программы для Windows в большинстве случаев используют, называется "набор символов ANSI" (ANSI character set), но фактически он является стандартом ISO.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00:	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
10:	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
20:		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30:	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40:	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50:	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60:	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70:	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	■
80:	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
90:	■	‘	’	■	■	■	■	■	■	■	■	■	■	■	■	■
A0:		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
B0:	°	±	^{2}	^{3}	´	µ	¶	·	¸	^{1}	º	»	¼	½	¾	¿
C0:	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D0:	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E0:	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F0:	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Рис. 5.8 Набор символов ANSI, упорядоченный по возрастанию значений кода символов

Когда ваша программа получает сообщение WM_CHAR, параметр *wParam* содержит символьный код ANSI. Набор символов ANSI показан на рис. 5.8. Как вы можете видеть, коды от 0x20 до 0x7E представляют собой те же самые символы, которые имеются в наборе символов OEM и наборе символов ASCII. Символы, показанные в виде закрашенных прямоугольников, не определены. Они могут оказаться иными на другом устройстве вывода информации (например, на принтере). Шрифты TrueType определяют для кодов ANSI несколько дополнительных символов в диапазоне от 0x80 до 0x9F.

Наборы символов OEM, ANSI и шрифты

В Windows имеются различные шрифты для вывода на экран символов из набора ANSI и OEM. Когда вы первый раз получаете описатель контекста устройства, то одним из атрибутов контекста устройства является шрифт. По

умолчанию им является SYSTEM_FONT или "системный шрифт" (system font), в котором используется набор символов ANSI. Если вы хотите выводить на экран символы из набора OEM, то вы можете выбрать OEM_FIXED_FONT (также называемый "терминальный шрифт" (terminal font) в контекст устройства, используя следующий оператор:

```
SelectObject(hdc, GetStockObject(OEM_FIXED_FONT));
```

Международные интересы

Здесь рассказывается о том, почему в середине главы, посвященной клавиатуре, нам приходится говорить о шрифтах. Мы установили, что когда пользователь Windows набирает на неамериканской клавиатуре символ с диакритическим знаком, то параметром *wParam* сообщения WM_CHAR является код этого символа из набора символов ANSI.

Поэтому, если вам необходимо получить на экране отображение этого символа, то вам было бы лучше пользоваться шрифтом из набора символов ANSI (таким как SYSTEM_FONT или SYSTEM_FIXED_FONT). Если вы вместо этого используете OEM_FIXED_FONT, то символ, который вы выводите на экран, окажется неправильным, и пользователь будет неприятно удивлен. Несколько других простых правил позволят сохранить логику работы с клавиатурой в вашей программе для Windows при адаптации к рынкам Европы.

Работа с набором символов

Если вы получаете сообщение WM_CHAR, то запомните, что значение параметра *wParam* вполне реально может оказаться больше, чем 128. И это не ошибка. Не думайте, что все, что больше 127 — это неправильные символы.

Вам может понадобиться преобразовать регистр символа, т. е. сделать из строчной буквы — прописную. Не используйте ваш собственный алгоритм:

```
if(ch >= 'a' && ch <= 'z')
       ch -= 32;                    // ОШИБКА!!!
```

Это плохой стиль даже для программ, написанных не для Windows. Но однако нельзя пользоваться и стандартной функцией C:

```
ch = toupper(ch);            // ОШИБКА!!!
```

Обе эти функции работают только с нижней половиной набора символов ANSI. Они не преобразуют 0xE0 в 0xC0.

Вместо них вам следует пользоваться функциями *CharUpper* и *CharLower* Windows. Если *pString* — это оканчивающаяся нулевым символом строка, то вы можете преобразовать ее в верхний регистр с помощью функции *CharUpper*:

```
CharUpper(pString);
```

Для строки, которая не оканчивается нулевым символом, нужно использовать функцию *CharUpperBuff*:

```
CharUpperBuff(pString, nLength);
```

Для преобразования одного символа также можно пользоваться функцией *CharUpper*, но требуется некоторая поправка, поскольку старшее слово параметра должно быть равно 0:

```
ch = CharUpper((PSTR)(LONG)(BYTE)ch);
```

Если *ch* определяется как беззнаковый символ, то преобразование типа BYTE не требуется. Кроме вышеперечисленных, в Windows используются функции *CharLower* и *CharLowerBuff* для преобразования прописных букв в строчные.

Если вы действительно серьезно намерены писать программы для Windows, которые можно было бы приспосабливать к иностранным языкам, вы должны также изучить функции *CharNext* и *CharPrev*. Эти функции облегчают работу с многобайтными наборами символов, часто используемых в странах Дальнего Востока. Для этих наборов символов требуется больше 256 символов, некоторые из которых задаются двумя байтами. Если вы используете обычную арифметику указателей C для просмотра строки (например, при поиске символа обратной косой черты в строке пути, содержащем каталоги), то можете решить, что нашли нужный символ, хотя фактически вы нашли второй байт двухбайтного символьного кода. Функциям *CharNext* и *CharPrev* передается дальний указатель на символьную строку и они возвращают дальний указатель, который необходимым образом увеличен или уменьшен с учетом последних двухбайтных символьных кодов.

Связь с MS-DOS

Если бы Windows была только одной операционной оболочкой, работающей на машине, то вы могли бы забыть о наборе символов OEM и работать только с набором символов ANSI. Однако пользователи могут создавать файлы в

среде MS-DOS, а использовать их в Windows; они также могут создавать файлы в Windows, а использовать в MS-DOS. К сожалению, в MS-DOS используется набор символов OEM.

Вот пример одной из проблем, которые могут встретиться. Предположим, что говорящий на немецком языке пользователь персонального компьютера создает в MS-DOS файл ÜBUNGEN.TXT "практические упражнения" в программе EDLIN. Для IBM PC буква Ü — это часть набора символов IBM (т. е. OEM) и ее код 154 или 0x9A. (При использовании MS-DOS с американской клавиатурой на IBM PC, вы можете набрать эту букву, напечатав <Alt>+154 на числовой клавиатуре.) MS-DOS использует этот код символа в записи каталога, соответствующей этому файлу.

Если программа для Windows использует вызовы функций MS-DOS для получения каталога файлов и вывода их имен затем прямо на экран с использованием шрифта, содержащего символы из набора символов ANSI, то первая буква ÜBUNGEN.TXT будет изображена в виде закрашенного прямоугольника, поскольку код 154 — это один из неопределенных символов набора символов ANSI. Программе для Windows необходимо преобразовать код 154 (или 0x9A) из расширенного набора символов IBM в код символа 220 (или 0xDC) из набора символов ANSI, который представляет из себя букву Ü. Эти задачи для вас решает функция Windows *OemToChar*. Она получает в качестве параметров два дальних указателя на строки. Символы OEM в первой строке преобразуются в символы ANSI и сохраняются во второй строке:

```
OemToChar(lpszOemStr, lpszAnsiStr);
```

Теперь рассмотрим противоположный пример. Пользователь, говорящий по-немецки, хочет воспользоваться вашей программой, написанной для Windows, для создания файла ÜBUNGEN.TXT. В имени файла, введенном пользователем, первая буква имеет код 220 (или 0xDC). Если вы используете для открытия этого файла вызов функции MS-DOS, то MS-DOS использует этот символ в имени файла. Если потом пользователь, находясь в MS-DOS, посмотрит на этот файл, то первый символ будет выглядеть как прямоугольник. Перед тем как использовать вызов функции MS-DOS, вы должны преобразовать имя файла в набор символов OEM:

```
CharToOem(lpszAnsiStr, lpszOemStr);
```

Этот вызов преобразует код 220 (или 0xDC) в код 154 (или 0x9A). Windows также содержит две функции *CharToOemBuff* и *OemToCharBuff*, для которых символ ноль в конце строки не требуется.

Кроме этих функций для подобных преобразований в Windows имеется функция *OpenFile*. Если вы используете эту функцию, то вам не нужно преобразование с помощью функции *CharToOem*. Если вы используете вызовы функций MS-DOS для получения списка имен файлов (как это делает в Windows программа File Manager), то эти имена файлов, перед их выводом на экран, следует передать в функцию *OemToChar*.

Преобразование содержимого файлов является еще одной проблемой, возникающей, когда файлы используются и в Windows и в MS-DOS. Если в вашей программе для Windows используются файлы, которые, как вы уверены, были созданы в программе для MS-DOS, тогда вам может понадобиться обработать текстовое содержимое этих файлов с помощью функции *OemToChar*. Аналогично, если в программе для Windows подготовлен файл для использования в программе MS-DOS, то для преобразования текста вам может понадобиться функция *CharToOem*.

Функции *OemToChar* и *CharToOem* реализованы в драйвере клавиатуры. В них включены очень простые таблицы. Программа функции *OemToChar* преобразует код OEM от 0x80 до 0xFF в код символа из набора ANSI, который больше всего похож на соответствующий символ OEM. В некоторых случаях, это преобразование является лишь очень грубым приближением. Например, большинство символов псевдографики в наборе символов IBM преобразуется в знаки плюсов, тире и вертикальных линий. Большинство кодов OEM от 0x00 до 0x1F не преобразуются в коды ANSI.

Функция *CharToOem* преобразует коды ANSI от 0xA0 до 0xFF в коды из набора символов OEM. Символы со знаками типа ударения в наборе символов ANSI, которых нет в наборе символов OEM, преобразуются в коды обычных, не имеющих диакритических знаков, символов ASCII.

Использование цифровой клавиатуры

Как вы, вероятно, знаете, клавиатура IBM PC и BIOS позволяет вам вводить коды расширенного набора символов IBM посредством нажатия клавиши <Alt> и набора десятичного кода из трех цифр, представляющего собой код символа OEM, на цифровой клавиатуре. Эта возможность воспроизводится в Windows двумя способами.

Во-первых, когда вы вводите <Alt>-[код OEM] на цифровой клавиатуре, то Windows выдает вам код того символа ANSI (в параметре *wParam* сообщения WM_CHAR), который имеет наибольшее сходство с соответствующим символом OEM, представленным кодом OEM. Вернее, Windows перед тем как выработать сообщение WM_CHAR, обрабатывает код с помощью функции *OemToChar*. Эта возможность очень удобна для пользователя: если у вас нет иноязычной клавиатуры, и вы привыкли печатать Ü с помощью <Alt>+154, то вы можете делать то же самое и в программе для Windows. Вам не нужно переучиваться на коды символов ANSI.

Во-вторых, если вам нужно генерировать коды расширенного набора символов ANSI с помощью американской клавиатуры, наберите <Alt>-0[код OEM] на числовой клавиатуре. Параметр *wParam* сообщения WM_CHAR получит этот код OEM. Таким образом, <Alt>-0220 тоже соответствует Ü. Вы можете попытаться проделать это в программах KEYLOOK или TYPER.

Решение проблемы с использованием системы UNICODE в Windows NT

Производители программ, создающие приложения для международного рынка, вынуждены были иметь дело с нестандартными решениями проблемы 7-разрядного кода ASCII, такими как кодовые страницы и наборы двухбайтных символов. Лучшее решение необходимо, и им может стать Unicode.

Unicode — это кодирование символа, которое использует единообразный 16-разрядный код для каждого символа. Это позволяет получать коды любого символа, написанного на любом языке мира, из тех, которые вероятнее всего будут использоваться в сфере компьютерных коммуникаций, включая иероглифы Китая, Японии и Кореи. Unicode разрабатывался консорциумом компьютерных компаний (включая самые крупные), и документирован в книге *Unicode Standart*, опубликованной издательством Addison-Wesley.

К сожалению, в Windows 95 имеются только некоторые элементы поддержки системы Unicode, и Windows 95 не обеспечивает работы с символами Unicode с помощью драйвера клавиатуры, в отличие от Windows NT, в которой изначально была заложена поддержка Unicode.

Очевидно, что адаптация программ (и умов программистов) к идее 16-разрядных символов — это непростая работа, но она окупится сторицей, если у нас появится возможность выводить на экраны и принтеры персональных компьютеров информацию на всех языках мира. Если вы интересуетесь концепцией и механикой системы кодирования Unicode, реализованной в Windows NT, то вы можете открыть рубрику "Enviroments" в *PC Magazine* за 1993 год, статьи за 26 октября, 9 ноября, 23 ноября и 7 декабря (где они были случайно не указаны в содержании, но тем не менее напечатаны, начиная со страницы 426).

Глава 6 Мышь

Мышь — это графическое устройство ввода информации с одной или более кнопками. Несмотря на многочисленные эксперименты с устройствами для ввода информации, такими как сенсорные экраны или световые перья, мышь (и ее варианты типа трекбола, широко используемого в портативных компьютерах) осталась единственным устройством, которое наиболее широко проникло на рынок персональных компьютеров.

Но так было не всегда. В самом деле, первые разработчики Windows чувствовали, что им нельзя заставлять пользователей покупать мышь для работы со своим программным продуктом. Поэтому они сделали мышь необязательным аксессуаром, и обеспечили доступ ко всем операциям в Windows и ее приложениям через интерфейс клавиатуры. Другим производителям также рекомендовалось дублировать функции мыши с помощью интерфейса клавиатуры.

Хотя мышь стала почти повсеместным атрибутом компьютеров с Windows, эта философия по-прежнему актуальна. Особенно машинистки предпочитают оставлять свои руки на клавиатуре, и предполагается, что каждый из вас когда-нибудь "терял" мышь на заваленном бумагами собственном столе. По этой причине по-прежнему рекомендуется, чтобы везде, где это возможно, вы добавляли интерфейс клавиатуры для дублирования функций мыши.

Базовые знания о мыши

Windows 95 поддерживает однокнопочную, двухкнопочную или трехкнопочную мышь, а также позволяет использовать джойстик или световое перо для имитации однокнопочной мыши. Поскольку однокнопочная мышь является простейшей, то многие программисты, работающие под Windows, традиционно не работают со второй и третьей кнопками. Однако, двухкнопочная мышь стала стандартом де-факто, поэтому традиционная сдержанность в использовании второй кнопки неоправданна. Действительно, вторая кнопка мыши нажимается или для появления "контекстного меню", т. е. меню, которое появляется в окне помимо обычной строки меню, или для специальных операций перетаскивания. (Перетаскивание будет рассмотрено ниже.)

Вы можете определить наличие мыши с помощью функции *GetSystemMetrics*:

```
fMouse = GetSystemMetrics(SM_MOUSEPRESENT);
```

Значение *fMouse* будет равным TRUE (ненулевым), если мышь установлена. Для определения количества кнопок установленной мыши используйте следующий вызов:

```
cButtons = GetSystemMetrics(SM_CMOUSEBUTTONS);
```

Если мышь не инсталлирована, то возвращаемым значением этой функции будет 0.

Пользователи-левши могут поменять назначение кнопок мыши с помощью программы Control Panel. Хотя приложение может определить, было ли такое переключение, передав в функцию *GetSystemMetrics* параметр SM_SWAPBUTTON, но обычно это не нужно. Кнопка, нажимаемая указательным пальцем, считается левой кнопкой, даже если физически она находится на правой стороне мыши. Однако в обучающих программах вы можете нарисовать мышь на экране, и в этом случае вам надо будет узнать, менялось ли назначение кнопок мыши.

Несколько кратких определений

Когда пользователь Windows перемещает мышь, Windows перемещает по экрану маленькую растровую картинку, которая называется "курсор мыши" (mouse cursor). Курсор мыши имеет "вершину" (hot spot) размером в один пиксель, точно указывающее положение мыши на экране.

В драйвере дисплея содержатся несколько ранее определенных курсоров мыши, которые могут использоваться в программах. Наиболее типичным курсором является наклонная стрелка, которая называется IDC_ARROW и определяется в заголовочных файлах Windows. Вершина — это конец стрелки. Курсор IDC_CROSS (используемый в приведенных в этой главе программах BLOKOUT) имеет вершину в центре крестообразного шаблона. Курсор IDC_WAIT в виде песочных часов обычно используется программами для индикации того, что они чем-то заняты. Программисты также могут спроектировать свои собственные курсоры (как это делается в главе 9). Курсор, устанавливаемый по умолчанию, для конкретного окна задается при определении структуры класса окна. Например:

```
wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
```

Ниже приведены определения терминов, соответствующих вашим действиям над кнопками мыши:

- Щелчок — нажатие и отпускание кнопки мыши
- Двойной щелчок — двойное быстрое одно за другим нажатие и отпускание кнопки мыши
- Перетаскивание — перемещение мыши при нажатой кнопке

На трехкнопочной мыши кнопки называются левой кнопкой, средней кнопкой и правой кнопкой. В связанных с мышью идентификаторах, определенных в заголовочных файлах Windows, используются аббревиатуры LBUTTON, MBUTTON и RBUTTON. Двухкнопочная мышь имеет только левую и правую кнопки. Единственная кнопка однокнопочной мыши является левой.

Сообщения мыши, связанные с рабочей областью окна

В предыдущей главе вы видели, как Windows посылает сообщения клавиатуры только тому окну, которое имеет фокус ввода. Сообщения мыши отличаются: оконная процедура получает сообщения мыши и когда мышь проходит через окно и при щелчке внутри окна, даже если окно неактивно или не имеет фокуса ввода. В Windows для мыши определен набор из 21 сообщения. Однако, 11 из этих сообщений не относятся к рабочей области, и программы для Windows обычно игнорируют их.

Если мышь перемещается по рабочей области окна, оконная процедура получает сообщение WM_MOUSEMOVE. Если кнопка мыши нажимается или отпускается внутри рабочей области окна, оконная процедура получает следующие сообщения:

Кнопка	Нажатие	Отпускание	Нажатие (Второй щелчок)
Левая	WM_LBUTTONDOWN	WM_LBUTTONUP	WM_LBUTTONDBLCLK
Средняя	WM_MBUTTONDOWN	WM_MBUTTONUP	WM_MBUTTONDBLCLK
Правая	WM_RBUTTONDOWN	WM_RBUTTONUP	WM_RBUTTONDBLCLK

Ваша оконная процедура получает сообщения "MBUTTON" только при наличии трехкнопочной мыши и сообщения "RBUTTON" только при наличии двух- или трехкнопочной мыши. Оконная процедура получает сообщения "DBLCLK" (двойной щелчок) только в том случае, если класс окна был определен так, чтобы их можно было получать (как описано ниже).

Для всех этих сообщений значение параметра *lParam* содержит положение мыши. Младшее слово — это координата *x*, а старшее слово — координата *y* относительно верхнего левого угла рабочей области окна. Вы можете извлечь координаты *x* и *y* из параметра *lParam* с помощью макросов LOWORD и HIWORD, определенных в заголовочных файлах Windows. Значение параметра *wParam* показывает состояние кнопок мыши и клавиш <Shift> и <Ctrl>. Вы можете проверить параметр *wParam* с помощью битовых масок, определенных в заголовочных файлах. Префикс MK означает "клавиша мыши" (mouse key).

MK_LBUTTON	Левая кнопка нажата
MK_MBUTTON	Средняя кнопка нажата
MK_RBUTTON	Правая кнопка нажата
MK_SHIFT	Клавиша <Shift> нажата
MK_CONTROL	Клавиша <Ctrl> нажата

При движении мыши по рабочей области окна, Windows не вырабатывает сообщение WM_MOUSEMOVE для всех возможных положений мыши. Количество сообщений WM_MOUSEMOVE, которые получает ваша программа, зависит от устройства мыши и от скорости, с которой ваша оконная процедура может обрабатывать сообщения о движении мыши. Вы получите хорошее представление о темпе получения сообщений WM_MOUSEMOVE, когда поэкспериментируете с представленной ниже программой CONNECT.

Если вы щелкните левой кнопкой мыши в рабочей области неактивного окна, Windows сделает активным окно, в котором вы произвели щелчок, и затем передаст оконной процедуре сообщение WM_LBUTTONDOWN. Если ваша

оконная процедура получает сообщение WM_LBUTTONDOWN, то ваша программа может уверенно считать, что ее окно активно. Однако, ваша оконная процедура может получить сообщение WM_LBUTTONUP, не получив вначале сообщения WM_LBUTTONDOWN. Это может случиться, если кнопка мыши нажимается в одном окне, мышь перемещается в ваше окно, и кнопка отпускается. Аналогично, оконная процедура может получить сообщение WM_LBUTTONDOWN без соответствующего ему сообщения WM_LBUTTONUP, если кнопка мыши отпускается во время нахождения в другом окне.

Из этих правил есть два исключения:

- Оконная процедура может "захватить мышь" (capture the mouse) и продолжать получать сообщения мыши, даже если она находится вне рабочей области окна. Позднее в этой главе вы узнаете, как захватить мышь.
- Если системное модальное окно сообщений или системное модальное окно диалога находится на экране, никакая другая программа не может получать сообщения мыши. Системные модальные окна сообщений и диалога запрещают переключение на другое окно программы, пока оно активно. (Примером системного модального окна сообщений является окно, которое появляется, когда вы завершаете работу с Windows.)

Простой пример обработки сообщений мыши

Программа CONNECT, приведенная на рис. 6.1, выполняет достаточно простую обработку сообщений мыши, что позволяет вам получить хорошее представление о том, как Windows посылает сообщения мыши вашей программе.

CONNECT.MAK

```
#-----------------------
# CONNECT.MAK make file
#-----------------------

connect.exe : connect.obj
     $(LINKER) $(GUIFLAGS) -OUT:connect.exe connect.obj $(GUILIBS)

connect.obj : connect.c
     $(CC) $(CFLAGS) connect.c
```

CONNECT.C

```
/*--------------------------------------------------
   CONNECT.C -- Connect-the-Dots Mouse Demo Program
               (c) Charles Petzold, 1996
  --------------------------------------------------*/

#include <windows.h>

#define MAXPOINTS 1000
#define MoveTo(hdc, x, y) MoveToEx(hdc, x, y, NULL)

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Connect";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);
```

```
     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Connect-the-Points Mouse Demo",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static POINT points[MAXPOINTS];
     static int   iCount;
     HDC          hdc;
     PAINTSTRUCT  ps;
     int          i, j;

     switch(iMsg)
          {
          case WM_LBUTTONDOWN :
               iCount = 0;
               InvalidateRect(hwnd, NULL, TRUE);
               return 0;

          case WM_MOUSEMOVE :
               if(wParam & MK_LBUTTON && iCount < 1000)
                    {
                    points[iCount  ].x = LOWORD(lParam);
                    points[iCount++].y = HIWORD(lParam);

                    hdc = GetDC(hwnd);
                    SetPixel(hdc, LOWORD(lParam), HIWORD(lParam), 0L);
                    ReleaseDC(hwnd, hdc);
                    }
               return 0;

          case WM_LBUTTONUP :
               InvalidateRect(hwnd, NULL, FALSE);
               return 0;

          case WM_PAINT :
               hdc = BeginPaint(hwnd, &ps);

               SetCursor(LoadCursor(NULL, IDC_WAIT));
               ShowCursor(TRUE);
               for(i = 0; i < iCount - 1; i++)
                    for(j = i + 1; j < iCount; j++)
                         {
                         MoveTo(hdc, points[i].x, points[i].y);
                         LineTo(hdc, points[j].x, points[j].y);
                         }

               ShowCursor(FALSE);
```

```
            SetCursor(LoadCursor(NULL, IDC_ARROW));

            EndPaint(hwnd, &ps);
            return 0;

        case WM_DESTROY :
            PostQuitMessage(0);
            return 0;
        }
    return DefWindowProc(hwnd, iMsg, wParam, lParam);
    }
```

Рис. 6.1 Программа CONNECT

Программа CONNECT обрабатывает три сообщения мыши:

- WM_LBUTTONDOWN — Программа очищает рабочую область.
- WM_MOUSEMOVE — Если левая кнопка мыши нажата, то программа рисует черную точку в рабочей области в текущем положении мыши.
- WM_LBUTTONUP — Программа соединяет все точки, нарисованные в рабочей области, друг с другом. Иногда в результате этого получается симпатичный рисунок, а иногда — плотно заполненный клубок. (См. рис. 6.2.)

Чтобы воспользоваться программой CONNECT, поместите курсор мыши в рабочую область, нажмите левую кнопку, немного подвигайте мышь, и отпустите левую кнопку. Программа CONNECT лучше работает с кривым шаблоном из нескольких точек, который вы можете нарисовать, быстро двигая мышь при нажатой левой кнопке. В программе используется несколько простых функций интерфейса графического устройства (GDI). Функция *SetPixel* рисует точку размером с один пиксель определенного цвета — в нашем случае черного. (На дисплее с высокой разрешающей способностью, пиксель может быть почти невидим.) Для рисования линий нужны две функции: *MoveTo* отмечает координаты *x* и *y* начала линии, а *LineTo* рисует линию. (Обратите внимание, что *MoveTo* определяется как макрос, использующий функцию *MoveToEx*.)

Если вы перемещаете курсор мыши за пределы рабочей области до того, как отпускаете кнопку, программа CONNECT не соединяет точки между собой, поскольку она не получает сообщения WM_LBUTTONUP. Если вы возвращаете курсор мыши обратно в рабочую область и снова нажимаете левую кнопку, то CONNECT очищает рабочую область. (Если вы хотите продолжить рисование после того, как отпустили кнопку вне рабочей области, то снова нажмите левую кнопку, пока мышь находится вне рабочей области, а затем переместите мышь обратно внутрь.)

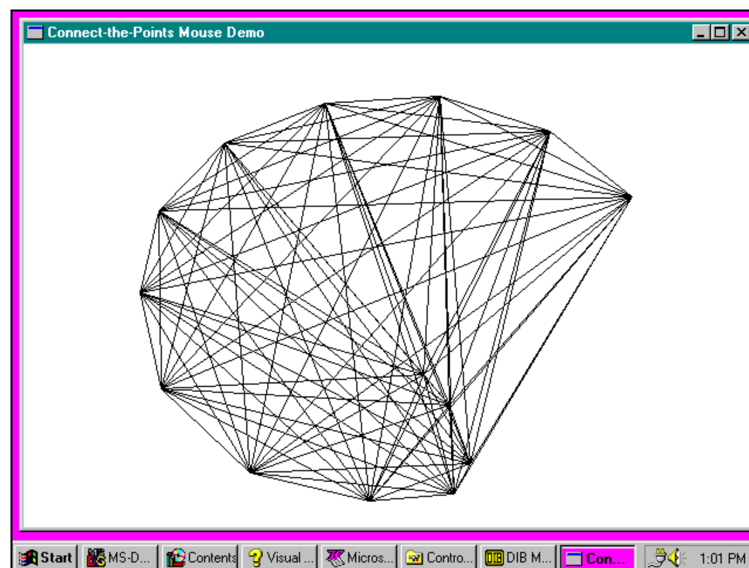


Рис. 6.2 Вывод на экран программы CONNECT

CONNECT хранит информацию о максимум 1000 точках. Число линий, которые рисует программа равно:

$((P) \times (P - 1))/2$

где P — это количество точек. При наличии всех 1000 точек, а это почти 500000 линий, на рисование может уйти несколько минут. Поскольку Windows 95 является вытесняющей многозадачной средой, вы можете в это время переключиться на другие программы. Однако, вы ничего не сможете сделать с программой CONNECT (например,

сдвинуть ее окно или изменить его размер), пока продолжается ее работа. В главе 14 мы изучим методы решения этой проблемы.

Поскольку программе CONNECT для рисования линий может потребоваться некоторое время, при обработке сообщения WM_PAINT она изменяет вид курсора на песочные часы, а после окончания рисования, возвращает курсор в предыдущее состояние. Для этого требуется два вызова функции *SetCursor*, в которых используются два стандартных курсора. В программе также дважды вызывается функция *ShowCursor*, первый раз с параметром TRUE и второй — с параметром FALSE. Позднее в этой главе в разделе "Эмуляция мыши с помощью клавиатуры" об этих вызовах будет рассказано более подробно.

Если программа CONNECT занята рисованием линий, вы можете нажать кнопку мыши, подвигать мышью и отпустить кнопку мыши, но ничего не произойдет. CONNECT не получает эти сообщения, поскольку она занята и не может сделать ни одного вызова *GetMessage*. После того, как программа закончит рисование линий, она опять не получает этих сообщений, поскольку кнопка мыши к этому времени отпущена. В этом отношении мышь не похожа на клавиатуру. Windows обращается с каждой нажатой на клавиатуре клавишей как с чем-то важным. Однако, если кнопка мыши нажата и отпущена в рабочей области, пока программа занята, щелчки мыши просто сбрасываются.

Теперь попытайтесь сделать следующее: пока программа CONNECT долго занимается рисованием, нажмите кнопку мыши и подвигайте курсором. После того, как программа CONNECT закончит рисование, она извлечет сообщение WM_LBUTTONDOWN из очереди сообщений (и обновит рабочую область), поскольку в этот момент кнопка нажата. Однако, она получает только сообщения WM_MOUSEMOVE, возникшие после получения сообщения WM_LBUTTONDOWN.

Иногда слово "слежение" (tracking) используется для ссылки на способ, которым программа обрабатывает движение мыши. Слежение, однако, не значит, что ваша программа остается в цикле своей оконной процедуры, пытаясь отслеживать движение мыши по экрану. Вместо этого оконная процедура обрабатывает каждое приходящее сообщение мыши и тут же быстро заканчивается.

Обработка клавиш <Shift>

Когда программа CONNECT получает сообщение WM_MOUSEMOVE, она выполняет поразрядную операцию AND со значениями *wParam* и MK_LBUTTON для определения того, нажата ли левая кнопка. Вы также можете использовать *wParam* для определения состояния клавиш <Shift>. Например, если обработка должна зависеть от состояния клавиш <Shift> и <Ctrl>, то вы могли бы воспользоваться следующей логикой:

```
if(MK_SHIFT & wParam)
        if(MK_CONTROL & wParam)
        {
                [нажаты клавиши <Shift> и <Ctrl>]
        }
        else
        {
                [нажата клавиша <Shift>]
        }
else
if(MK_CONTROL & wParam)
        {
                [нажата клавиша <Ctrl>]
        }
        else
        {
                [клавиши <Shift> и <Ctrl> не нажаты]
        }
```

Если вы хотите в вашей программе использовать и левую и правую кнопки мыши, и если вы также хотите обеспечить возможность работы пользователям однокнопочной мыши, вы можете так написать вашу программу, чтобы действие клавиши <Shift> в сочетании с левой кнопкой мыши было тождественно действию правой кнопки. В этом случае ваша обработка щелчков кнопки могла бы выглядеть так:

```
case WM_LBUTTONDOWN:
        if(!MK_SHIFT & wParam)
        {
                [логика обработки левой кнопки]
                return 0;
        }
// идем дальше вниз
case WM_RBUTTONDOWN:
        [логика обработки правой кнопки]
```

```
        return 0;
```

Функция *GetKeyState* (описанная в главе 4) также может возвращать состояние кнопок мыши или клавиш <Shift>, используя виртуальные коды клавиш VK_LBUTTON, VK_RBUTTON, VK_MBUTTON, VK_SHIFT и VK_CONTROL. При нажатой кнопке или клавише возвращаемое значение функции *GetKeyState* отрицательно. Функция *GetKeyState* возвращает состояние мыши или клавиши в связи с обрабатываемым в данный момент сообщением, т. е. информация о состоянии должным образом синхронизируется с сообщениями. Но поскольку вы не можете использовать функцию *GetKeyState* для клавиши, которая еще только должна быть нажата, ее нельзя использовать и для кнопки мыши, которая еще только должна быть нажата. Не делайте так:

```
while(GetKeyState(VK_LBUTTON) >= 0);      // ОШИБКА!!!
```

Функция *GetKeyState* сообщит о том, что левая кнопка нажата только в том случае, если левая кнопка уже нажата, когда вы обрабатываете сообщение и вызываете *GetKeyState*.

Двойные щелчки клавиш мыши

Двойным щелчком мыши называются два, следующих один за другим в быстром темпе, щелчка мыши. Для того, чтобы два последовательных щелчка мыши считались двойным щелчком, они должны произойти в течение очень короткого промежутка времени, который называется "временем двойного щелчка" (double-click time). Если вы хотите, чтобы ваша оконная процедура получала сообщения двойного щелчка мыши, то вы должны включить идентификатор CS_DBLCLKS при задании стиля окна в классе окна перед вызовом функции *RegisterClassEx*:

```
wndclass.style = CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS;
```

Если вы не включите CS_DBLCLKS в стиль окна, и пользователь дважды в быстром темпе щелкнет левой кнопкой мыши, то ваша оконная процедура получит следующие сообщения: WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDOWN и WM_LBUTTONUP. (Оконная процедура вполне может между этими сообщениями от кнопок мыши получать и другие сообщения.) Если вы хотите реализовать собственную логику обработки двойного щелчка мыши, то для получения относительного времени сообщений WM_LBUTTONDOWN, вы можете использовать функцию Windows *GetMessageTime*. Более подробно об этой функции рассказывается в главе 7.

Если вы включаете в свой класс окна идентификатор CS_DBLCLKS, то оконная процедура при двойном щелчке мыши получает следующие сообщения: WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDBLCLK и WM_LBUTTONUP. Сообщение WM_LBUTTONDBLCLK просто заменяет второе сообщение WM_LBUTTONDOWN.

Двойной щелчок мыши гораздо легче обрабатывать, если первый щелчок выполняет в оконной процедуре те же самые действия, которые выполняет простой щелчок. Затем второй щелчок (сообщение WM_LBUTTONBLCLK) выполняет какие-то дополнительные, относительно первого щелчка, действия. Например, посмотрите, как работает мышь со списком файлов в программе Windows Explorer. С помощью одного щелчка выбирается файл. Программа Windows Explorer выделяет выбранный файл, инвертируя цвет строки. Двойной щелчок выполняет два действия: при первом щелчке выбирается файл, точно также, как и при единственном щелчке; а второй щелчок побуждает программу Windows Explorer запустить файл. Логика здесь элементарна. Если бы первый щелчок двойного щелчка мыши не выполнял те же действия, которые выполняет единственный щелчок, то логика управления мышью могла бы быть гораздо сложнее.

Сообщения мыши нерабочей области

Те десять сообщений мыши, которые мы только что обсудили, относятся к ситуации, когда действия с мышью, т. е. перемещения мыши и щелчки ее кнопками, происходят в рабочей области окна. Если мышь оказывается вне рабочей области окна, но все еще внутри окна, Windows посылает оконной процедуре сообщения мыши "нерабочей области". Нерабочая область включает в себя панель заголовка, меню и полосы прокрутки окна.

Обычно вам нет необходимости обрабатывать сообщения мыши нерабочей области. Вместо этого вы просто передаете их в *DefWindowProc*, чтобы Windows могла выполнить системные функции. В этом смысле сообщения мыши нерабочей области похожи на системные сообщения клавиатуры WM_SYSKEYDOWN, WM_SYSKEYUP и WM_SYSCHAR.

Сообщения мыши нерабочей области почти полностью такие же как и сообщения мыши рабочей области. В названия сообщений входят буквы "NC", что означает "нерабочая" (nonclient). Если мышь перемещается внутри нерабочей области окна, то оконная процедура получает сообщение WM_NCMOUSEMOVE. Кнопки мыши вырабатывают следующие сообщения:

Кнопка	Нажатие	Отпускание	Нажатие (Второй щелчок)
Левая	WM_NCLBUTTONDOWN	WM_NCLBUTTONUP	WM_NCLBUTTONDBLCLK
Средняя	WM_NCMBUTTONDOWN	WM_NCMBUTTONUP	WM_NCMBUTTONDBLCLK

Кнопка	Нажатие	Отпускание	Нажатие (Второй щелчок)
Правая	WM_NCRBUTTONDOWN	WM_NCRBUTTONUP	WM_NCRBUTTONDBLCLK

Однако, параметры *wParam* и *lParam* для сообщений мыши нерабочей области отличаются от соответствующих параметров для сообщений мыши рабочей области. Параметр *wParam* показывает зону нерабочей области, в которой произошло перемещение или щелчок мыши. Его значение приравнивается одному из идентификаторов, начинающихся с HT (что означает "тест попадания" (hit-test), которые определяются в заголовочных файлах Windows.

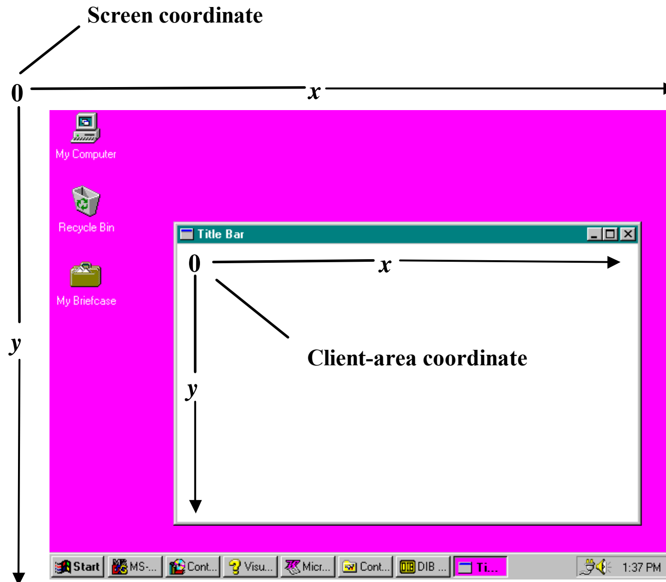


Рис. 6.3 Координаты экрана и координаты рабочей области

Переменная *lParam* содержит в младшем слове значение координаты *x*, а в старшем — *y*. Однако, эти координаты являются координатами экрана, а не координатами рабочей области, как это было у сообщений мыши рабочей области. Значения координат *x* и *y* верхнего левого угла экрана равны 0. Если вы движетесь вправо, то увеличивается значение координаты *x*, если вниз, то значение координаты *y*. (См. рис. 6.3.)

Вы можете преобразовать экранные координаты в координаты рабочей области окна и наоборот с помощью двух функций Windows:

```
ScreenToClient(hwnd, pPoint);
ClientToScreen(hwnd, pPoint);
```

Параметр *pPoint* — это указатель на структуру тира POINT. Две эти функции преобразуют хранящиеся в структуре данные без сохранения их прежних значений. Отметьте, что если точка с экранными координатами находится выше рабочей области окна, то преобразованное значение координаты *y* рабочей области будет отрицательным. И аналогично, значение *x* экранной координаты левее рабочей области после преобразования в координату рабочей области станет отрицательным.

Сообщение теста попадания

Если вы вели подсчет, то знаете, что мы рассмотрели 20 из 21 сообщения мыши. Последним сообщением является WM_NCHITTEST, означающее "тест попадания в нерабочую область" (nonclient hit-test). Это сообщение предшествует всем остальным сообщениям мыши рабочей и нерабочей области. Параметр *lParam* содержит значения *x* и *y* экранных координат положения мыши. Параметр *wParam* не используется.

В приложениях для Windows это сообщение обычно передается в *DefWindowProc*. В этом случае Windows использует сообщение WM_NCHITTEST для выработки всех остальных сообщений мыши на основе положения мыши. Для сообщений мыши нерабочей области возвращаемое значение функции *DefWindowProc* при обработке сообщения WM_NCHITTEST передается как параметр *wParam* в сообщении мыши. Это значение может быть любым из множества значений *wParam*, которое бывает у этого параметра для сообщений мыши нерабочей области, плюс следующие:

HTCLIENT	Рабочая область
HTNOWHERE	Нет ни на одном из окон
HTTRANSPARENT	Окно перекрыто другим окном

HTERROR	Заставляет *DefWindowProc* генерировать гудок

Если функция *DefWindowProc* после обработки сообщения WM_NCHITTEST возвращает значение HTCLIENT, то Windows преобразует экранные координаты в координаты рабочей области вырабатывает сообщение мыши рабочей области.

Если вы вспомните, как мы запретили все системные функции клавиатуры при обработке сообщения WM_SYSKEYDOWN, то вы, наверное, удивитесь, если сможете сделать что-нибудь подобное, используя сообщения мыши. Действительно, если вы вставите строки:

```
case WM_NCHITTEST:
       return(LRESULT) HTNOWHERE;
```

в вашу оконную процедуру, то вы полностью запретите все сообщения мыши рабочей и нерабочей области вашего окна. Кнопки мыши просто не будут работать до тех пор, пока мышь будет находится где-либо внутри вашего окна, включая значок системного меню, кнопки минимизации, максимизации и закрытия окна.

Сообщения порождают сообщения

Windows использует сообщение WM_NCHITTEST для выработки всех остальных сообщений мыши. Идея сообщений, порождающих другие сообщения, характерна для Windows. Давайте рассмотрим пример. Если вы дважды щелкните мышью на значке системного меню Windows-программы, то программа завершится. Двойной щелчок генерирует серию сообщений WM_NCHITTEST. Поскольку мышь установлена над значком системного меню, то возвращаемым значением функции *DefWindowProc* является HTSYSMENU, и Windows ставит в очередь сообщение WM_NCLBUTTONDBLCLK с параметром *wParam*, равным HTSYSMENU.

Оконная процедура обычно передает это сообщение *DefWindowProc*. Когда функция *DefWindowProc* получает сообщение WM_NCLBUTTONDBLCLK с параметром *wParam*, равным HTSYSMENU, то она ставит в очередь сообщение WM_SYSCOMMAND с параметром *wParam*, равным SC_CLOSE. (Это сообщение WM_SYSCOMMAND также генерируется, когда пользователь выбирает в системном меню пункт Close.) Оконная процедура вновь передает это сообщение в *DefWindowProc*. *DefWindowProc* обрабатывает сообщение, отправляя оконной процедуре синхронное сообщение WM_CLOSE.

Если программе для своего завершения требуется получить от пользователя подтверждение, оконная процедура может обработать сообщение WM_CLOSE. В противном случае оконная процедура обрабатывает сообщение WM_CLOSE, вызывая функцию *DestroyWindow*. Помимо других действий, функция *DestroyWindow* посылает оконной процедуре синхронное сообщение WM_DESTROY. Обычно оконная процедура обрабатывает сообщение WM_DESTROY следующим образом:

```
case WM_DESTROY:
       PostQuitMessage(0);
       return 0;
```

Функция *PostQuitMessage* заставляет Windows поместить в очередь сообщений сообщение WM_QUIT. До оконной процедуры это сообщение никогда не доходит, поскольку оно является причиной того, что функция *GetMessage* возвращает 0, что вызывает завершение цикла обработки сообщений и программы в целом.

Тестирование попадания в ваших программах

Ранее говорилось о том, как программа Windows Explorer реагирует на одиночные и двойные щелчки мыши. Очевидно, что программа должна определить, на какой именно файл пользователь указывает с помощью мыши. Это называется "тестом попадания". Так же как функция *DefWindowProc* должна осуществлять тестирование попадания при обработке сообщений WM_NCHITTEST, также очень часто и оконная процедура должна выполнять тестирование попадания внутри рабочей области. Как правило тест попадания включает в себя расчеты с использованием координат *x* и *y*, переданных вашей оконной процедуре в параметре *lParam* сообщения мыши.

Гипотетический пример

Рассмотрим пример. Ваша программа выводит на экран несколько столбцов файлов, отсортированных в алфавитном порядке. Список файлов начинается вверху рабочей области, имеющей ширину *cxClient* пикселей и высоту *cyClient* пикселей; высота каждого символа равна *cyChar* пикселей. Имена файлов хранятся в отсортированном массиве указателей на символьные строки, который называется *szFileNames*.

Давайте предположим, что ширина столбцов равна *cxColWidth* пикселей. Число файлов, которые вы сможете разместить в каждом столбце равно:

```
iNumInCol = cyClient / cyChar;
```

Вы получаете сообщение о щелчке мыши с координатами *cxMouse* и *cyMouse*, извлеченными из параметра *lParam*. На какой столбец имен файлов пользователь указывает с помощью мыши, можно определить по формуле:

```
iColumn = cxMouse / cxColWidth;
```

Положение имени файла относительно вершины столбца равно:

```
iFromTop = cyMouse / cyChar;
```

Теперь можно рассчитать индекс массива *szFileNames*:

```
iIndex = iColumn * iNumInCol + iFromTop;
```

Очевидно, что если значение *iIndex* превосходит число файлов, содержащихся в массиве, пользователь производит щелчок на пустой области экрана.

Во многих случаях тест попадания более сложен, чем предложенный в этом примере. Он может быть очень запутанным, например, в программе текстового редактора WORDPAD, в котором используются шрифты различных размеров. Когда вы что-нибудь выводите в рабочую область окна, то должны определить координаты каждого выводимого на экран объекта. В расчетах при тестировании попадания вы должны двигаться в обратном направлении: от координат к объекту. Однако, если объектом, который вы выводите на экран, является строка, такое движение в обратном направлении включает в себя и определение положения символов внутри строки.

Пример программы

Программа CHECKER1, приведенная на рис. 6.4, демонстрирует несколько простых тестов попадания. Программа делит рабочую область на 25 прямоугольников, получая таким образом массив размером 5 на 5. Если вы щелкаете мышью на одном из прямоугольников, то в прямоугольнике рисуется символ X. При повторном щелчке символ X удаляется.

CHECKER1.MAK

```
#------------------------
# CHECKER1.MAK make file
#------------------------

checker1.exe : checker1.obj
     $(LINKER) $(GUIFLAGS) -OUT:checker1.exe checker1.obj $(GUILIBS)

checker1.obj : checker1.c
     $(CC) $(CFLAGS) checker1.c
```

CHECKER1.C

```
/*-------------------------------------------------
   CHECKER1.C -- Mouse Hit-Test Demo Program No. 1
                 (c) Charles Petzold, 1996
  -------------------------------------------------*/

#include <windows.h>

#define DIVISIONS 5
#define MoveTo(hdc, x, y) MoveToEx(hdc, x, y, NULL)

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR  szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Checker1";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
```

```
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Checker1 Mouse Hit-Test Demo",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static BOOL fState[DIVISIONS][DIVISIONS];
     static int  cxBlock, cyBlock;
     HDC         hdc;
     PAINTSTRUCT ps;
     RECT        rect;
     int         x, y;

     switch(iMsg)
          {
          case WM_SIZE :
               cxBlock = LOWORD(lParam) / DIVISIONS;
               cyBlock = HIWORD(lParam) / DIVISIONS;
               return 0;

          case WM_LBUTTONDOWN :
               x = LOWORD(lParam) / cxBlock;
               y = HIWORD(lParam) / cyBlock;

               if(x < DIVISIONS && y < DIVISIONS)
                    {
                    fState [x][y] ^= 1;

                    rect.left   = x * cxBlock;
                    rect.top    = y * cyBlock;
                    rect.right  =(x + 1) * cxBlock;
                    rect.bottom =(y + 1) * cyBlock;

                    InvalidateRect(hwnd, &rect, FALSE);
                    }
               else
                    MessageBeep(0);
               return 0;

          case WM_PAINT :
               hdc = BeginPaint(hwnd, &ps);
```

```
        for(x = 0; x < DIVISIONS; x++)
             for(y = 0; y < DIVISIONS; y++)
                  {
                  Rectangle(hdc, x * cxBlock, y * cyBlock,
                           (x + 1) * cxBlock,(y + 1) * cyBlock);

                  if(fState [x][y])
                       {
                       MoveTo(hdc,  x    * cxBlock,  y    * cyBlock);
                       LineTo(hdc,(x+1) * cxBlock,(y+1) * cyBlock);
                       MoveTo(hdc,  x    * cxBlock,(y+1) * cyBlock);
                       LineTo(hdc,(x+1) * cxBlock,  y    * cyBlock);
                       }
                  }
        EndPaint(hwnd, &ps);
        return 0;

   case WM_DESTROY :
        PostQuitMessage(0);
        return 0;
   }
return DefWindowProc(hwnd, iMsg, wParam, lParam);
}
```

Рис. 6.4 Программа CHECKER1

На рис. 6.5 показан вывод программы CHECKER1. У всех 25 прямоугольников одинаковая высота и ширина. Значения ширины и высоты хранятся в *cxBlock* и *cyBlock*, и пересчитываются при изменении размеров рабочей области. В логике обработки сообщений WM_LBUTTONDOWN для определения прямоугольника, на котором был произведен щелчок, используются координаты мыши. Этот обработчик устанавливает текущее состояние прямоугольника в массиве *fState*, и делает соответствующий прямоугольник недействительным для выработки сообщения WM_PAINT. Если ширина или высота рабочей области не делится без остатка на пять, узкая полоса справа или внизу рабочей области не будет зарисована прямоугольниками. В ответ на щелчок мыши в этой области, программа CHECKER1, вызывая функцию *MessageBeep*, сообщит об ошибке.

Когда программа CHECKER1 получает сообщение WM_PAINT, она перерисовывает всю рабочую область, рисуя прямоугольники с помощью функции GDI *Rectangle*. Если установлено значение *fState*, то программа CHECKER1 с помощью функций *MoveTo* и *LineTo* рисует две линии. Перед перерисовкой, при обработке сообщения WM_PAINT, программа не проверяет, действительна ли каждая прямоугольная область, хотя могла бы это делать. Первый метод такой проверки заключается в создании внутри цикла структуры RECT для каждого прямоугольного блока (с использованием тех же формул, что и для сообщения WM_LBUTTONDOWN) и проверки, с помощью функции *IntersectRect*, пересекается ли он с недействительным прямоугольником (*ps.rcPaint*). Другим методом может быть использование функции *PtInRect* для определения того, находится ли любой из четырех углов прямоугольного блока внутри недействительного прямоугольника.

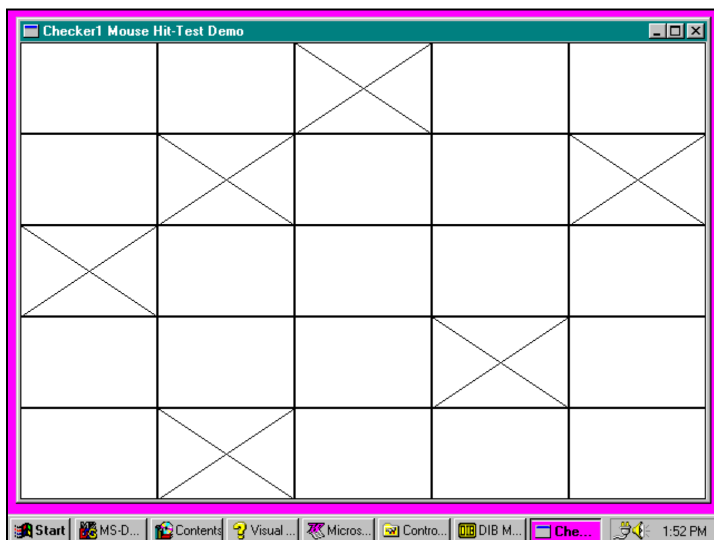


Рис. 6.5 Вывод на экран программы CHECKER1

Эмуляция мыши с помощью клавиатуры

Программа CHECKER1 работает только при наличии у вас мыши. Добавим к программе интерфейс клавиатуры примерно также, как мы это делали для программы SYSMETS в главе 5. Однако, добавление интерфейса клавиатуры к программе, использующей курсор мыши для выбора объекта, требует, чтобы мы также побеспокоились об отображении и перемещении курсора на экране.

Даже если мышь не установлена, Windows все же может вывести курсор мыши на экран. Windows поддерживает для курсора "счетчик отображения" (display count). Если мышь инсталлирована, то начальное значение счетчика отображения равно 0; если нет, то его начальное значение равно —1. Курсор мыши выводится на экран только в том случае, если значение счетчика отображения неотрицательно. Вы можете увеличить значение счетчика отображения на 1, вызывая функцию:

```
ShowCursor(TRUE);
```

и уменьшить его на 1 с помощью вызова:

```
ShowCursor(FALSE);
```

Перед использованием функции *ShowCursor* нет нужды определять, инсталлирована мышь или нет. Если вы хотите вывести на экран курсор мыши независимо от наличия собственно мыши, просто увеличьте на 1 счетчик отображения. После однократного увеличения счетчика отображения, его уменьшение скроет курсор только в том случае, если мышь не была инсталлирована, но при наличии мыши курсор останется на экране. Вывод на экран курсора мыши относится ко всей операционной системе Windows, поэтому вам следует быть уверенным, что вы увеличивали и уменьшали счетчик отображения равное число раз.

В вашей оконной процедуре можете использовать следующую простую логику:

```
case WM_SETFOCUS:
       ShowCursor(TRUE);
       return 0;
case WM_KILLFOCUS:
       ShowCursor(FALSE);
       return 0;
```

Оконная процедура получает сообщение WM_SETFOCUS, когда окно приобретает фокус ввода клавиатуры, и сообщение WM_KILLFOCUS, когда теряет фокус ввода. Эти события являются самыми подходящими для того, чтобы вывести на экран или убрать с экрана курсор мыши. Во-первых, сообщения WM_SETFOCUS и WM_KILLFOCUS являются взаимодополняющими — следовательно, оконная процедура равное число раз увеличит и уменьшит счетчик отображения курсора. Во-вторых, для версий Windows, в которых мышь не инсталлирована, использование сообщений WM_SETFOCUS и WM_KILLFOCUS вызовет появление курсора только в том окне, которое имеет фокус ввода. И только в этом случае, используя интерфейс клавиатуры, который вы создадите, пользователь сможет перемещать курсор.

Windows отслеживает текущее положение курсора мыши, даже если сама мышь не инсталлирована. Если мышь не установлена, и вы выводите курсор мыши на экран, то он может оказаться в любой части экрана и будет оставаться там до тех пор, пока вы не переместите его. Получить положение курсора можно с помощью функции:

```
GetCursorPos(pPoint);
```

где *pPoint* — это указатель на структуру POINT. Функция заполняет поля структуры POINT значениями координат *x* и *y* мыши. Установить положение курсора можно с помощью функции:

```
SetCursorPos(x, y);
```

В обоих этих случаях значения координат *x* и *y* являются координатами экрана, а не рабочей области. (Это очевидно, поскольку этим функциям не нужен параметр *hwnd*.) Как уже отмечалось, вы можете преобразовать экранные координаты в координаты рабочей области и наоборот с помощью функций *ScreenToClient* и *ClientToScreen*.

Если вы при обработке сообщения мыши вызываете функцию *GetCursorPos* и преобразуете экранные координаты в координаты рабочей области, то они могут слегка отличаться от координат, содержащихся в параметре *lParam* сообщения мыши. Координаты, которые возвращаются из функции *GetCursorPos*, показывают текущее положение мыши. Координаты в параметре *lParam* сообщения мыши — это координаты мыши в момент генерации сообщения.

Вы, вероятно, захотите так написать логику работы клавиатуры, чтобы двигать курсор мыши с помощью клавиш управления курсором клавиатуры и воспроизводить кнопки мыши с помощью клавиши пробела <Spacebar> или <Enter>. При этом не следует делать так, чтобы курсор мыши сдвигался на один пиксель за одно нажатие клавиши. В этом случае вам для того, чтобы переместить указатель мыши от одной стороны экрана до другой, надо было бы держать клавишу со стрелкой нажатой более минуты.

Если нужно реализовать интерфейс клавиатуры так, чтобы точность позиционирования курсора мыши по-прежнему оставалась равной одному пикселю, тогда обработка сообщения клавиатуры должна идти следующим образом: при нажатии клавиши со стрелкой курсор мыши начинает двигаться медленно, а затем, если клавиша остается нажатой, его скорость возрастает. Вспомните, что параметр *lParam* в сообщениях WM_KEYDOWN показывает, являются ли сообщения о нажатии клавиш результатом автоповтора. Превосходное применение этой информации!

Добавление интерфейса клавиатуры к программе CHECKER

Программа CHECKER2, представленная на рис. 6.6, является той же программой CHECKER1, за исключением того, что здесь добавлен интерфейс клавиатуры. Вы можете использовать клавиши со стрелками для перемещения курсора между 25 прямоугольниками. Клавиша <Home> перемещает курсор к верхнему левому углу прямоугольника; клавиша <End> опускает его в нижний правый прямоугольник. Клавиши и <Spacebar> и <Enter> включают и выключают вывод символа перечеркивания X.

CHECKER2.MAK

```
#------------------------
# CHECKER2.MAK make file
#------------------------

checker2.exe : checker2.obj
     $(LINKER) $(GUIFLAGS) -OUT:checker2.exe checker2.obj $(GUILIBS)

checker2.obj : checker2.c
     $(CC) $(CFLAGS) checker2.c
```

CHECKER2.C

```
/*-------------------------------------------------
   CHECKER2.C -- Mouse Hit-Test Demo Program No. 2
                 (c) Charles Petzold, 1996
  -------------------------------------------------*/

#include <windows.h>

#define DIVISIONS 5
#define MoveTo(hdc, x, y) MoveToEx(hdc, x, y, NULL)

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Checker2";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Checker2 Mouse Hit-Test Demo",
                          WS_OVERLAPPEDWINDOW,
```

```
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static BOOL fState[DIVISIONS][DIVISIONS];
     static int  cxBlock, cyBlock;
     HDC         hdc;
     PAINTSTRUCT ps;
     POINT       point;
     RECT        rect;
     int         x, y;

     switch(iMsg)
          {
          case WM_SIZE :
               cxBlock = LOWORD(lParam) / DIVISIONS;
               cyBlock = HIWORD(lParam) / DIVISIONS;
               return 0;

          case WM_SETFOCUS :
               ShowCursor(TRUE);
               return 0;

          case WM_KILLFOCUS :
               ShowCursor(FALSE);
               return 0;

          case WM_KEYDOWN :
               GetCursorPos(&point);
               ScreenToClient(hwnd, &point);

               x = max(0, min(DIVISIONS - 1, point.x / cxBlock));
               y = max(0, min(DIVISIONS - 1, point.y / cyBlock));

               switch(wParam)
                    {
                    case VK_UP :
                         y--;
                         break;

                    case VK_DOWN :
                         y++;
                         break;

                    case VK_LEFT :
                         x--;
                         break;

                    case VK_RIGHT :
                         x++;
                         break;
```

```
          case VK_HOME :
               x = y = 0;
               break;

          case VK_END :
               x = y = DIVISIONS - 1;
               break;

          case VK_RETURN :
          case VK_SPACE :
               SendMessage(hwnd, WM_LBUTTONDOWN, MK_LBUTTON,
                         MAKELONG(x * cxBlock, y * cyBlock));
               break;
          }
     x =(x + DIVISIONS) % DIVISIONS;
     y =(y + DIVISIONS) % DIVISIONS;

     point.x = x * cxBlock + cxBlock / 2;
     point.y = y * cyBlock + cyBlock / 2;

     ClientToScreen(hwnd, &point);
     SetCursorPos(point.x, point.y);
     return 0;
case WM_LBUTTONDOWN :
     x = LOWORD(lParam) / cxBlock;
     y = HIWORD(lParam) / cyBlock;

     if(x < DIVISIONS && y < DIVISIONS)
          {
          fState[x][y] ^= 1;

          rect.left   = x * cxBlock;
          rect.top    = y * cyBlock;
          rect.right  =(x + 1) * cxBlock;
          rect.bottom =(y + 1) * cyBlock;

          InvalidateRect(hwnd, &rect, FALSE);
          }
     else
          MessageBeep(0);
     return 0;

case WM_PAINT :
     hdc = BeginPaint(hwnd, &ps);

     for(x = 0; x < DIVISIONS; x++)
          for(y = 0; y < DIVISIONS; y++)
               {
               Rectangle(hdc, x * cxBlock, y * cyBlock,
                        (x + 1) * cxBlock,(y + 1) * cyBlock);

               if(fState [x][y])
                    {
                    MoveTo(hdc,  x    * cxBlock,  y    * cyBlock);
                    LineTo(hdc,(x+1) * cxBlock,(y+1) * cyBlock);
                    MoveTo(hdc,  x    * cxBlock,(y+1) * cyBlock);
                    LineTo(hdc,(x+1) * cxBlock,  y    * cyBlock);
                    }
               }
     EndPaint(hwnd, &ps);
     return 0;

case WM_DESTROY :
```

```
            PostQuitMessage(0);
            return 0;
        }
    return DefWindowProc(hwnd, iMsg, wParam, lParam);
    }
```

Рис. 6.6 Программа CHECKER2

Логика обработки сообщения WM_KEYDOWN в программе CHECKER2 следующая: определяется положение курсора (*GetCursorPos*), координаты экрана преобразуются в координаты рабочей области (*ScreenToClient*), а затем координаты делятся на ширину и высоту прямоугольного блока. Полученные значения *x* и *y* показывают положение прямоугольника в массиве 5 на 5. Курсор мыши при нажатии клавиши не всегда находится в рабочей области, поэтому *x* и *y* должны быть обработаны макросами *min* и *max* таким образом, чтобы гарантировать их попадание в диапазон от 0 до 4.

Для клавиш со стрелками программа CHECKER2 увеличивает или уменьшает на 1 соответственно значение *x* или *y*. Если нажатой клавишей является клавиша <Enter> (VK_RETURN) или клавиша <Spacebar> (VK_SPACE), то программа CHECKER2 использует функцию *SendMessage* для посылки себе же синхронного сообщения WM_LBUTTONDOWN. Такая технология напоминает метод, использованный в программе SYSMETS в главе 5, где для полосы прокрутки окна в программу добавлен интерфейс клавиатуры. Логика обработки сообщения WM_KEYDOWN заканчивается расчетом координат рабочей области, которые являются координатами центра прямоугольника, преобразованием их в координаты экрана (*ClientToScreen*) и установкой положения курсора (*SetCursorPos*).

Использование дочерних окон для тестирования попадания

В некоторых программах, например в программе Windows PAINT, рабочая область окна делится на более мелкие логические области. Программа PAINT, окно которой представлено на рис. 6.7, имеет слева область меню со значками, а внизу область меню выбора цветов.

Рис. 6.7 Программа Windows PAINT

Программа PAINT при тестировании попадания в эти две области, перед определением выбранного пользователем пункта меню, должна учитывать положение меню внутри рабочей области.

А можно иначе. В действительности, в программе PAINT рисование меню и тестирование попадания упрощается, благодаря использованию "дочерних окон" (child windows). Дочерние окна делят рабочую область на несколько более мелких участков. Каждое дочернее окно имеет собственный описатель окна, оконную процедуру и рабочую область. Каждая оконная процедура получает сообщения мыши, которые относятся только к его дочернему окну. Параметр *lParam* в сообщении мыши содержит координаты, заданные относительно верхнего левого угла рабочей области дочернего, а не родительского окна.

Дочерние окна, используемые таким образом, помогают сделать ваши программы более структурированными и модульными. Если дочерние окна используют разные классы окна, то каждое дочернее окно может иметь собственную оконную процедуру. Для разных классов окна могут также определяться и разный цветовой фон и разные, задаваемые по умолчанию, курсоры. В главе 8, мы рассмотрим "дочерние окна управления" (child window

controls) — предопределенные дочерние окна, имеющие форму полос прокрутки, кнопок и окон редактирования. А сейчас давайте рассмотрим, как можно использовать дочерние окна в программе CHECKER.

Дочерние окна в програме CHECKER

На рис. 6.8 представлена программа CHECKER3. В этой версии программы для обработки щелчков мыши создаются 25 дочерних окон. Здесь отсутствует интерфейс клавиатуры, но он может быть легко добавлен.

В программе CHECKER3 имеется две оконные процедуры, которые называются *WndProc* и *ChildWndProc*. *WndProc* — это по-прежнему оконная процедура главного (или родительского) окна. *ChildWndProc* — это оконная процедура 25 дочерних окон. Обе оконные процедуры должны быть определены как функции обратного вызова (CALLBACK).

Поскольку оконная процедура задается в структуре класса окна, которую вы регистрируете в Windows с помощью функции *RegisterClassEx*, то для двух оконных процедур в программе CHECKER3.C требуется два класса окна. Первый класс окна — это класс главного окна, и называется он "Checker3". Второму классу окна назначено имя "Checker3_Child".

Большинство полей структурной переменной *wndclass* просто повторно используются при регистрации класса "Checker3_Child" в *WinMain*. Поле *lpszClassName* устанавливается в "Checker3_Child" (имя класса). Поле *lpfnWndProc* устанавливается в *ChildWndProc* — оконную процедуру этого класса, а поля *hIcon* и *hIconSm* устанавливаются в NULL, поскольку с дочерними окнами значки не используются. Для класса окна "Checker3_Child" поле *cbWndExtra* структурной переменной *wndclass* устанавливается равной 2 байтам, или точнее, *sizeof(WORD)*. Это поле требует от Windows зарезервировать 2 байта дополнительного пространства в структуре, которую Windows строит для каждого окна, создаваемого на основе данного класса окна. Вы можете использовать это пространство для хранения информации, которая может быть индивидуальной для каждого окна.

CHECKER3.MAK

```
#------------------------
# CHECKER3.MAK make file
#------------------------

checker3.exe : checker3.obj
     $(LINKER) $(GUIFLAGS) -OUT:checker3.exe checker3.obj $(GUILIBS)

checker3.obj : checker3.c
     $(CC) $(CFLAGS) checker3.c
```

CHECKER3.C

```
/*-------------------------------------------------
   CHECKER3.C -- Mouse Hit-Test Demo Program No. 3
                 (c) Charles Petzold, 1996
  -------------------------------------------------*/

#include <windows.h>

#define DIVISIONS 5
#define MoveTo(hdc, x, y) MoveToEx(hdc, x, y, NULL)

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

char      szChildClass[] = "Checker3_Child";

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Checker3";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
```

```
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     wndclass.lpfnWndProc   = ChildWndProc;
     wndclass.cbWndExtra    = sizeof(WORD);
     wndclass.hIcon         = NULL;
     wndclass.lpszClassName = szChildClass;
     wndclass.hIconSm       = NULL;

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Checker3 Mouse Hit-Test Demo",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HWND hwndChild[DIVISIONS][DIVISIONS];
     int         cxBlock, cyBlock, x, y;

     switch(iMsg)
          {
          case WM_CREATE :
               for(x = 0; x < DIVISIONS; x++)
                    for(y = 0; y < DIVISIONS; y++)
                         {
                         hwndChild[x][y] = CreateWindow(szChildClass, NULL,
                              WS_CHILDWINDOW | WS_VISIBLE,
                              0, 0, 0, 0,
                              hwnd,(HMENU)(y << 8 | x),
                             (HINSTANCE) GetWindowLong(hwnd, GWL_HINSTANCE),
                              NULL);
                         }
               return 0;

          case WM_SIZE :
               cxBlock = LOWORD(lParam) / DIVISIONS;
               cyBlock = HIWORD(lParam) / DIVISIONS;

               for(x = 0; x < DIVISIONS; x++)
                    for(y = 0; y < DIVISIONS; y++)
                         MoveWindow(hwndChild[x][y],
```

```
                              x * cxBlock, y * cyBlock,
                              cxBlock, cyBlock, TRUE);
               return 0;

          case WM_LBUTTONDOWN :
               MessageBeep(0);
               return 0;

          case WM_DESTROY :
               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }

LRESULT CALLBACK ChildWndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     HDC         hdc;
     PAINTSTRUCT ps;
     RECT        rect;

     switch(iMsg)
          {
          case WM_CREATE :
               SetWindowWord(hwnd, 0, 0);       // on/off flag
               return 0;

          case WM_LBUTTONDOWN :
               SetWindowWord(hwnd, 0, 1 ^ GetWindowWord(hwnd, 0));
               InvalidateRect(hwnd, NULL, FALSE);
               return 0;

          case WM_PAINT :
               hdc = BeginPaint(hwnd, &ps);

               GetClientRect(hwnd, &rect);
               Rectangle(hdc, 0, 0, rect.right, rect.bottom);

               if(GetWindowWord(hwnd, 0))
                    {
                    MoveTo(hdc, 0,          0);
                    LineTo(hdc, rect.right, rect.bottom);
                    MoveTo(hdc, 0,          rect.bottom);
                    LineTo(hdc, rect.right, 0);
                    }

               EndPaint(hwnd, &ps);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

Рис. 6.8 Программа CHECKER3

Вызов функции *CreateWindow* в *WinMain* создает на основе класса "Checker3" главное окно. Это нормально. Однако, когда *WndProc* получает сообщение WM_CREATE, она, чтобы создать 25 дочерних окон на основе класса "Checker3_Child", 25 раз вызывает функцию *CreateWindow*. В следующей таблице дано сравнение параметров вызова функции *CreateWindow* в *WinMain*, создающей главное окно и вызова функции *CreateWindow* в *WndProc*, создающей 25 дочерних окон.

Параметр	Главное окно	Дочернее окно
класс окна	"Checker3"	"Checker3_Child"
заголовок окна	"Checker3... "	NULL
стиль окна	WS_OVERLAPPEDWINDOW	WS_CHILDWINDOW| WS_VISIBLE

Параметр	Главное окно	Дочернее окно
положение по горизонтали	CW_USEDEFAULT	0
положение по вертикали	CW_USEDEFAULT	0
ширина	CW_USEDEFAULT	0
высота	CW_USEDEFAULT	0
описатель родительского окна	NULL	*hwnd*
описатель меню/ идентификатор дочернего окна	NULL	(HMENU)(y << 8 | x)
описатель экземпляра	*hInstance*	(HINSTANCE) *GetWindowLong* (*hwnd*, GWL_HINSTANCE)
дополнительные параметры	NULL	NULL

Обычно для дочерних окон требуются параметры положения, ширины и высоты, но в программе CHECKER3 положение и размер дочерних окон устанавливаются позже в *WndProc*. Описатель родительского окна для главного окна равен NULL, поскольку оно родительское. Описатель родительского окна необходим при вызове функции *CreateWindow* для создания дочернего окна.

В главном окне отсутствует меню, поэтому соответствующий параметр равен NULL. Для дочерних окон этот же параметр называется "идентификатором дочернего окна" (child ID). Это число уникально для каждого дочернего окна. Идентификатор дочернего окна приобретает важное значение при управлении дочерними окнами, поскольку сообщения для родительского окна, как мы увидим в главе 8, идентифицируются этим идентификатором дочернего окна. В программе CHECKER3 идентификаторы дочерних окон установлены так, чтобы они соответствовали положению, которое каждое дочернее окно занимает в массиве 5 на 5 внутри главного окна.

Описателем экземпляра в обоих классах является *hInstance*. Когда создается дочернее окно, значение *hInstance* извлекается из структуры, которую Windows поддерживает для окна, с помощью функции *GetWindowLong*. (Вместо функции *GetWindowLong* можно было бы сохранить значение *hInstance* в глобальной переменной и непосредственно его использовать.)

Каждое дочернее окно имеет свой описатель окна, который хранится в массиве *hwndChild*. Когда *WndProc* получает сообщение WM_SIZE, она вызывает функцию *MoveWindow* для каждого из 25 дочерних окон. Параметры задают верхний левый угол дочернего окна относительно начала координат рабочей области родительского окна, ширину и высоту дочернего окна, а также необходимость перерисовки дочернего окна.

Теперь давайте рассмотрим *ChildWndProc*. Эта оконная процедура обрабатывает сообщения для всех 25 дочерних окон. Параметр *hwnd* для *ChildWndProc* является описателем дочернего окна, получающего сообщение. Когда *ChildWndProc* обрабатывает сообщение WM_CREATE (что происходит 25 раз, поскольку имеется 25 дочерних окон), она использует функцию *SetWindowWord* для хранения 0 в дополнительном пространстве, зарезервированном внутри структуры окна. (Вспомните, что мы зарезервировали это пространство с помощью поля *cbWndExtra* при определении структуры класса окна.) *ChildWndProc* использует это значение для хранения текущего состояния прямоугольника (зачеркнут он символом X или нет). Когда в дочернем окне происходит щелчок мыши, логика обработки сообщения WM_LBUTTONDOWN просто изменяет значение этого слова (0 на 1 или 1 на 0) и делает недействительной всю рабочую область дочернего окна. Эта область и является тем самым прямоугольником, в котором произошел щелчок. Обработка сообщения WM_PAINT вполне обычна, поскольку размер рисуемого прямоугольника равен размеру рабочей области окна.

Поскольку файл с текстом исходной программы на С и исполняемый файл .EXE программы CHECKER3 больше, чем аналогичные файлы для программы CHECKER1, то не будем утверждать, что CHECKER3 "проще", чем CHECKER1. Но обратите внимание, что нам больше не нужно делать никакого тестирования попадания для мыши! Если какое-нибудь дочернее окно в программе CHECKER3 получает сообщение WM_LBUTTONDOWN, то значит в этом окне и было нажатие, и это все, что ему нужно знать.

Если вы хотите добавить к программе CHECKER3 интерфейс клавиатуры, запомните, что главное окно по-прежнему получает сообщения клавиатуры, поскольку оно имеет фокус ввода. Более подробно дочерние окна мы рассмотрим в главе 8.

Захват мыши

Оконная процедура обычно получает сообщения мыши только тогда, когда курсор мыши находится в рабочей или в нерабочей области окна. Но программе может понадобиться получать сообщения мыши и тогда, когда курсор мыши находится вне окна. Если это так, то программа может произвести "захват" (capture) мыши.

Рисование прямоугольника

Для того, чтобы понять, для чего может понадобиться захват мыши, давайте рассмотрим программу BLOKOUT1, представленную на рис. 6.9.

BLOKOUT1.MAK

```
#------------------------
# BLOKOUT1.MAK make file
#------------------------

blokout1.exe : blokout1.obj
     $(LINKER) $(GUIFLAGS) -OUT:blokout1.exe blokout1.obj $(GUILIBS)

blokout1.obj : blokout1.c
     $(CC) $(CFLAGS) blokout1.c
```

BLOKOUT1.C

```
/*-----------------------------------------
   BLOKOUT1.C -- Mouse Button Demo Program
                (c) Charles Petzold, 1996
  -----------------------------------------*/

#include <windows.h>

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "BlokOut1";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;
     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Mouse Button Demo",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

```
void DrawBoxOutline(HWND hwnd, POINT ptBeg, POINT ptEnd)
     {
     HDC hdc;

     hdc = GetDC(hwnd);

     SetROP2(hdc, R2_NOT);
     SelectObject(hdc, GetStockObject(NULL_BRUSH));
     Rectangle(hdc, ptBeg.x, ptBeg.y, ptEnd.x, ptEnd.y);

     ReleaseDC(hwnd, hdc);
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static BOOL  fBlocking, fValidBox;
     static POINT ptBeg, ptEnd, ptBoxBeg, ptBoxEnd;
     HDC          hdc;
     PAINTSTRUCT  ps;
     switch(iMsg)
          {
          case WM_LBUTTONDOWN :
               ptBeg.x = ptEnd.x = LOWORD(lParam);
               ptBeg.y = ptEnd.y = HIWORD(lParam);

               DrawBoxOutline(hwnd, ptBeg, ptEnd);

               SetCursor(LoadCursor(NULL, IDC_CROSS));

               fBlocking = TRUE;
               return 0;

          case WM_MOUSEMOVE :
               if(fBlocking)
                    {
                    SetCursor(LoadCursor(NULL, IDC_CROSS));

                    DrawBoxOutline(hwnd, ptBeg, ptEnd);

                    ptEnd.x = LOWORD(lParam);
                    ptEnd.y = HIWORD(lParam);

                    DrawBoxOutline(hwnd, ptBeg, ptEnd);
                    }
               return 0;

          case WM_LBUTTONUP :
               if(fBlocking)
                    {
                    DrawBoxOutline(hwnd, ptBeg, ptEnd);

                    ptBoxBeg   = ptBeg;
                    ptBoxEnd.x = LOWORD(lParam);
                    ptBoxEnd.y = HIWORD(lParam);

                    SetCursor(LoadCursor(NULL, IDC_ARROW));

                    fBlocking = FALSE;
                    fValidBox  = TRUE;

                    InvalidateRect(hwnd, NULL, TRUE);
                    }
               return 0;
```

```
        case WM_CHAR :
             if(fBlocking & wParam == '\x1B')       // ie, Escape
                  {
                  DrawBoxOutline(hwnd, ptBeg, ptEnd);

                  SetCursor(LoadCursor(NULL, IDC_ARROW));
                  fBlocking = FALSE;
                  }
             return 0;

        case WM_PAINT :
             hdc = BeginPaint(hwnd, &ps);

             if(fValidBox)
                  {
                  SelectObject(hdc, GetStockObject(BLACK_BRUSH));
                  Rectangle(hdc, ptBoxBeg.x, ptBoxBeg.y,
                                 ptBoxEnd.x, ptBoxEnd.y);
                  }

             if(fBlocking)
                  {
                  SetROP2(hdc, R2_NOT);
                  SelectObject(hdc, GetStockObject(NULL_BRUSH));
                  Rectangle(hdc, ptBeg.x, ptBeg.y, ptEnd.x, ptEnd.y);
                  }

             EndPaint(hwnd, &ps);
             return 0;

        case WM_DESTROY :
             PostQuitMessage(0);
             return 0;
        }
   return DefWindowProc(hwnd, iMsg, wParam, lParam);
   }
```

Рис. 6.9 Программа BLOKOUT1

Эта программа показывает немногое из того, что могло бы быть реализовано в программе рисования в Windows. Вы начинаете рисовать, нажимая левую кнопку мыши, чтобы указать один из углов прямоугольника, и удерживая кнопку нажатой, вы двигаете мышь. Программа рисует контур прямоугольника, со вторым противоположным углом в текущей позиции курсора. После отпускания кнопки мыши, программа закрашивает прямоугольник. На рис. 6.10 показаны два прямоугольника, один из которых уже нарисован, а второй находится в процессе рисования.

При нажатии левой кнопки мыши, программа BLOKOUT1 сохраняет координаты мыши и первый раз вызывает функцию *DrawBoxOutline*. Функция рисует прямоугольник с использованием растровой операции R2_NOT, которая меняет цвет рабочей области на противоположный. При обработке последующих сообщений WM_MOUSEMOVE программа снова рисует такой же прямоугольник, полностью стирая предыдущий. Затем она использует новые координаты мыши для рисования нового прямоугольника. Наконец, когда программа BLOKOUT1 получает сообщение WM_LBUTTONUP, координаты мыши сохраняются, и окно делается недействительным, генерируя сообщение WM_PAINT для вывода на экран полученного прямоугольника.

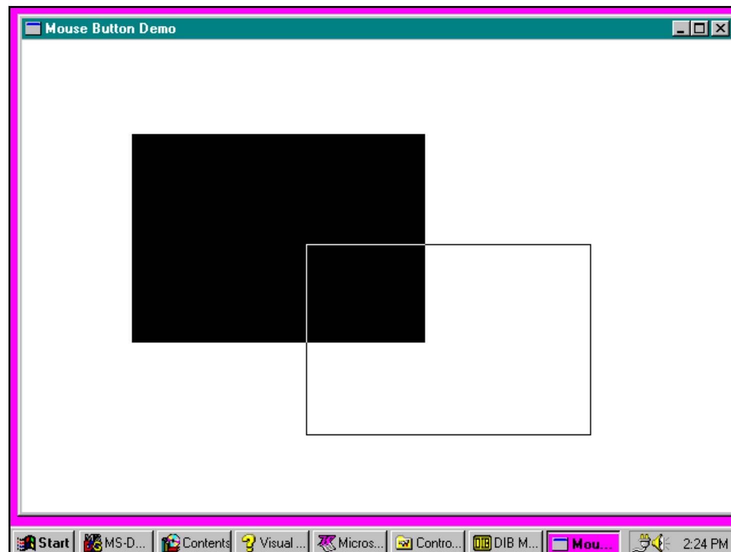


Рис. 6.10 Вид окна программы BLOKOUT1

В чем же проблема?

Попытайтесь сделать следующее: нажмите левую кнопку мыши внутри рабочей области окна программы BLOKOUT1, а затем переместите курсор за пределы окна. Программа перестает получать сообщения WM_MOUSEMOVE. Теперь отпустите кнопку. Программа не получит сообщение WM_LBUTTONUP, поскольку курсор находится вне рабочей области. Верните курсор внутрь рабочей области окна программы BLOKOUT1. Оконная процедура по-прежнему считает, что кнопка остается нажатой.

Это нехорошо. Программа не знает что происходит.

Решение проблемы — захват

Программа BLOKOUT1 показала свои некоторые общие возможности, но она имеет и очевидные недостатки. Для решения проблем такого типа был введен механизм захвата мыши. Если пользователь двигает мышь, то выход мыши за границы окна не должен создавать трудностей. Программа должна по-прежнему контролировать мышь.

Захватить мышь проще, чем поймать ее в мышеловку. Вам достаточно только вызвать функцию:

```
SetCapture(hwnd);
```

После вызова этой функции, Windows посылает все сообщения мыши в оконную процедуру того окна, описателем которого является *hwnd*. Сообщения мыши всегда остаются сообщениями рабочей области, даже если мышь оказывается в нерабочей области окна. Параметр *lParam* по-прежнему показывает положение мыши в координатах рабочей области. Эти координаты, однако, могут стать отрицательными, если мышь окажется левее или выше рабочей области.

Пока мышь захвачена, системные функции клавиатуры тоже не действуют. Когда вы захотите освободить мышь, вызовите функцию:

```
ReleaseCapture();
```

Эта функция возвращает обработку мыши в нормальный режим.

При работе под Windows 95 захват мыши несколько более ограничен, чем это было в предыдущих версиях Windows. Точнее, если мышь была захвачена, а кнопка мыши в данный момент не нажата и курсор мыши перемещается в другое окно, то вместо захватившего мышь окна, сообщения мыши получит окно, на котором находится курсор, а не окно, захватившее мышь. Это сделано для предотвращения в системе ситуации, когда одна из программ захватит и не освободит мышь.

Другими словами, захватывайте мышь только в том случае, если кнопка нажимается в вашей рабочей области. Освобождайте мышь, когда кнопка отпускается.

Программа BLOKOUT2

На рис. 6.11 представлена программа BLOKOUT2, иллюстрирующая зах-ват мыши.

BLOKOUT2.MAK

```
#-----------------------
# BLOKOUT2.MAK make file
#-----------------------

blokout2.exe : blokout2.obj
     $(LINKER) $(GUIFLAGS) -OUT:blokout2.exe blokout2.obj $(GUILIBS)

blokout2.obj : blokout2.c
     $(CC) $(CFLAGS) blokout2.c
```

BLOKOUT2.C

```
/*---------------------------------------------------
   BLOKOUT2.C -- Mouse Button & Capture Demo Program
                 (c) Charles Petzold, 1996
  ---------------------------------------------------*/
#include <windows.h>

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "BlokOut2";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Mouse Button & Capture Demo",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

void DrawBoxOutline(HWND hwnd, POINT ptBeg, POINT ptEnd)
     {
     HDC hdc;

     hdc = GetDC(hwnd);
```

```
     SetROP2(hdc, R2_NOT);
     SelectObject(hdc, GetStockObject(NULL_BRUSH));
     Rectangle(hdc, ptBeg.x, ptBeg.y, ptEnd.x, ptEnd.y);

     ReleaseDC(hwnd, hdc);
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static BOOL  fBlocking, fValidBox;
     static POINT ptBeg, ptEnd, ptBoxBeg, ptBoxEnd;
     HDC          hdc;
     PAINTSTRUCT  ps;

     switch(iMsg)
          {
          case WM_LBUTTONDOWN :
               ptBeg.x = ptEnd.x = LOWORD(lParam);
               ptBeg.y = ptEnd.y = HIWORD(lParam);

               DrawBoxOutline(hwnd, ptBeg, ptEnd);

               SetCapture(hwnd);
               SetCursor(LoadCursor(NULL, IDC_CROSS));

               fBlocking = TRUE;
               return 0;

          case WM_MOUSEMOVE :
               if(fBlocking)
                    {
                    SetCursor(LoadCursor(NULL, IDC_CROSS));

                    DrawBoxOutline(hwnd, ptBeg, ptEnd);

                    ptEnd.x = LOWORD(lParam);
                    ptEnd.y = HIWORD(lParam);

                    DrawBoxOutline(hwnd, ptBeg, ptEnd);
                    }
               return 0;

          case WM_LBUTTONUP :
               if(fBlocking)
                    {
                    DrawBoxOutline(hwnd, ptBeg, ptEnd);

                    ptBoxBeg   = ptBeg;
                    ptBoxEnd.x = LOWORD(lParam);
                    ptBoxEnd.y = HIWORD(lParam);

                    ReleaseCapture();
                    SetCursor(LoadCursor(NULL, IDC_ARROW));

                    fBlocking = FALSE;
                    fValidBox  = TRUE;

                    InvalidateRect(hwnd, NULL, TRUE);
                    }
               return 0;

          case WM_CHAR :
               if(fBlocking & wParam == '\x1B')       // i.e., Escape
                    {
```

```
                DrawBoxOutline(hwnd, ptBeg, ptEnd);
                ReleaseCapture();
                SetCursor(LoadCursor(NULL, IDC_ARROW));

                fBlocking = FALSE;
                }
           return 0;

      case WM_PAINT :
           hdc = BeginPaint(hwnd, &ps);

           if(fValidBox)
                {
                SelectObject(hdc, GetStockObject(BLACK_BRUSH));
                Rectangle(hdc, ptBoxBeg.x, ptBoxBeg.y,
                                ptBoxEnd.x, ptBoxEnd.y);
                }

           if(fBlocking)
                {
                SetROP2(hdc, R2_NOT);
                SelectObject(hdc, GetStockObject(NULL_BRUSH));
                Rectangle(hdc, ptBeg.x, ptBeg.y, ptEnd.x, ptEnd.y);
                }

           EndPaint(hwnd, &ps);
           return 0;

      case WM_DESTROY :
           PostQuitMessage(0);
           return 0;
      }
 return DefWindowProc(hwnd, iMsg, wParam, lParam);
 }
```

Рис. 6.11 Программа BLOKOUT2

Программа BLOKOUT2 — это та же самая программа BLOKOUT1, за исключением трех новых строчек кода: вызова функции *SetCapture* при обработке сообщения WM_LBUTTONDOWN и вызовов функции *ReleaseCapture* при обработке сообщений WM_LBUTTONUP и WM_CHAR. (Обработка сообщения WM_CHAR позволяет отказаться от захвата мыши при нажатии пользователем клавиши <Escape>.)

Проверьте работу программы теперь: измените размер окна так, чтобы оно стало меньше экрана целиком, начните рисовать прямоугольник внутри рабочей области, а затем уведите курсор за границы рабочей области влево или вниз, и наконец, отпустите кнопку мыши. Программа получит координаты целого прямоугольника. Чтобы его увидеть, увеличьте окно.

Вам следует всегда пользоваться захватом мыши, когда нужно отслеживать сообщения WM_MOUSEMOVE после того, как кнопка мыши была нажата в вашей рабочей области, и до момента отпускания кнопки. Ваша программа станет проще, и ожидания пользователя будут удовлетворены.

Глава 7 Таймер

Таймер в Windows является устройством ввода информации, которое периодически извещает приложение о том, что истек заданный интервал времени. Ваша программа задает Windows интервал, как бы говоря системе: "Подталкивай меня каждые 10 секунд." Тогда Windows посылает вашей программе периодические сообщения WM_TIMER, сигнализируя об истечении интервала времени.

Сначала таймер Windows может показаться менее важным устройством ввода, чем клавиатура или мышь, и, конечно, это верно для многих приложений Windows. Но таймер более полезен, чем вы можете подумать, и не только для программ, которые индицируют время, таких как программа Windows clock, появляющаяся на панели задач. (В эту главу включены две программы clock — аналоговая и цифровая.) Существуют также и другие применения таймера в Windows, некоторые из которых, может быть, не столь очевидны:

- Многозадачность — Хотя Windows 95 является вытесняющей многозадачной средой, иногда самое эффективное решение для программы — как можно быстрее вернуть управление Windows. Если программа должна выполнять большой объем работы, она может разделить задачу на части и отрабатывать каждую часть при получении сообщения WM_TIMER. (Этот вопрос более полно будет рассмотрен в главе 14.)
- Поддержка обновления информации о состоянии — Программа может использовать таймер для вывода на экран обновляемой в "реальном времени" (real-time), постоянно меняющейся информации, связанной либо с системными ресурсами, либо с процессом выполнения определенной задачи.
- Реализация "автосохранения" — Таймер может предложить программе для Windows сохранять работу пользователя на диске всегда, когда истекает заданный интервал времени.
- Завершение демонстрационных версий программ — Некоторые демонстрационные версии программ рассчитаны на свое завершение, скажем, через 30 минут после запуска. Таймер может сигнализировать таким приложениям, когда их время истекает.
- Задание темпа изменения — Графические объекты в играх или окна с результатами в обучающих программах могут нуждаться в задании установленного темпа изменения. Использование таймера устраняет неритмичность, которая могла бы возникнуть из-за разницы в скоростях работы различных микропроцессоров.
- Мультимедиа — Программы, которые проигрывают аудиодиски, звук или музыку, часто допускают воспроизведение звуковых данных в фоновом режиме. Программа может использовать таймер для периодической проверки объема воспроизведенной информации и координации его с информацией, выводимой на экран.

В эту главу также включены разделы, в которых использование таймера распространено и на другие области программирования для Windows. Мы уже касались концепции функций обратного вызова при работе с оконной процедурой, но функции обратного вызова встречаются и при программировании таймера. В этой главе рассказывается и о том, что делать, если программа не может получить доступа к таймеру. Эта тема теперь не столь актуальна, как в прежних версиях Windows, но сам метод, представленный здесь, также может применяться для обработки ошибок в других программах. И наконец, образцы программ, представленные здесь, связаны с такими совершенно нетаймерными задачами, как использование типа окна, известного как "всплывающее окно" (popup), доступ к файлу WIN.INI для получения информации о форматах международного времени и даты, и использование тригонометрии для эффективного вращения на экране графических объектов.

Основы использования таймера

Вы можете присоединить таймер к своей программе при помощи вызова функции *SetTimer*. Функция *SetTimer* содержит целый параметр, задающий интервал, который может находиться в пределах (теоретически) от 1 до 4 294 967 295 миллисекунд, что составляет около 50 дней. Это значение определяет темп, с которым Windows посылает

вашей программе сообщения WM_TIMER. Например, интервал в 1000 миллисекунд заставит Windows каждую секунду посылать вашей программе сообщение.

Если в вашей программе есть таймер, то она вызывает функцию *KillTimer* для остановки потока сообщений от таймера. Вы можете запрограммировать "однократный" таймер, вызывая функцию *KillTimer* при обработке сообщения WM_TIMER. Вызов функции *KillTimer* очищает очередь сообщений от всех необработанных сообщений WM_TIMER. После вызова функции *KillTimer* ваша программа никогда не получит случайного сообщения WM_TIMER.

Система и таймер

Таймер в Windows является относительно простым расширением таймерной логики, встроенной в аппаратуру PC и ROM BIOS. ROM BIOS компьютера инициализирует микросхему таймера так, чтобы она генерировала аппаратное прерывание. Это прерывание иногда называют "тиком таймера". Эти прерывания генерируются каждые 54.925 миллисекунды или примерно 18,2 раза в секунду. Некоторые программы, написанные для MS-DOS, сами обрабатывают это аппаратное прерывание для реализации часов и таймеров.

В программах, сделанных для Windows, так не делается. Windows сама обрабатывает аппаратные прерывания и приложения их не получают. Для каждой программы, где в данный момент установлен таймер, Windows обрабатывает таймерное прерывание путем уменьшения на 1 значения счетчика, изначально переданного вызовом функции *SetTimer*. Когда это значение становится равным 0, Windows помещает сообщение WM_TIMER в очередь сообщений соответствующего приложения и восстанавливает начальное значение счетчика.

Поскольку приложения Windows получают сообщения WM_TIMER из обычной очереди сообщений, вам не нужно беспокоится о том, что ваша программа во время работы будет "прервана" внезапным сообщением WM_TIMER. В этом смысле таймер похож на клавиатуру и мышь: драйвер обрабатывает асинхронные аппаратные прерывания, а Windows преобразует эти прерывания в регулярные, структурированные, последовательные сообщения.

Таймер в Windows имеет ту же самую разрешающую способность 54.925 миллисекунды, что и встроенный таймер PC. Отсюда следуют два важных вывода:

- Приложение Windows при использовании простого таймера не сможет получать сообщения WM_TIMER в темпе, превышающем 18,2 раза в секунду.
- Временной интервал, который вы задаете при вызове функции *SetTimer* всегда округляется вниз до целого числа кратного частоте срабатываний таймера. Например, интервал в 1000 миллисекунд, разделенный на 54.925 миллисекунды равен 18.207 срабатываниям таймера, которые округляются вниз до 18 срабатываний, что фактически составляет интервал в 989, а не 1000 миллисекунд. Для интервалов, меньших 55 миллисекунд, каждое срабатывание таймера генерирует одно сообщение WM_TIMER.

Таймерные сообщения не являются асинхронными

Как уже упоминалось, программы под DOS, написанные для IBM PC и совместимых компьютеров, могут использовать аппаратные срабатывания таймера, перехватывая аппаратное прерывание. Когда происходит аппаратное прерывание, выполнение текущей программы приостанавливается и управление передается обработчику прерываний. Когда прерывание обработано, управление возвращается прерванной программе.

Также как аппаратные прерывания клавиатуры и мыши, аппаратное прерывание таймера иногда называется асинхронным прерыванием, поскольку оно происходит случайно по отношению к прерываемой программе. (Фактически, термин "асинхронные" не совсем точен, поскольку прерывания случаются через одинаковые промежутки времени. Но по отношению к другим процессам прерывания остаются асинхронными.)

Хотя Windows тоже обрабатывает асинхронные таймерные прерывания, сообщения WM_TIMER, которые Windows посылает приложению, не являются асинхронными. Сообщения Windows ставятся в обычную очередь сообщений и обрабатываются как все остальные сообщения. Поэтому, если вы задаете функции *SetTimer* 1000 миллисекунд, то вашей программе не гарантируется получение сообщения WM_TIMER каждую секунду или даже (как уже упоминалось выше) каждые 989 миллисекунд. Если ваше приложение занято больше, чем секунду, то оно вообще не получит ни одного сообщения WM_TIMER в течение этого времени. Вы можете убедиться в этом с помощью представленных в этой главе программ. Фактически, Windows обрабатывает сообщения WM_TIMER во многом также, как сообщения WM_PAINT. Оба эти сообщения имеют низкий приоритет, и программа получит только их, если в очереди нет других сообщений.

Сообщения WM_TIMER похожи на сообщения WM_PAINT и в другом смысле: Windows не хранит в очереди сообщений несколько сообщений WM_TIMER. Вместо этого Windows объединяет несколько сообщений WM_TIMER из очереди в одно сообщение. Поэтому, приложение не получает за раз группу сообщений WM_TIMER, хотя оно может получить два таких сообщения, быстро следующих одно за другим. В результате приложение не может определить число "потерянных" сообщений WM_TIMER.

Если вы воспользуетесь строкой заголовка программы clock для перемещения окна по экрану, эта программа прекратит получение сообщений WM_TIMER. Когда программа clock снова получит управление, часы "прыгнут" вперед, чтобы исправить время, но происходит это вовсе не потому, что программа получает несколько сообщений WM_TIMER подряд. Программа должна определить реальное время и затем восстановить свое состояние. Сообщения WM_TIMER только информируют программу, когда ей следует обновить данные. Сама программа не может хранить время, используя подсчет сообщений WM_TIMER. (Дальше в этой главе мы напишем два приложения clock, которые обновляют данные каждую секунду, и увидим, как это делается.)

Давайте условимся, что при рассказе о таймере будут использоваться выражения типа "получение сообщения WM_TIMER каждую секунду". Но запомните, что эти сообщения — это не точные таймерные прерывания.

Использование таймера: три способа

Если вам нужен таймер для измерения продолжительности работы вашей программы, вы, вероятно, вызовите *SetTimer* из функции *WinMain* или при обработке сообщения WM_CREATE, а *KillTimer* в ответ на сообщение WM_DESTROY. Установка таймера в функции *WinMain* обеспечивает простейшую обработку ошибки, если таймер недоступен. Вы можете использовать таймер одним из трех способов, в зависимости от параметров функции *SetTimer*.

Первый способ

Этот простейший способ заставляет Windows посылать сообщения WM_TIMER обычной оконной процедуре приложения. Вызов функции *SetTimer* выглядит следующим образом:

```
SetTimer(hwnd, 1, iMsecInterval, NULL);
```

Первый параметр — это описатель того окна, чья оконная процедура будет получать сообщения WM_TIMER. Вторым параметром является идентификатор таймера, значение которого должно быть отличным от нуля. В этом примере он произвольно установлен в 1. Третий параметр — это 32-разрядное беззнаковое целое, которое задает интервал в миллисекундах. Значение 60000 задает генерацию сообщений WM_TIMER один раз в минуту.

Вы можете в любое время остановить поток сообщений WM_TIMER (даже во время обработки сообщения WM_TIMER), вызвав функцию:

```
KillTimer(hwnd, 1);
```

Вторым параметром здесь является тот же идентификатор таймера, который использовался при вызове функции *SetTimer*. Вы должны перед завершением вашей программы в ответ на сообщение WM_DESTROY уничтожить все активные таймеры.

Когда ваша оконная процедура получает сообщение WM_TIMER, значение *wParam* равно значению идентификатора таймера (который равен 1 в приведенном примере), а *lParam* равно 0. Если вам нужно более одного таймера, используйте для каждого таймера свой идентификатор. Значение параметра *wParam* позволит различать передаваемые в оконную процедуру сообщения WM_TIMER. Для того, чтобы вашу программу было легче читать, для разных идентификаторов таймера лучше использовать инструкции *#define*:

```
#define TIMER_SEC 1
#define TIMER_MIN 2
```

Два таймера можно задать, если дважды вызвать функцию *SetTimer*:

```
SetTimer(hwnd, TIMER_SEC, 1000, NULL);
SetTimer(hwnd, TIMER_MIN, 60000, NULL);
```

Логика обработки сообщения WM_TIMER может выглядеть 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;
            }
       return 0;
```

Если вы захотите установить новое время срабатывания для существующего таймера, уничтожьте таймер и снова вызовите функцию *SetTimer*. В этих инструкциях предполагается, что идентификатор таймера равен 1:

```
KillTimer(hwnd, 1);
```

```
SetTimer(hwnd, 1, iMsecInterval, NULL);
```

Параметр *iMsecInterval* задается равным новому времени срабатывания в миллисекундах. Вы можете использовать эту схему в программах часов, имеющих возможность выбора — показывать секунды или нет. Вам нужно просто изменить таймерный интервал с 1000 на 60 000 миллисекунд.

Что делать, если таймер недоступен

В ранних версиях Windows в каждый конкретный момент времени во всей системе могли быть активными только 16 таймеров. Пользователи Windows быстро обнаружили это, пытаясь запустить как можно больше программ Clock. Попытка запустить программу 17 раз приводила к появлению сообщения об ошибке "No more clocks or timers" (Нет доступных таймеров). Хотя Windows 3.0 удвоила количество допустимых таймеров до 32, а Windows 95 практически вообще сняла какие бы то ни было ограничения, включение обработки ошибок в программе Windows по-прежнему считается хорошим стилем программирования. Рассмотрим обработку таких ситуаций.

Если нет доступных таймеров, возвращаемым значением функции *SetTimer* является NULL. Ваша программа могла бы нормально работать и без таймера, но если таймер вам необходим (как в программе clock), то у приложения не остается выбора, как только завершиться из-за невозможности работать. Если вы вызываете в *WinMain* функцию *SetTimer*, то вы можете завершить свою программу, просто возвращая FALSE из *WinMain*.

Предположим, что вам нужен 1000-миллисекундный таймер. Сразу за вызовом *CreateWindow*, но перед циклом обработки сообщений, вы могли бы вставить следующую инструкцию:

```
if(!SetTimer(hwnd, 1, 1000, NULL))
    return FALSE;
```

Но это некрасивый способ завершения программы. Пользователь останется в неведении, почему не загружается его приложение. Гораздо удобнее — и намного проще — для вывода сообщения на экран использовать окно сообщений Windows. (Полный рассказ об окнах сообщений ожидает вас в главе 12, а сейчас мы начнем знакомиться с ними.)

Окно сообщений — это всплывающее окно, которое появляется всегда в центре экрана. В окнах сообщений есть строка заголовка, но нет рамки, позволяющей изменять размеры окна. Строка заголовка обычно содержит имя приложения. Окно сообщений включает в себя само сообщение и одну, две или три кнопки (какие-либо сочетания кнопок OK, Retry, Cancel, Yes, No и других). В окне сообщений может также находиться ранее определенный значок: строчное "i" (что означает "information" — информация), восклицательный, вопросительный или запрещающий знаки. Последний представляет собой белый символ X на красном фоне (как настоящий знак "стоп"). Вы, вероятно, уже видели множество окон сообщений при работе с Windows.

Этот код создает окно сообщений, которое вы можете использовать, когда функция *SetTimer* терпит неудачу при установке таймера:

```
if(!SetTimer(hwnd, 1, 1000, NULL))
{
       MessageBox(hwnd,
              "Too many clocks or timers!",
              "Program Name",
              MB_ICONEXCLAMATION | MB_OK);
       return FALSE;
}
```

Это окно сообщений представлено на рис. 7.1. Когда пользователь нажимает клавишу <Enter> или щелкает на кнопке OK, *WinMain* завершается и возвращает значение FALSE.

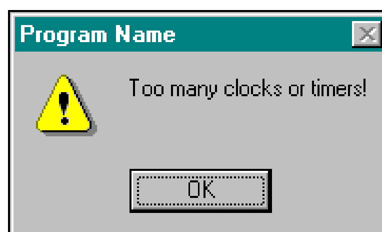


Рис. 7.1 Окно сообщений при "дружественном" завершении программы

По умолчанию окна сообщений являются "модальными окнами приложения" (application modal). Это означает, что пользователь должен как-то отреагировать на окно сообщений перед тем, как приложение продолжит работу. Однако, пользователь может переключиться на другие приложения. Следовательно, почему бы не дать пользователю возможность закрыть одно из использующих таймер приложений и тогда уже успешно загрузить свое приложение с помощью следующего кода:

```
while ( !SetTimer(hwnd, 1, 1000, NULL) )
       if ( MessageBox(hwnd, "Too many clocks or timers!", "Program Name",
                     MB_ICONEXCLAMATION | MB_RETRYCANCEL) == IDCANCEL )
              return FALSE;
```

Окно сообщений, приведенное на рис. 7.2, имеет две кнопки с надписями Retry и Cancel. Если пользователь щелкает на кнопке Cancel, функция *MessageBox* возвращает значение равное IDCANCEL, и программа завершается. Если пользователь щелкает на кнопке Retry, функция *SetTimer* вызывается снова.

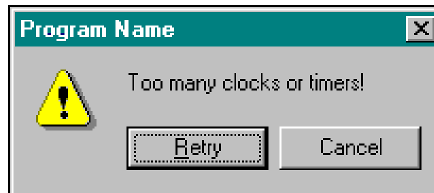


Рис. 7.2 Окно сообщений, которое предлагает выбор

Пример программы

На рис. 7.3 представлен пример программы, в которой пользователь использует таймер. Эта программа, названная BEEPER1, устанавливает таймер на временной интервал, равный 1 секунде. При получении сообщения WM_TIMER, программа изменяет цвет рабочей области с голубого на красный или с красного на голубой и, вызывая функцию *MessageBeep*, издает гудок. Хотя в документации функция *MessageBeep* описывается в связи с функцией *MessageBox*, реально она всегда может использоваться для звукового сигнала. В компьютерах, оборудованных звуковой платой, можно использовать различные параметры MB_ICON в функции *MessageBeep* для воспроизведения разнообразных звуков.

В программе BEEPER1 таймер устанавливается в функции *WinMain*, а сообщения WM_TIMER обрабатываются в оконной процедуре *WndProc*. При обработке сообщения WM_TIMER программа BEEPER1 вызывает функцию *MessageBeep*, инвертирует значение *bFlipFlop* и, для генерации сообщения WM_PAINT, делает окно недействительным. При обработке сообщения WM_PAINT программа BEEPER1 с помощью вызова функции *GetClientRect* получает структуру RECT, соответствующую размерам окна целиком, и с помощью вызова функции *FillRect*, закрашивает окно.

BEEPER1.MAK

```
#----------------------
# BEEPER1.MAK make file
#----------------------

beeper1.exe : beeper1.obj
     $(LINKER) $(GUIFLAGS) -OUT:beeper1.exe beeper1.obj $(GUILIBS)

beeper1.obj : beeper1.c
     $(CC) $(CFLAGS) beeper1.c
```

BEEPER1.C

```
/*-----------------------------------------
   BEEPER1.C  -- Timer Demo Program No. 1
                 (c) Charles Petzold, 1996
  -----------------------------------------*/

#include <windows.h>

#define ID_TIMER    1

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Beeper1";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;
```

```
     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);
     hwnd = CreateWindow(szAppName, "Beeper1 Timer Demo",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     while(!SetTimer(hwnd, ID_TIMER, 1000, NULL))
          if(IDCANCEL == MessageBox(hwnd,
                             "Too many clocks or timers!", szAppName,
                             MB_ICONEXCLAMATION | MB_RETRYCANCEL))
               return FALSE;

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static BOOL fFlipFlop = FALSE;
     HBRUSH      hBrush;
     HDC         hdc;
     PAINTSTRUCT ps;
     RECT        rc;

     switch(iMsg)
          {
          case WM_TIMER :
               MessageBeep(0);

               fFlipFlop = !fFlipFlop;
               InvalidateRect(hwnd, NULL, FALSE);

               return 0;

          case WM_PAINT :
               hdc = BeginPaint(hwnd, &ps);

               GetClientRect(hwnd, &rc);

               hBrush = CreateSolidBrush(fFlipFlop ? RGB(255,0,0) :
                                                     RGB(0,0,255));
               FillRect(hdc, &rc, hBrush);
```

```
            EndPaint(hwnd, &ps);
            DeleteObject(hBrush);
            return 0;

        case WM_DESTROY :
            KillTimer(hwnd, ID_TIMER);
            PostQuitMessage(0);
            return 0;
        }
    return DefWindowProc(hwnd, iMsg, wParam, lParam);
    }
```

Рис. 7.3 Программа BEEPER1

Поскольку BEEPER1 издает звуковой сигнал при каждом получении сообщения WM_TIMER, вы сможете, загружая BEEPER1 и выполняя какие-то действия в Windows, получить представление о нерегулярной природе сообщений WM_TIMER. Например, попытайтесь сдвинуть или изменить размер окна программы BEEPER1. Это заставит программу войти в "модальный цикл обработки сообщений" (modal message loop). Windows предотвращает все действия, которые могут помешать операции перемещения или изменения размера окна, перехватывая все сообщения с помощью цикла обработки сообщений, организуемого в самой операционной системе, чтобы сообщения не поступали в цикл обработки сообщений вашей программы. Большинство сообщений, предназначенных для окна программы, проходя через этот цикл, просто отбрасываются, поэтому BEEPER1 перестает сигналить. Если вы прекратите двигать окно или менять его размер, то заметите, что BEEPER1 не получила всех пропущенных сообщений WM_TIMER, они были потеряны, хотя первые два сообщения могут появиться с интервалом менее одной секунды.

Второй способ

При первом способе установки таймера сообщения WM_TIMER посылаются в обычную оконную процедуру. С помощью второго способа вы можете заставить Windows пересылать сообщение таймера другой функции из вашей программы.

Функция, которая будет получать эти таймерные сообщения, называется функцией "обратного вызова" (call-back). Это функция вашей программы, которую вызывает Windows. Вы сообщаете Windows адрес этой функции, а позже Windows вызывает ее. Это должно быть вам знакомым, поскольку оконная процедура программы фактически является такой функцией обратного вызова. При регистрации класса окна вы сообщаете Windows адрес оконной процедуры, и затем Windows, посылая сообщения программе, вызывает эту функцию.

SetTimer — это не единственная функция Windows, использующая функцию обратного вызова. Функции *CreateDialog* и *DialogBox* (обсуждаемые в главе 14) используют функции обратного вызова для обработки сообщений в окне диалога; несколько функций Windows (*EnumChildWindows*, *EnumFonts*, *EnumObjects*, *EnumProps* и *EnumWindows*) передают перечисляемую информацию функциям обратного вызова; для нескольких реже используемых функций (*GrayString*, *LineDDA* и *SetWindowsHookEx*) также требуются функции обратного вызова.

Также как и оконная процедура, функция обратного вызова должна определяться как CALLBACK, поскольку Windows вызывает ее вне кодового пространства программы. Параметры функции обратного вызова и ее возвращаемое значение зависят от назначения функции обратного вызова. В случае функции обратного вызова для таймера, входными параметрами являются те же параметры, что и параметры оконной процедуры. Таймерная функция обратного вызова не имеет возвращаемого в Windows значения.

Давайте назовем функцию обратного вызова *TimerProc*. (Вы можете дать ей любое имя.) Она будет обрабатывать только сообщения WM_TIMER.

```
VOID CALLBACK TimerProc(HWND hwnd, _UINT iMsg, _UINT iTimerID, DWORD dwTime)
{
        [обработка сообщений WM_TIMER]
}
```

Входной параметр *hwnd* — это описатель окна, задаваемый при вызове функции *SetTimer*. Windows будет посылать функции *TimerProc* только сообщения WM_TIMER, следовательно параметр *iMsg* всегда будет равен WM_TIMER. Значение *iTimerID* — это идентификатор таймера, а значение *dwTime* — системное время.

Как уже говорилось выше, для первого способа установки таймера требуется следующий вызов функции *SetTimer*:

```
SetTimer(hwnd, iTimerID, iMsecInterval, NULL);
```

При использовании функции обратного вызова для обработки сообщений WM_TIMER, четвертый параметр функции *SetTimer* заменяется адресом функции обратного вызова:

```
SetTimer(hwnd, iTimerID, iMsecInterval,(TIMERPROC) TimerProc);
```

Пример программы

Давайте рассмотрим пример программы, чтобы вы могли увидеть, как это все работает. Программа BEEPER2, представленная на рис. 7.4, функционально такая же, как и программа BEEPER1 за исключением того, что Windows посылает таймерные сообщения не в *WndProc*, а в *TimerProc*.

BEEPER2.MAK

```
#-----------------------
# BEEPER2.MAK make file
#-----------------------

beeper2.exe : beeper2.obj
     $(LINKER) $(GUIFLAGS) -OUT:beeper2.exe beeper2.obj $(GUILIBS)

beeper2.obj : beeper2.c
     $(CC) $(CFLAGS) beeper2.c
```

BEEPER2.C

```
/*----------------------------------------
   BEEPER2.C -- Timer Demo Program No. 2
               (c) Charles Petzold, 1996
  ----------------------------------------*/

#include <windows.h>

#define ID_TIMER    1

LRESULT CALLBACK WndProc  (HWND, UINT, WPARAM, LPARAM);
VOID    CALLBACK TimerProc(HWND, UINT, UINT,   DWORD );

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Beeper2";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Beeper2 Timer Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL);

     while(!SetTimer(hwnd, ID_TIMER, 1000,(TIMERPROC) TimerProc))
          if(IDCANCEL == MessageBox(hwnd,
                              "Too many clocks or timers!", szAppName,
```

```
                              MB_ICONEXCLAMATION | MB_RETRYCANCEL))
               return FALSE;

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     switch(iMsg)
          {
          case WM_DESTROY :
               KillTimer(hwnd, ID_TIMER);
               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }

VOID CALLBACK TimerProc(HWND hwnd, UINT iMsg, UINT iTimerID, DWORD dwTime)
     {
     static BOOL fFlipFlop = FALSE;
     HBRUSH      hBrush;
     HDC         hdc;
     RECT        rc;

     MessageBeep(0);
     fFlipFlop = !fFlipFlop;

     GetClientRect(hwnd, &rc);

     hdc = GetDC(hwnd);
     hBrush = CreateSolidBrush(fFlipFlop ? RGB(255,0,0) : RGB(0,0,255));

     FillRect(hdc, &rc, hBrush);
     ReleaseDC(hwnd, hdc);
     DeleteObject(hBrush);
     }
```

Рис. 7.4 Программа BEEPER2

Третий способ

Третий способ установки таймера напоминает второй, за исключением того, что параметр *hwnd* функции *SetTimer* устанавливается в NULL, а второй параметр (обычно идентификатор таймера) игнорируется. Функция возвращает ID таймера:

```
iTimerID = SetTimer(NULL, 0, wMsecInterval,(TIMERPROC) TimerProc);
```

Возвращаемое функцией *SetTimer* значение *iTimerID* будет равно NULL, если таймер недоступен.

Первый параметр функции *KillTimer* (обычно описатель окна) также должен быть равен NULL. Идентификатор таймера должен быть равен значению, возвращаемому функцией *SetTimer*.

```
KillTimer(NULL, iTimerID);
```

Параметр *hwnd*, передаваемый в *TimerProc*, также должен быть равен NULL.

Такой метод установки таймера используется редко. Он удобен, если в программе в разное время делается много вызовов функции *SetTimer*, и при этом не запоминаются те таймерные идентификаторы, которые уже использовались.

Теперь, поскольку вы знаете, как использовать таймер в Windows, вы готовы к знакомству с парой полезных программ, использующих таймер.

Использование таймера для часов

Часы — это наиболее очевидное применение таймера, поэтому давайте рассмотрим два типа часов, одни цифровые, другие аналоговые. Программа DIGCLOCK, представленная на рис. 7.5, создает всплывающее окно, которое позиционируется в верхнем правом углу экрана. Программа выводит день недели, дату и время, как показано на рис. 7.6.

DIGCLOCK.MAK

```
#------------------------
# DIGCLOCK.MAK make file
#------------------------

digclock.exe : digclock.obj
     $(LINKER) $(GUIFLAGS) -OUT:digclock.exe digclock.obj $(GUILIBS)

digclock.obj : digclock.c
     $(CC) $(CFLAGS) digclock.c
```

DIGCLOCK.C

```
/*-----------------------------------------
   DIGCLOCK.C -- Digital Clock Program
                 (c) Charles Petzold, 1996
  -----------------------------------------*/

#include <windows.h>
#include <time.h>

#define ID_TIMER    1

#define YEAR (datetime->tm_year % 100)
#define MONTH(datetime->tm_mon  + 1)
#define MDAY (datetime->tm_mday)
#define WDAY (datetime->tm_wday)
#define HOUR (datetime->tm_hour)
#define MIN  (datetime->tm_min)
#define SEC  (datetime->tm_sec)

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
void SizeTheWindow(int *, int *, int *, int *);

char  sDate[2], sTime[2], sAMPM[2][5];
int   iDate, iTime;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "DigClock";
     HWND        hwnd;
     MSG         msg;
     int         xStart, yStart, xClient, yClient;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
```

```
    wndclass.hIcon         = NULL;
    wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
    wndclass.lpszMenuName  = NULL;
    wndclass.lpszClassName = szAppName;
    wndclass.hIconSm       = NULL;

    RegisterClassEx(&wndclass);

    SizeTheWindow(&xStart, &yStart, &xClient, &yClient);
    hwnd = CreateWindow(szAppName, szAppName,
                        WS_POPUP | WS_DLGFRAME | WS_SYSMENU,
                        xStart,  yStart,
                        xClient, yClient,
                        NULL, NULL, hInstance, NULL);

    if(!SetTimer(hwnd, ID_TIMER, 1000, NULL))
         {
         MessageBox(hwnd, "Too many clocks or timers!", szAppName,
                    MB_ICONEXCLAMATION | MB_OK);
         return FALSE;
         }

    ShowWindow(hwnd, SW_SHOWNOACTIVATE);
    UpdateWindow(hwnd);

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

void SizeTheWindow(int *pxStart,  int *pyStart, int *pxClient, int *pyClient)
    {
    HDC        hdc;
    TEXTMETRIC tm;

    hdc = CreateIC("DISPLAY", NULL, NULL, NULL);
    GetTextMetrics(hdc, &tm);
    DeleteDC(hdc);

    *pxClient = 2 * GetSystemMetrics(SM_CXDLGFRAME) + 16*tm.tmAveCharWidth;
    *pxStart  =     GetSystemMetrics(SM_CXSCREEN)   - *pxClient;
    *pyClient = 2 * GetSystemMetrics(SM_CYDLGFRAME) + 2*tm.tmHeight;
    *pyStart  =     0;
    }

void SetInternational(void)
    {
    static char cName [] = "intl";

    iDate = GetProfileInt(cName, "iDate", 0);
    iTime = GetProfileInt(cName, "iTime", 0);

    GetProfileString(cName, "sDate",  "/", sDate,    2);
    GetProfileString(cName, "sTime",  ":", sTime,    2);
    GetProfileString(cName, "s1159", "AM", sAMPM[0], 5);
    GetProfileString(cName, "s2359", "PM", sAMPM[1], 5);
    }

void WndPaint(HWND hwnd, HDC hdc)
    {
```

```
     static char szWday[] = "Sun\0Mon\0Tue\0Wed\0Thu\0Fri\0Sat";
     char        cBuffer[40];
     int         iLength;
     RECT        rect;
     struct tm  *datetime;
     time_t      lTime;

     time(&lTime);
     datetime = localtime(&lTime);

     iLength = wsprintf(cBuffer, "  %s  %d%s%02d%s%02d  \r\n",
              (PSTR) szWday + 4 * WDAY,
               iDate == 1 ? MDAY  : iDate == 2 ? YEAR  : MONTH,(PSTR) sDate,
               iDate == 1 ? MONTH : iDate == 2 ? MONTH : MDAY, (PSTR) sDate,
               iDate == 1 ? YEAR  : iDate == 2 ? MDAY  : YEAR);

     if(iTime == 1)
          iLength += wsprintf(cBuffer + iLength, "  %02d%s%02d%s%02d  ",
                              HOUR,(PSTR) sTime, MIN,(PSTR) sTime, SEC);
     else
          iLength += wsprintf(cBuffer + iLength, "  %d%s%02d%s%02d %s  ",
                             (HOUR % 12) ?(HOUR % 12) : 12,
                             (PSTR) sTime, MIN,(PSTR) sTime, SEC,
                             (PSTR) sAMPM [HOUR / 12]);

     GetClientRect(hwnd, &rect);
     DrawText(hdc, cBuffer, -1, &rect, DT_CENTER | DT_NOCLIP);
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     HDC         hdc;
     PAINTSTRUCT ps;

     switch(iMsg)
          {
          case WM_CREATE :
               SetInternational();
               return 0;

          case WM_TIMER :
               InvalidateRect(hwnd, NULL, FALSE);
               return 0;

          case WM_PAINT :
               hdc = BeginPaint(hwnd, &ps);
               WndPaint(hwnd, hdc);
               EndPaint(hwnd, &ps);
               return 0;

          case WM_WININICHANGE :
               SetInternational();
               InvalidateRect(hwnd, NULL, TRUE);
               return 0;

          case WM_DESTROY :
               KillTimer(hwnd, ID_TIMER);
               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

Рис. 7.5 Программа DIGCLOCK

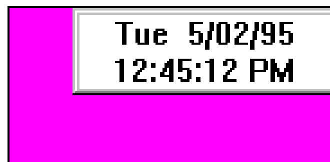


Рис. 7.6 Окно программы DIGCLOCK

Во всех приведенных до сих пор программах использовался стиль окна WS_OVERLAPPEDWINDOW в качестве третьего параметра функции *CreateWindow*. В программе DIGCLOCK используется стиль окна:

```
WS_POPUP | WS_DLGFRAME | WS_SYSMENU
```

Это выражение задает "всплывающее" (popup) окно с рамкой окна диалога и системным меню. Стиль всплывающего окна чаще используется в окнах диалога и окнах сообщений, и очень редко для приложений. Кроме этого в программе DIGCLOCK используется еще и другой вариант вызова функции *ShowWindow*:

```
ShowWindow(hwnd, SW_SHOWNOACTIVATE);
```

Обычно окно программы становится активным при запуске. SW_SHOWNOACTIVATE сообщает Windows о том, что программа DIGCLOCK не должна изменять активное окно. Однако вы можете сделать активным окно DIGCLOCK, щелкнув на нем мышью, или нажав клавиши <Alt>+<Tab> или <Alt>+<Esc>. Хотя в программе DIGCLOCK нет символа системного меню, но если она активна, вы по-прежнему можете получить доступ к системному меню, нажимая комбинацию клавиш <Alt>+<Spacebar> (<Alt>+<пробел>).

Позиционирование и изменение размеров всплывающего окна

Всплывающее окно программы DIGCLOCK устанавливается в верхнем правом углу экрана. Окно должно быть достаточно большим, чтобы вместить две строки текста по 16 символов в каждой. Процедура *SizeTheWindow* в DIGCLOCK.C определяет подходящие параметры для использования их при вызове функции *CreateWindow*. Обычно программа не может получить размер текста без предварительного создания самого окна, поскольку описатель окна нужен для получения описателя контекста устройства. Программе DIGCLOCK удается обойти эту проблему путем получения информации об экране из описателя контекста устройства с помощью функции *CreateIC*. Эта функция похожа на функцию *CreateDC*, обычно используемую для создания контекста принтера (как будет показано в главе 15), но эта функция используется для получения информации из контекста устройства. Размера текста в сочетании с информацией, полученной от *GetSystemMetrics*, достаточно для получения начального положения и размеров окна.

Получение даты и времени

В своей функции *WndPaint* программа DIGCLOCK использует функции языка С *time* и *locatetime* для определения текущих даты и времени. Функция *locatetime* помещает всю необходимую вам информацию в структуру; несколько макроопределений в начале программы помогают сделать вызовы *wsprintf* более читаемыми. (Вы должны избегать выполнения функций MS-DOS и ROM BIOS в вашей программе для Windows; вместо этого пользуйтесь функциями Windows или библиотекой языка С периода выполнения.)

Обеспечение международной поддержки

Windows обеспечивает международную поддержку. Файл WIN.INI, создаваемый при инсталляции Windows, содержит раздел, озаглавленный [*intl*]. Это список информации, относящейся к формату даты, времени, валюты и чисел. Вы можете выводить на экран даты в одном из трех разных форматов: месяц-день-год, год-месяц-день или день-месяц-год. Разделителем между этими тремя числами может быть наклонная черта, тире, точка или, фактически, любой понравившийся вам символ. Вы можете выводить время либо в 12-ти, либо в 24-часовом формате; для разделения часов, минут и секунд обычно используется точка или двоеточие.

Функция *SetInternational* в DIGCLOCK извлекает эту информацию для форматирования из WIN.INI с помощью функций *GetProfileInt* (для целых) и *GetProfileString* (для строк). Эти вызовы должны содержать значения по умолчанию, если Windows не сможет найти требуемые значения в WIN.INI. Функция *SetInternational* сохраняет эти значения в глобальных переменных, имеющих такие же имена, что и текстовые строки, которые идентифицируют их в WIN.INI. Функция *WndPaint* использует эти значения, полученные из WIN.INI, для форматирования выводимых на экран даты и времени, а затем вызывает функцию *DrawText* для выравнивания двух строк текста в окне.

Независимо от времени получения сообщения WM_TIMER, оконная процедура программы DIGCLOCK, для выработки сообщения WM_PAINT, делает окно недействительным. Но *WndProc* также делает окно недействительным при получении сообщения WM_WININICHANGE. Любое приложение, которое изменяет файл WIN.INI, посылает сообщение WM_WININICHANGE всем активным приложениям Windows. Если секция [*intl*]

WIN.INI файла изменяется, то программа DIGCLOCK узнает об этом и получит новую международную информацию. Чтобы увидеть, как это работает, загрузите DIGCLOCK, дважды щелкните на иконке Regional Settings в Control Panel и измените либо формат даты, либо разделитель даты, либо формат времени, либо разделитель времени. Теперь нажмите <Enter>. Файл WIN.INI обновляется, выводимая программой DIGCLOCK на экран информация отражает это изменение — так в Windows работает магия сообщений.

Когда оконная процедура получает сообщение WM_WININICHANGE, она делает окно недействительным, используя функцию:

```
InvalidateRect(hwnd, NULL, TRUE);
```

Когда программа DIGCLOCK получает сообщение WM_TIMER, она делает окно недействительным, используя функцию:

```
InvalidateRect(hwnd, NULL, FALSE);
```

Значение TRUE в последнем параметре сообщает Windows о необходимости обновить фон окна перед рисованием. Значение FALSE просто сообщает Windows о необходимости перерисовки на существующем фоне. При обработке сообщения WM_TIMER мы используем FALSE, поскольку это снижает нежелательное мерцание экрана. Вы можете удивиться, зачем нам вообще нужно использовать TRUE.

Значение TRUE необходимо при обработке сообщения WM_WININICHANGE, поскольку длина выводимой строки может измениться на несколько символов, если вы измените формат времени с 12 на 24 часа. Однако, наибольшее изменение происходит в результате сообщения WM_TIMER, оно равно двум символам — например, при переходе даты с 12/31/95 на 1/1/96 — и поэтому строка формата, которую использует *WndPaint* для вывода информации на экран содержит по два пробела на концах для учета такого изменения в длине и учета пропорционального шрифта.

Мы могли бы также обрабатывать в программе DIGCLOCK сообщения WM_TIMECHANGE, которые извещают приложение об изменении системных даты и времени. Но это не нужно, поскольку сообщения WM_TIMER вызывают обновление в программе DIGCLOCK каждую секунду. Обработка сообщений WM_TIMECHANGE имела бы больше смысла для таких часов, которые обновляют информацию каждую минуту.

Создание аналоговых часов

Программе аналоговых часов нет необходимости учитывать международные аспекты, но сложность графики с лихвой компенсирует такое упрощение. Для того, чтобы сделать эту программу правильно, вам понадобится узнать, как использовать режимы отображения и даже немного тригонометрию. Программа ANACLOCK представлена на рис. 7.7, а ее вид на экране — на рис. 7.8.

ANACLOCK.MAK

```
#-----------------------
# ANACLOCK.MAK make file
#-----------------------

anaclock.exe : anaclock.obj
     $(LINKER) $(GUIFLAGS) -OUT:anaclock.exe anaclock.obj $(GUILIBS)

anaclock.obj : anaclock.c
     $(CC) $(CFLAGS) anaclock.c
```

ANACLOCK.C

```
/*----------------------------------------
   ANACLOCK.C -- Analog Clock Program
                 (c) Charles Petzold, 1996
  ----------------------------------------*/

#include <windows.h>
#include <string.h>
#include <time.h>
#include <math.h>

#define ID_TIMER    1
#define TWOPI       (2 * 3.14159)

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
```

```
                 PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "AnaClock";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;
     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = NULL;
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = NULL;

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Analog Clock",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     if(!SetTimer(hwnd, ID_TIMER, 1000, NULL))
          {
          MessageBox(hwnd, "Too many clocks or timers!", szAppName,
                      MB_ICONEXCLAMATION | MB_OK);
          return FALSE;
          }

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

void SetIsotropic(HDC hdc, int cxClient, int cyClient)
     {
     SetMapMode(hdc, MM_ISOTROPIC);
     SetWindowExtEx(hdc, 1000, 1000, NULL);
     SetViewportExtEx(hdc, cxClient / 2, -cyClient / 2, NULL);
     SetViewportOrgEx(hdc, cxClient / 2,  cyClient / 2, NULL);
     }

void RotatePoint(POINT pt[], int iNum, int iAngle)
     {
     int   i;
     POINT ptTemp;

     for(i = 0; i < iNum; i++)
          {
          ptTemp.x =(int)(pt[i].x * cos(TWOPI * iAngle / 360) +
                            pt[i].y * sin(TWOPI * iAngle / 360));

          ptTemp.y =(int)(pt[i].y * cos(TWOPI * iAngle / 360) -
```

```
                         pt[i].x * sin(TWOPI * iAngle / 360));

          pt[i] = ptTemp;
          }
     }

void DrawClock(HDC hdc)
     {
     int   iAngle;
     POINT pt[3];

     for(iAngle = 0; iAngle < 360; iAngle += 6)
          {
          pt[0].x =   0;
          pt[0].y = 900;

          RotatePoint(pt, 1, iAngle);

          pt[2].x = pt[2].y = iAngle % 5 ? 33 : 100;

          pt[0].x -= pt[2].x / 2;
          pt[0].y -= pt[2].y / 2;

          pt[1].x  = pt[0].x + pt[2].x;
          pt[1].y  = pt[0].y + pt[2].y;

          SelectObject(hdc, GetStockObject(BLACK_BRUSH));

          Ellipse(hdc, pt[0].x, pt[0].y, pt[1].x, pt[1].y);
          }
     }

void DrawHands(HDC hdc, struct tm *datetime, BOOL bChange)
     {
     static POINT pt[3][5] = { 0, -150, 100, 0, 0, 600, -100, 0, 0, -150,
                               0, -200,  50, 0, 0, 800,  -50, 0, 0, -200,
                               0,    0,   0, 0, 0,   0,    0, 0, 0,  800 };
     int          i, iAngle[3];
     POINT        ptTemp[3][5];

     iAngle[0] =(datetime->tm_hour * 30) % 360 + datetime->tm_min / 2;
     iAngle[1] =  datetime->tm_min  *  6;
     iAngle[2] =  datetime->tm_sec  *  6;

     memcpy(ptTemp, pt, sizeof(pt));

     for(i = bChange ? 0 : 2; i < 3; i++)
          {
          RotatePoint(ptTemp[i], 5, iAngle[i]);

          Polyline(hdc, ptTemp[i], 5);
          }
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int          cxClient, cyClient;
     static struct tm    dtPrevious;
     BOOL                bChange;
     HDC                 hdc;
     PAINTSTRUCT         ps;
     time_t              lTime;
     struct tm          *datetime;
```

```
switch(iMsg)
     {
     case WM_CREATE :
          time(&lTime);
          datetime = localtime(&lTime);

          dtPrevious = * datetime;
          return 0;

     case WM_SIZE :
          cxClient = LOWORD(lParam);
          cyClient = HIWORD(lParam);
          return 0;

     case WM_TIMER :
          time(&lTime);
          datetime = localtime(&lTime);

          bChange = datetime->tm_hour != dtPrevious.tm_hour ||
                    datetime->tm_min  != dtPrevious.tm_min;

          hdc = GetDC(hwnd);

          SetIsotropic(hdc, cxClient, cyClient);

          SelectObject(hdc, GetStockObject(WHITE_PEN));
          DrawHands(hdc, &dtPrevious, bChange);

          SelectObject(hdc, GetStockObject(BLACK_PEN));
          DrawHands(hdc, datetime, TRUE);

          ReleaseDC(hwnd, hdc);

          dtPrevious = *datetime;
          return 0;

     case WM_PAINT :
          hdc = BeginPaint(hwnd, &ps);

          SetIsotropic(hdc, cxClient, cyClient);
          DrawClock   (hdc);
          DrawHands   (hdc, &dtPrevious, TRUE);

          EndPaint(hwnd, &ps);
          return 0;

     case WM_DESTROY :
          KillTimer(hwnd, ID_TIMER);
          PostQuitMessage(0);
          return 0;
     }
return DefWindowProc(hwnd, iMsg, wParam, lParam);
}
```

Рис. 7.7 Программа ANACLOCK

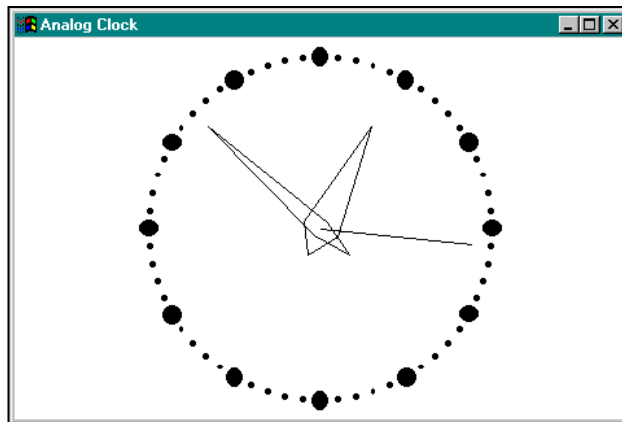


Рис. 7.8 Окно программы ANACLOCK

Режим отображения "isotropic" является идеальным для такого приложения, а устанавливает его в ANACLOCK.C функция *SetIsotropic*. После вызова функции *SetMapMode*, она устанавливает протяженности окна равными 1000, а протяженности области вывода равными половине ширины рабочей области и половине высоты рабочей области, взятой со знаком минус. Начало координат области вывода устанавливается в центр рабочей области. Как рассказывалось в главе 4, в результате создается Декартова система координат с точкой (0, 0) в центре рабочей области и размерами, равными 1000 единиц по всем направлениям.

Функция *RotatePoint* вводит в игру тригонометрию. Три параметра функции — это массив из одной или более точек, число точек в массиве и угол поворота в градусах. Функция вращает точки по часовой стрелке (как это принято для часов) вокруг начала координат. Например, если переданная функции точка имеет координаты (0, 100) — фактически это положение 12:00 — и угол поворота равен 90 градусам, то точка оказывается в положении с координатами (100, 0) — что соответствует 3:00. Это делается с помощью формул:

$x' = x * cos(a) + y * sin(a)$
$y' = y * cos(a) — x * sin(a)$

Функция *RotatePoint* полезна, как мы вскоре увидим, для рисования как точек на циферблате, так и стрелок часов.

Функция *DrawClock* рисует 60 точек циферблата, начиная сверху (верхняя точка соответствует 12:00). Каждая из этих точек находится на расстоянии 900 единиц от начала координат, поэтому первая имеет координаты (0, 900), а каждая следующая отстоит от предыдущей на 6 градусов по часовой стрелке. У двенадцати точек диаметр равен 100 единицам; у остальных — диаметр 33 единицы. Точки рисуются с помощью функции *Ellipse*.

Функция *DrawHands* рисует часовую, минутную и секундную стрелки часов. Координаты, определяющие очертания стрелок (когда они находятся в вертикальном положении) хранятся в массиве структур POINT. В зависимости от времени, эти координаты вращаются с помощью функции *RotatePoint* и выводятся на экран с помощью функции Windows *PolyLine*. Обратите внимание, что часовая и минутная стрелки выводятся на экран только в том случае, если параметр *bChange* функции *DrawHands* равен TRUE. Когда программа обновляет стрелки часов, в большинстве случаев часовую и минутную стрелки перерисовывать не надо.

Теперь давайте обратим свое внимание на оконную процедуру. При обработке сообщения WM_CREATE оконная процедура получает текущее время, и сохраняет его в переменной *dtPrevious*. Эта переменная позже будет использоваться для определения, произошло ли изменение часов или минут с момента предыдущего обновления.

Первый раз часы рисуются при обработке первого сообщения WM_PAINT с помощью вызовов функций *SetIsotropic*, *DrawClock* и *DrawHands*. При вызове последней функции параметр *bChange* задается равным TRUE.

При обработке сообщения WM_TIMER *WndProc* сначала получает новое время и определяет необходимость перерисовки часовой и минутной стрелки. Если такая необходимость появляется, все стрелки рисуются белым пером, что стирает предыдущее положение стрелок. В противном случае белым пером стирается только секундная стрелка. Затем все стрелки рисуются черным пером.

Стандартное время Windows

Если вы просматривали руководства по функциям Windows, то могли бы удивиться, почему функция Windows *GetCurrentTime* не использовалась в программах DIDCLOCK и ANACLOCK. Ответ состоит в том, что функция *GetCurrentTime* сообщает вам "время Windows" (Windows time), а не реальное время. Отсчет времени (в миллисекундах) ведется с момента начала текущего сеанса работы Windows. Функция *GetCurrentTime* используется в основном для вычисления разницы со значением, возвращаемым функцией *GetMessageTime*. При

обработке сообщения вы можете использовать обе эти функции, чтобы определить, как долго сообщение находилось в очереди сообщений до того, как оно начало обрабатываться.

Анимация

В главе 4 мы изучили некоторые приемы использования битовых образов и контекстов памяти. Поскольку вывод на экран небольших битовых образов происходит достаточно быстро, вы можете использовать битовые образы (в сочетании с таймером Windows) для элементарной анимации.

Теперь самое время рассмотреть программу со скачущим мячом. В программе BOUNCE, приведенной на рис. 7.9, создается мяч, который скачет по рабочей области окна. Для задания темпа скачков мяча в программе используется таймер. Сам мяч является битовым образом. Сначала программа создает мяч путем создания битового образа, который она выбирает в контекст памяти, а затем вызывает простые функции GDI. Программа рисует растровый мяч на экране путем передачи блока битов из другого контекста памяти.

BOUNCE.MAK

```
#----------------------
# BOUNCE.MAK make file
#----------------------

bounce.exe : bounce.obj
     $(LINKER) $(GUIFLAGS) -OUT:bounce.exe bounce.obj $(GUILIBS)

bounce.obj : bounce.c
     $(CC) $(CFLAGS) bounce.c
```

BOUNCE.C

```
/*---------------------------------------
   BOUNCE.C -- Bouncing Ball Program
              (c) Charles Petzold, 1996
  ---------------------------------------*/

#include <windows.h>
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Bounce";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Bouncing Ball",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);
```

```
     if(!SetTimer(hwnd, 1, 50, NULL))
          {
          MessageBox(hwnd, "Too many clocks or timers!",
                      szAppName, MB_ICONEXCLAMATION | MB_OK);
          return FALSE;
          }

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HBITMAP hBitmap;
     static int     cxClient, cyClient, xCenter, yCenter, cxTotal, cyTotal,
                    cxRadius, cyRadius, cxMove, cyMove, xPixel, yPixel;
     HBRUSH         hBrush;
     HDC            hdc, hdcMem;
     int            iScale;

     switch(iMsg)
          {
          case WM_CREATE :
               hdc = GetDC(hwnd);
               xPixel = GetDeviceCaps(hdc, ASPECTX);
               yPixel = GetDeviceCaps(hdc, ASPECTY);
               ReleaseDC(hwnd, hdc);
               return 0;

          case WM_SIZE :
               xCenter =(cxClient = LOWORD(lParam)) / 2;
               yCenter =(cyClient = HIWORD(lParam)) / 2;

               iScale = min(cxClient * xPixel, cyClient * yPixel) / 16;

               cxRadius = iScale / xPixel;
               cyRadius = iScale / yPixel;

               cxMove = max(1, cxRadius / 2);
               cyMove = max(1, cyRadius / 2);

               cxTotal = 2 *(cxRadius + cxMove);
               cyTotal = 2 *(cyRadius + cyMove);

               if(hBitmap)
                    DeleteObject(hBitmap);

               hdc = GetDC(hwnd);
               hdcMem = CreateCompatibleDC(hdc);
               hBitmap = CreateCompatibleBitmap(hdc, cxTotal, cyTotal);
               ReleaseDC(hwnd, hdc);

               SelectObject(hdcMem, hBitmap);
               Rectangle(hdcMem, -1, -1, cxTotal + 1, cyTotal + 1);

               hBrush = CreateHatchBrush(HS_DIAGCROSS, 0L);
               SelectObject(hdcMem, hBrush);
```

```
               SetBkColor(hdcMem, RGB(255, 0, 255));
               Ellipse(hdcMem, cxMove, cyMove, cxTotal - cxMove,
                                                cyTotal - cyMove);
               DeleteDC(hdcMem);
               DeleteObject(hBrush);
               return 0;
          case WM_TIMER :
               if(!hBitmap)
                    break;

               hdc = GetDC(hwnd);
               hdcMem = CreateCompatibleDC(hdc);
               SelectObject(hdcMem, hBitmap);

               BitBlt(hdc, xCenter - cxTotal / 2,
                           yCenter - cyTotal / 2, cxTotal, cyTotal,
                       hdcMem, 0, 0, SRCCOPY);

               ReleaseDC(hwnd, hdc);
               DeleteDC(hdcMem);

               xCenter += cxMove;
               yCenter += cyMove;

               if((xCenter + cxRadius >= cxClient) ||
                  (xCenter - cxRadius <= 0))
                        cxMove = -cxMove;

               if((yCenter + cyRadius >= cyClient) ||
                  (yCenter - cyRadius <= 0))
                        cyMove = -cyMove;
               return 0;

          case WM_DESTROY :
               if(hBitmap)
                    DeleteObject(hBitmap);

               KillTimer(hwnd, 1);
               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

Рис. 7.9 Программа BOUNCE

При каждом получении сообщения WM_SIZE программа BOUNCE удаляет, а затем снова создает мяч. Для этого необходим совместимый с экраном контекст памяти:

```
hdcMem = CreateCompatibleDC(hdc);
```

Диаметр мяча равен одной шестнадцатой минимума из высоты и ширины рабочей области. Однако, программа строит растровый образ, который больше мяча: с каждой из его четырех сторон, размеры растрового образа превосходят размеры мяча на половину радиуса мяча:

```
hbitmap = CreateCompatibleBitmap(hdc, cxTotal, cyTotal);
```

После того, как битовый образ выбран в контекст памяти, весь его фон закрашивается белым цветом:

```
Rectangle(hdcMem, -1, -1, xTotal + 1, yTotal +1);
```

В контекст памяти выбирается кисть с диагональной штриховкой, и в центре растрового образа рисуется мяч:

```
Ellipse(hdcMem, xMove, yMove, xTotal — xMove, yTotal — yMove);
```

Поле вокруг границ мяча стирает предыдущий образ мяча при его движении. Перерисовка мяча в другом положении требует просто вызова функции *BitBlt*, в котором используется ROP код SRCCOPY:

```
BitBlt(hdc, xCenter — xTotal / 2, yCenter — yTotal / 2, xTotal, yTotal, hdcMem, 0, 0, SRCCOPY);
```

Программа BOUNCE демонстрирует простейший способ перемещения образа по экрану, но для серьезных задач такой подход неудовлетворителен. Если вам интересна анимация, вам понадобится изучить несколько других ROP кодов (например, SRCINVERT), который выполняет операцию исключающего OR для источника и приемника. Другие приемы анимации состоят в использовании палитры Windows (включая функцию *AnimatePalette*) и функции *CreateDIBSection*.

Глава 8 Дочерние окна управления

8

В главе 6 были представлены программы из серии CHECKER, которые выводили на экран сетку из прямоугольников. Когда делается щелчок мышью в прямоугольнике, программа рисует символ зачеркивания X. При повторном щелчке символ зачеркивания исчезает. Если в первых двух версиях этой программы, CHECKER1 и CHECKER2, используется только одно главное окно, то уже в версии CHECKER3 для каждого прямоугольника используется свое собственное дочернее окно. Прямоугольники обслуживаются отдельной оконной процедурой *ChildWndProc*.

При желании мы могли бы добавить в процедуру *ChildWndProc* средство для посылки сообщения своей родительской оконной процедуре (*WndProc*), вне зависимости от того, помечается прямоугольник или пометка убирается. Это делается так: дочерняя оконная процедура может определить описатель родительского окна с помощью вызова функции *GetParent*:

```
hwndParent = GetParent(hwnd);
```

где *hwnd* — это описатель дочернего окна. Теперь можно послать сообщение родительской оконной процедуре:

```
SendMessage(hwndParent, iMsg, wParam, lParam);
```

Чему мог бы быть равен параметр *iMsg*? Всему, чему угодно, но конечно в пределах численного диапазона от WM_USER до 0X7FFF. Эти числа представляют сообщения, которые не конфликтуют с уже предопределенными сообщениями типа WM_. Возможно, для такого сообщения дочернее окно могло бы установить значение *wParam* равным идентификатору этого дочернего окна. Тогда *lParam* мог бы устанавливаться в 1 при пометке дочернего окна, и в 0 при снятии пометки. Это одна из возможностей.

В результате создается "дочернее окно управления" (child window control). Дочернее окно обрабатывает сообщения мыши и клавиатуры и извещает родительское окно о том, что состояние дочернего окна изменилось. В этом случае дочернее окно становится для родительского окна устройством ввода. Оно инкапсулирует особые действия, связанные с графическим представлением окна на экране, реакцией на пользовательский ввод, и извещения другого окна при вводе важной информации.

Можно создавать свои собственные дочерние окна управления, но есть также возможность использовать преимущества нескольких уже определенных классов окна (и оконных процедур), с помощью которых ваша программа может создавать стандартные дочерние окна управления, которые вы, несомненно, уже наблюдали в других программах для Windows. Такие дочерние окна имеют вид кнопок (buttons), флажков (check boxes), окон редактирования (edit boxes), списков (list boxes), комбинированных списков (combo boxes), строк текста (text strings) и полос прокрутки (scroll bars). Например, если есть необходимость иметь кнопку с надписью "Recalculate" в углу вашей программы электронных таблиц, то можно создать ее с помощью одного вызова функции *CreateWindow*. Вам нет нужды беспокоиться о логике обработки мыши, или о логике рисования кнопок, или о том, чтобы кнопка при щелчке на ней мыши "нажималась". Все это делается в Windows. Все, что остается делать — это обрабатывать сообщения WM_COMMAND, которыми кнопка информирует вашу оконную процедуру о том, что она была нажата.

Действительно ли это так просто? Да, почти.

Дочерние окна управления наиболее часто используются в окнах диалога. Как будет видно в главе 11, положение и размер дочерних окон управления определяются в шаблоне окон диалога, который хранится в описании ресурсов программы. Вы также можете пользоваться предопределенными дочерними окнами управления на поверхности рабочей области обычного окна. Каждое дочернее окно создается с помощью вызова функции *CreateWindow*, где с помощью функции *MoveWindow* задается его положение и размер. Оконная процедура родительского окна

посылает сообщения дочерним окнам управления, а дочерние окна управления посылают сообщения обратно оконной процедуре.

Для создания обычного окна приложения, во-первых, определите класс окна и зарегистрируйте его в Windows с помощью функции *RegisterClassEx*. Затем с помощью функции *CreateWindow* создайте окно на основе этого класса. Однако, если вы используете одно из предопределенных дочерних окон управления, то для этого дочернего окна класс окна регистрировать не надо. Такой класс уже существует в Windows и имеет одно из следующих имен: "button" (кнопка), "static" (статическое), "scrollbar" (полоса прокрутки), "edit" (окно редактирования), "listbox" (окно списка) или "combobox" (окно комбинированного списка). Вы просто используете имя в качестве параметра класса окна в функции *CreateWindow*. Параметр стиля окна функции *CreateWindow* более точно определяет вид и свойства дочернего окна управления. Windows включает в себя оконные процедуры, обрабатывающие сообщения тех дочерних окон, которые созданы на основе перечисленных классов.

Использование дочерних окон управления прямо на поверхности окна требует решения задач более нижнего уровня, чем те, которые необходимо решить при работе с дочерними окнами управления в окнах диалога, где диспетчер окна диалога добавляет уровень, изолирующий вашу программу от собственно окон управления. В частности, вы узнаете, что у дочерних окон управления, которые вы создаете на поверхности вашего окна, нет встроенной поддержки перемещения фокуса ввода с одного дочернего окна управления на другое при помощи клавиши <Tab> или клавиш управления курсором. Дочернее окно управления может получить фокус ввода, но после того, как это сделано, оно уже не может легко вернуть его обратно родительскому окну. С этой проблемой мы будем бороться на протяжении всей главы.

Класс кнопок

Наше изучение класса окна button мы начнем с программы BTNLOOK "вид кнопки" (button look), которая представлена на рис. 8.1. В программе BTNLOOK создается 10 дочерних окон управления в виде кнопок, по одному на каждый из 10 стандартных стилей кнопок.

BTNLOOK.MAK

```
#-----------------------
# BTNLOOK.MAK make file
#-----------------------

btnlook.exe : btnlook.obj
     $(LINKER) $(GUIFLAGS) -OUT:btnlook.exe btnlook.obj $(GUILIBS)

btnlook.obj : btnlook.c
     $(CC) $(CFLAGS) btnlook.c
```

BTNLOOK.C

```
/*----------------------------------------
   BTNLOOK.C -- Button Look Program
               (c) Charles Petzold, 1996
  ----------------------------------------*/

#include <windows.h>

struct
     {
     long style;
     char *text;
     }
     button[] =
     {
     BS_PUSHBUTTON,      "PUSHBUTTON",
     BS_DEFPUSHBUTTON,   "DEFPUSHBUTTON",
     BS_CHECKBOX,        "CHECKBOX",
     BS_AUTOCHECKBOX,    "AUTOCHECKBOX",
     BS_RADIOBUTTON,     "RADIOBUTTON",
     BS_3STATE,          "3STATE",
     BS_AUTO3STATE,      "AUTO3STATE",
     BS_GROUPBOX,        "GROUPBOX",
     BS_AUTORADIOBUTTON, "AUTORADIO",
     BS_OWNERDRAW,       "OWNERDRAW"
```

```
     };

#define NUM(sizeof button / sizeof button[0])

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "BtnLook";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Button Look",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

     while(GetMessage(&msg, NULL, 0, 0))
          {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
          }
     return msg.wParam;
     }
LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char  szTop[]    = "iMsg            wParam       lParam",
                  szUnd[]    = "____            ______       ______",
                  szFormat[] = "%-16s%04X-%04X    %04X-%04X",
                  szBuffer[50];
     static HWND  hwndButton[NUM];
     static RECT  rect;
     static int   cxChar, cyChar;
     HDC          hdc;
     PAINTSTRUCT  ps;
     int          i;
     TEXTMETRIC   tm;

     switch(iMsg)
          {
          case WM_CREATE :
               hdc = GetDC(hwnd);
               SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));
```

```
            GetTextMetrics(hdc, &tm);
            cxChar = tm.tmAveCharWidth;
            cyChar = tm.tmHeight + tm.tmExternalLeading;
            ReleaseDC(hwnd, hdc);

            for(i = 0; i < NUM; i++)
                 hwndButton[i] = CreateWindow("button", button[i].text,
                           WS_CHILD | WS_VISIBLE | button[i].style,
                           cxChar, cyChar *(1 + 2 * i),
                           20 * cxChar, 7 * cyChar / 4,
                           hwnd,(HMENU) i,
                          ((LPCREATESTRUCT) lParam) -> hInstance, NULL);
            return 0;

       case WM_SIZE :
            rect.left   = 24 * cxChar;
            rect.top    =  2 * cyChar;
            rect.right  = LOWORD(lParam);
            rect.bottom = HIWORD(lParam);
            return 0;

       case WM_PAINT :
            InvalidateRect(hwnd, &rect, TRUE);

            hdc = BeginPaint(hwnd, &ps);
            SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));
            SetBkMode(hdc, TRANSPARENT);

            TextOut(hdc, 24 * cxChar, cyChar, szTop, sizeof(szTop) - 1);
            TextOut(hdc, 24 * cxChar, cyChar, szUnd, sizeof(szUnd) - 1);

            EndPaint(hwnd, &ps);
            return 0;
       case WM_DRAWITEM :
       case WM_COMMAND :
            ScrollWindow(hwnd, 0, -cyChar, &rect, &rect);

            hdc = GetDC(hwnd);
            SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));

            TextOut(hdc, 24 * cxChar, cyChar *(rect.bottom / cyChar - 1),
                     szBuffer,
                     wsprintf(szBuffer, szFormat,
                     iMsg == WM_DRAWITEM ? "WM_DRAWITEM" : "WM_COMMAND",
                     HIWORD(wParam), LOWORD(wParam),
                     HIWORD(lParam), LOWORD(lParam)));

            ReleaseDC(hwnd, hdc);
            ValidateRect(hwnd, &rect);

            break;

       case WM_DESTROY :
            PostQuitMessage(0);
            return 0;
       }
  return DefWindowProc(hwnd, iMsg, wParam, lParam);
  }
```

Рис. 8.1 Программа BTNLOOK

При щелчке на любой из кнопок она посылает сообщение WM_COMMAND оконной процедуре родительского окна *WndProc*. *WndProc* программы BTNLOOK выводит на экран параметры *wParam* и *lParam* этого сообщения в правой половине рабочей области, как показано на рис. 8.2.

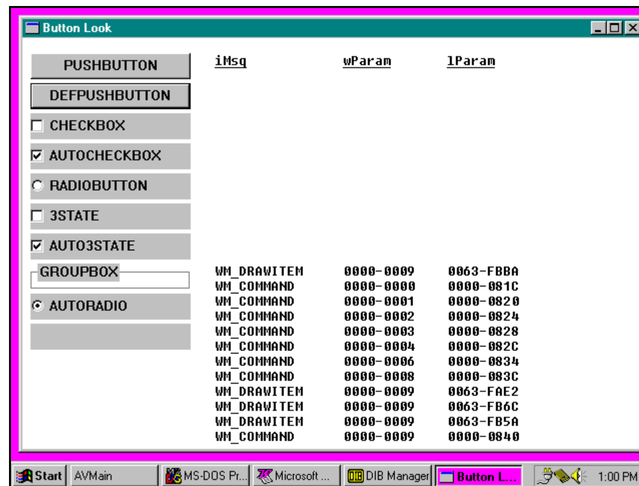


Рис. 8.2 Вывод на экран программы BTNLOOK

Кнопка со стилем BS_OWNERDRAW выводится в окне только в виде затенения фона, поскольку стиль этой кнопки рассчитан на то, что за рисование отвечает сама программа. Эта кнопка показывает, что ее необходимо рисовать при обработке сообщения WM_DRAWITEM с параметром *lParam*, являющимся указателем на структуру типа DRAWITEMSTRUCT. Эти сообщения также выводятся на экран в программе BTNLOOK. (Более подробно о кнопках, определяемых пользователем, будет рассказано далее в этой главе.)

Создание дочерних окон

В программе BTNLOOK определяется структура с именем *button*, в которой содержатся стили окон кнопок и строки текста для каждой из 10 типов кнопок. Все стили окон кнопок начинаются с префикса BS, что означает "button style" (стиль кнопки).

Десять дочерних окон в виде кнопок создаются в цикле *for* при обработке сообщения WM_CREATE в *WndProc*. При вызове функции *CreateWindow* используются следующие параметры:

Имя класса	`"button"`
Текст окна	`button[i].text`
Стиль окна	`WS_CHILD | WS_VISIBLE | button[i].style`
Положение по *x*	`cxChar`
Положение по *y*	`cyChar * (1 + 2 * i)`
Ширина	`20 * cxChar`
Высота	`7 * cyChar / 4`
Родительское окно	`hwnd`
Идентификатор дочернего окна	`(HMENU)i`
Описатель экземпляра	`((LPCREATESTRUCT) lParam) -> hInstance`
Дополнительные параметры	`NULL`

Параметр "имя класса" — его предопределенное имя. При задании стиля окна используются стили WS_CHILD, WS_VISIBLE и один из десяти стилей кнопок (BS_PUSHBUTTON, BS_DEFPUSHBUTTON и т. д.), которые определены в структуре *button*. Параметр "текст окна" (который у обычного окна появляется в строке заголовка) — это текст, который будет выводиться вместе с каждой кнопкой. Здесь просто использовался текст, идентифицирующий стиль кнопки.

Параметры "положение по *x*" и "положение по *y*" показывают положение верхнего левого угла дочернего окна относительно верхнего левого угла рабочей области родительского окна. Параметры "ширина" и "высота" задают ширину и высоту каждого дочернего окна.

Параметр "идентификатор дочернего окна" должен быть для каждого окна уникальным. Этот идентификатор помогает оконной процедуре при обработке сообщений WM_COMMAND определить дочернее окно — источник сообщений. Обратите внимание, что идентификатор дочернего окна передается в функцию *CreateWindow* в виде параметра, который обычно используется для задания меню программы, поэтому его тип должен быть преобразован к HMENU.

Параметр "описатель экземпляра" функции *CreateWindow* выглядит несколько странно, но при этом во время обработки сообщения WM_CREATE параметр *lParam* фактически равен указателю на структуру типа CREATESTRUCT ("creation structure" — структура создания), членом которой является *hInstance*. Поэтому мы приводим тип параметра *lParam* к типу указатель на структуру CREATESTRUCT и извлекаем *hInstance*.

(Некоторые программы для Windows используют глобальную переменную *hInst* для того, чтобы обеспечить доступ оконной процедуры к описателю экземпляра, который находится в *WinMain*. В *WinMain* перед созданием главного окна нужно просто написать:

```
hInst = hInstance;
```

В программе CHECKER в главе 6 мы использовали функцию *GetWindowLong* для получения описателя экземпляра:

```
GetWindowLong(hwnd, GWL_HINSTANCE)
```

Можно использовать любой из этих методов.)

После вызова функции *CreateWindow* нам больше ничего не нужно делать с этими дочерними окнами. Оконная процедура кнопки внутри Windows поддерживает эти кнопки и управляет всеми процессами перерисовки. (За исключением кнопки стиля BS_OWNERDRAW; как позже будет рассказано, этот стиль кнопки требует программы для рисования кнопки.) При завершении программы, когда удаляется родительское окно, Windows удаляет и дочерние окна.

Сообщения дочерних окон родительскому окну

Когда вы запускаете программу BTNLOOK, вы видите различные типы кнопок, выводимые на экран в левой части рабочей области. Как уже говорилось, когда вы щелкаете мышью на кнопке, дочернее окно управления посылает сообщение WM_COMMAND своему родительскому окну. Программа BTNLOOK обрабатывает сообщение WM_COMMAND и выводит на экране значения параметров *wParam* и *lParam*. Здесь приведен их смысл:

LOWORD (*wParam*)	Идентификатор дочернего окна
HIWORD (*wParam*)	Код уведомления
lParam	Описатель дочернего окна

Если вы модифицируете программы, написанные для 16-разрядных версий Windows, то знайте, что эти три параметра сообщения были изменены с учетом того, что описатели стали 32-разрядными.

Идентификатор дочернего окна — это значение, передаваемое функции *CreateWindow*, когда создается дочернее окно. В программе BTNLOOK этими идентификаторами являются значения от 0 до 9 для 10 выводимых в рабочую область кнопок. Описатель дочернего окна — это значение, которое Windows возвращает при вызове функции *CreateWindow*.

Код уведомления — это дополнительный код, который дочернее окно использует для того, чтобы сообщить родительскому окну более точные сведения о сообщении. Возможные значения кодов уведомления для кнопок определены в заголовочных файлах Windows:

Идентификатор кода уведомления кнопки	Значение
BN_CLICKED	0
BN_PAINT	1
BN_HILITE	2
BN_UNHILITE	3
BN_DISABLE	4
BN_DOUBLECLICKED	5

Коды уведомления от 1 до 5 — это коды для кнопок устаревшего стиля BS_USERBUTTON, поэтому вы столкнетесь только с кодами BN_CLICKED.

Обратите внимание, что при щелчке мышью текст кнопки обводится пунктирной линией. Это говорит о том, что кнопка имеет фокус ввода. Теперь весь ввод клавиатуры направлен на дочернее окно кнопки, а не на главное окно. Однако, если кнопка имеет фокус ввода, она игнорирует любые нажатия клавиш за исключением <Spacebar>, которая теперь оказывает то же действие, что и щелчок мыши.

Сообщения родительского окна дочерним окнам

Оконная процедура также может посылать сообщения дочернему окну управления, хотя в программе BTNLOOK и не отражен этот факт. Пять специальных сообщений для кнопок определены в заголовочных файлах Windows; каждое из которых начинается с префикса "BM", что означает "button message" (сообщение кнопки). Вот эти сообщения:

```
BM_GETCHECK
BM_SETCHECK
BM_GETSTATE
BM_SETSTATE
BM_SETSTYLE
```

Сообщения BM_GETCHECK и BM_SETCHECK посылаются родительским окном дочернему окну управления для установки и снятия контрольных меток флажков (check boxes) и переключателей (radio buttons). Сообщения BM_GETSTATE и BM_SETSTATE касаются обычного или "нажатого" состояния окна при щелчке мышью или нажатии клавиши <Spacebar>. (Мы рассмотрим как работают эти сообщения при изучении кнопки каждого типа.) Сообщение BM_SETSTYLE позволяет вам изменять стиль кнопки после ее создания.

Каждое дочернее окно имеет описатель окна и его идентификатор, который является уникальным среди других. Знание одного из этих элементов позволяет вам получить другой. Если вы знаете описатель дочернего окна, то можете получить его идентификатор:

```
id = GetWindowLong(hwndChild, GWL_ID);
```

Программа CHECKER3 в главе 6 показала, что окно может хранить данные в специальной области, зарезервированной при регистрации класса окна. Область, в которой хранится идентификатор дочернего окна резервируется операционной системой Windows при его создании. Вы можете также использовать функцию:

```
id = GetDlgCtrlID(hwndChild);
```

Хотя часть имени функции "Dlg" относится к окну диалога, на самом деле эта функция общего назначения.

Зная идентификатор, вы можете получить описатель дочернего окна:

```
hwndChild = GetDlgItem(hwndParent, id);
```

Нажимаемые кнопки

Первые две кнопки, представленные в программе BTNLOOK, являются "нажимаемыми" кнопками (push buttons). Каждая из этих кнопок представляет собой прямоугольник, внутри которого находится текст, заданный в параметре текста окна функции *CreateWindow*. Ширина и высота прямоугольника полностью определяется размерами, заданными в функциях *CreateWindow* или *MoveWindow*. Текст располагается в центре прямоугольника.

Нажимаемые кнопки управления используются в основном для запуска немедленного действия без сохранения какой бы то ни было индикации положения кнопки типа включено/выключено. Эти два типа нажимаемых кнопок управления имеют стили окна, которые называются BS_PUSHBUTTON и BS_DEFPUSHBUTTON. Строка "DEF" в BS_DEFPUSHBUTTON означает "по умолчанию — default". Если при создании окон диалога использовать кнопки BS_PUSHBUTTON и BS_DEFPUSHBUTTON, то их функционирование отличается друг от друга. Если же их использовать для создания дочерних окон управления, то эти два типа нажимаемых кнопок действуют одинаково, хотя кнопка BS_DEFPUSHBUTTON имеет более жирную рамку.

Нажимаемые кнопки выглядят лучше, если их высота составляет 7/4 высоты символа шрифта SYSTEM_FONT, который используется в программе BTNLOOK. Ширина нажимаемых кнопок должна, по крайней мере, соответствовать длине выводимого текста плюс два дополнительных символа.

Когда курсор мыши находится на нажимаемой кнопке, щелчок мышью заставит кнопку перерисовать саму себя, используя стиль 3D с тенью, чтобы выглядеть нажатой. Отпускание кнопки мыши восстанавливает начальный облик нажимаемой кнопки, а родительскому окну посылается сообщение WM_COMMAND с кодом уведомления BN_CLICKED. Как и тогда, когда дело касается кнопок других типов, если нажимаемая кнопка имеет фокус ввода, то текст обводится штриховой линией, а нажатие и отпускание клавиши <Spacebar> имеет тот же эффект, что и нажатие и отпускание кнопки мыши.

Вы можете имитировать нажатие кнопки, посылая окну сообщение WM_SETSTATE. Следующий оператор приводит к нажатию кнопки:

```
SendMessage(hwndButton, BM_SETSTATE, 1, 0);
```

Следующий вызов заставляет кнопку вернуться к своему нормальному состоянию:

```
SendMessage(hwndButton, BM_SETSTATE, 0, 0);
```

Описатель окна *hwndButton* является возвращаемым значением функции *CreateWindow*.

Вы также можете послать нажимаемой кнопке сообщение WM_GETSTATE. Дочерняя кнопка управления возвращает текущее состояние — TRUE, если кнопка нажата и FALSE (или 0), если она в обычном состоянии. Однако, для большинства приложений эта информация не требуется. И поскольку нажимаемая кнопка не сохраняет информацию о своем положении типа включено/выключено, сообщения BM_GETCHECK и BM_SETCHECK не используется.

Флажки

Флажки (check boxes) представляют из себя маленькие квадратные окна с текстом; текст обычно размещается справа от окна флажка. (Если при создании кнопки вы используете стиль BS_LEFTTEXT, то текст окажется слева.) В программах флажки обычно объединяются, что дает пользователю возможность установить опции. Флажки, как

правило, действуют как двухпозиционные переключатели: один щелчок вызывает появление контрольной метки (галочки); другой щелчок приводит к исчезновению контрольной метки (галочки).

Двумя наиболее используемыми стилями для флажков являются BS_CHECKBOX и BS_AUTOCHECKBOX. При использовании стиля BS_CHECKBOX вы должны сами устанавливать контрольную метку, посылая сообщение BM_SETCHECK. Параметр *wParam* устанавливается в 1 для установки контрольной метки и в 0 для ее удаления. Вы можете получить текущее состояние флажка, посылая управляющее сообщение BM_GETCHECK. Вы могли бы использовать следующие инструкции для переключения метки X при обработке сообщения WM_COMMAND:

```
SendMessage((HWND)lParam, BM_SETCHECK,(WPARAM)
!SendMessage((HWND)lParam, BM_GETCHECK, 0, 0),0);
```

Обратите внимание на операцию *!* перед вторым вызовом функции *SendMessage*. Значение параметра *lParam* является описателем дочернего окна, переданным в вашу оконную процедуру сообщением WM_COMMAND. Если вам позже понадобится узнать состояние кнопки, пошлите ей другое сообщение BM_GETCHECK. Вы можете также сохранять текущее состояние контрольной метки в статической переменной внутри вашей оконной процедуры. Вы можете также инициализировать флажок BS_CHECKBOX меткой X, посылая ему сообщение BM_SETCHECK:

```
SendMessage(hwndButton, BM_SETCHECK, 1, 0);
```

При стиле BS_AUTOCHECKBOX флажок сам включает или выключает контрольную метку. Ваша оконная процедура может игнорировать сообщения WM_COMMAND. Если вам необходимо текущее состояние кнопки, пошлите сообщение BM_GETCHECK:

```
iCheck =(int) SendMessage(hwndButton, BM_GETCHECK, 0, 0);
```

Значение *iCheck* равно TRUE (не равно 0), если кнопка помечена, FALSE (или 0), если нет.

Двумя другими стилями флажков являются BS_3STATE и BS_AUTO3STATE. Как показывают их имена, эти стили могут отображать третье состояние — серый цвет внутри окна флажка — которое имеет место, когда вы посылаете сообщение BM_SETCHECK с параметром *wParam* равным 2. Серый цвет показывает пользователю, что его выбор неопределен или не имеет отношения к делу. В этом случае флажок не может быть включен — т. е. он запрещает какой-либо выбор в данный момент. Однако, флажок продолжает посылать сообщения родительскому окну, если щелкать на нем мышью. Более удобные методы полного запрещения работы с флажком описаны дальше.

Окно флажка помещается в левой части и в центре относительно верхней и нижней сторон прямоугольника, который был задан при вызове функции *CreateWindow*. Щелчок мыши в любом месте внутри прямоугольника вызывает посылку родительскому окну сообщения WM_COMMAND. Минимальная высота флажка равна высоте символа. Минимальная ширина равна количеству символов в тексте плюс два.

Переключатели

Переключатели (radio buttons) похожи на флажки, но их форма не квадратная, а круглая. Жирная точка внутри кружка показывает, что переключатель помечен. Переключатель имеет стиль окна BS_RADIOBUTTON или BS_AUTORADIOBUTTON, но последний используется только в окнах диалога. В окнах диалога группы переключателей, как правило, используются для индикации нескольких взаимоисключающих опций. В отличие от флажков, если повторно щелкнуть на переключателе, его состояние не изменится.

При получении сообщения WM_COMMAND от переключателя, необходимо отобразить его отметку, отправив сообщение BM_SETCHECK с параметром *wParam*, равным 1:

```
SendMessage(hwndButton, BM_SETCHECK, 1, 0);
```

Для всех остальных переключателей этой группы можно отключить контрольную метку, послав сообщение BM_SETCHECK с параметром *wParam*, равным 0:

```
SendMessage(hwndButton, BM_SETCHECK, 0, 0);
```

Окна группы

Окно группы (group boxes) — стиль BS_GROUPBOX — является исключением в классе кнопок. Оно не обрабатывает ни сообщения от клавиатуры, ни сообщения от мыши, оно не посылает своему родительскому окну сообщений WM_COMMAND. Окно группы представляет собой прямоугольную рамку с текстом вверху. Окна групп часто используются для того, чтобы в них размещать другие кнопки управления.

Изменение текста кнопки

Вы можете изменить текст кнопки (или любого другого окна) с помощью вызова функции *SetWindowText*:

```
SetWindowText(hwnd, pszString);
```

где *hwnd* — это описатель окна, в котором изменяется текст, а *pszString* — это указатель на оканчивающуюся нулем строку. Для обычного окна этот текст — текст строки заголовка. Для кнопок управления — это текст, который выводится на экран вместе с кнопкой.

Вы также можете получить текущий текст окна:

```
iLength = GetWindowText(hwnd, pszBuffer, iMaxLength);
```

Параметр *iMaxLength* задает максимальное число символов для копирования в буфер, который определяется указателем *pszBuffer*. Возвращаемым значением функции является длина скопированной строки. Вы можете подготовить вашу программу для приема строки конкретной длины, вызвав сначала функцию:

```
iLength = GetWindowTextLength(hwnd);
```

Видимые и доступные кнопки

Для получения ввода от мыши и от клавиатуры дочернее окно должно быть одновременно видимым (отображенным на экране) и доступным (разрешенным) для ввода. Если дочернее окно является видимым, но недоступным, Windows выводит на экран текст окна не черным, а серым цветом.

Если при создании дочернего окна, вы не включили в класс окна идентификатор WS_VISIBLE, то дочернее окно не появится на экране до тех пор, пока вы не вызовете функцию *ShowWindow*:

```
ShowWindow(hwndChild, SW_SHOWNORMAL);
```

Если вы включили в класс окна идентификатор WS_VISIBLE, то вам нет необходимости вызывать функцию *ShowWindow*. Однако, с помощью вызова этой функции можно скрыть дочернее окно:

```
ShowWindow(hwndChild, SW_HIDE);
```

Определить, является ли дочернее окно видимым, можно, вызвав функцию:

```
IsWindowVisible(hwndChild);
```

Вы также можете сделать дочернее окно доступным или недоступным для ввода. По умолчанию дочернее окно доступно. Вы можете сделать его недоступным с помощью вызова функции:

```
EnableWindow(hwndChild, FALSE);
```

Для кнопок этот вызов приводит к изображению текстовой строки кнопки серым цветом. Кнопка перестает реагировать на ввод с клавиатуры и мыши. Это лучший способ продемонстрировать, что какая-то опция, соответствующая кнопке, в данный момент недоступна.

Вы можете вновь сделать дочернее окно доступным, вызвав функцию:

```
EnableWindow(hwndChild, TRUE);
```

Определить, доступно или нет дочернее окно, можно с помощью функции:

```
IsWindowEnabled(hwndChild);
```

Кнопки и фокус ввода

Как уже упоминалось в этой главе, нажимаемые кнопки, флажки, переключатели и кнопки, определяемые пользователем, получают фокус ввода при щелчке мыши на них. Признаком наличия фокуса ввода служит окружающая текст пунктирная линия. Когда дочерние окна управления получают фокус ввода, родительское окно теряет его; весь ввод с клавиатуры направлен теперь не на родительское окно, а на дочернее окно управления. Однако, дочернее окно управления реагирует только на клавишу <Spacebar>, которая в этот момент действует аналогично мыши. Такая ситуация создает очевидную проблему: ваша программа теряет контроль над обработкой сообщений клавиатуры. Давайте посмотрим, что можно с этим сделать.

Как говорилось в главе 5, когда Windows переключает фокус ввода с одного окна (например, родительского) на другое (например, дочернее окно управления), она первым делом посылает сообщение WM_KILLFOCUS окну, теряющему фокус ввода. Параметр сообщения *wParam* является описателем окна, которое должно получить фокус ввода. Затем Windows посылает сообщение WM_SETFOCUS окну, получающему фокус ввода, при этом параметр сообщения *wParam* является описателем окна, которое теряет фокус ввода. (В обоих случаях, *wParam* может быть равен NULL, который показывает, что нет окна, которое имеет или получает фокус ввода.)

Родительское окно, обрабатывая сообщения WM_KILLFOCUS, может предотвратить получение фокуса ввода дочерним окном. Предположим, что массив *hwndChild* содержит описатели всех дочерних окон. (Которые были помещены в массив при создании окон с помощью вызовов функций *CreateWindow*.) Пусть NUM — это число дочерних окон, тогда:

```
case WM_KILLFOCUS:
       for(i = 0; i < NUM; i++)
              if(hwndChild[ i ] ==(HWND) wParam)
              {
                     SetFocus(hwnd);
                     break;
              }
       return 0;
```

Этот фрагмент кода показывает, что, если родительское окно определяет, что его фокус ввода переходит к одному из дочерних окон управления, оно вызывает функцию *SetFocus* и восстанавливает фокус ввода на себя.

Далее представлен более простой (но менее очевидный) способ добиться того же самого:

```
case WM_KILLFOCUS:
       if(hwnd == GetParent((HWND) wParam))
              SetFocus(hwnd);
       return 0;
```

Однако, оба эти метода имеют недостатки: они не дают кнопкам возможности реагировать на клавишу <Spacebar>, поскольку кнопки никогда не получают фокус ввода. Лучше было бы дать кнопкам возможность получить фокус ввода, но при этом и пользователю обеспечить возможность переходить от кнопки к кнопке с помощью клавиши <Tab>. На первый взгляд это кажется невозможным, но далее будет показано, как это сделать с помощью приема названного "window subclassing" (установка новой оконной процедуры) в программе COLORS1, представленной далее в этой главе.

Дочерние окна управления и цвета

Как вы можете видеть на рис. 8.2, показанные здесь несколько кнопок выглядят не слишком привлекательно. Нажимаемые кнопки смотрятся неплохо, но остальные нарисованы в виде серого прямоугольника, которого здесь просто не должно быть. Так происходит потому, что кнопки предназначены для вывода на экран в окнах диалога, а окна диалога в Windows 95 имеют серую поверхность. Поверхность нашего окна белая, поскольку так мы определили ее в структуре WNDCLASS:

```
wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
```

Мы это делали, поскольку часто занимались выводом текста в рабочую область, а GDI использует цвет текста и фона, которые определяются в задаваемом по умолчанию контексте устройства. Этими цветами всегда являются белый и черный. Чтобы кнопки выглядели привлекательней, мы должны либо изменить цвет рабочей области, согласовав его с цветом фона кнопок, либо как-то изменить цвет фона кнопок на белый.

Вначале необходимо понять, как Windows использует системные цвета (system colors).

Системные цвета

Windows поддерживает 25 системных цветов, предназначенных для рисования различных элементов экрана. Вы можете получить и установить текущие значения этих цветов с помощью функций *GetSysColor* и *SetSysColors*. Идентификаторы, определенные в заголовочных файлах Windows, задают системный цвет. Установка системного цвета с помощью функции *SetSysColors* меняет его только на время текущего сеанса работы Windows.

Вы можете изменить некоторые (но не все) системные цвета, используя раздел Display окна Control Panel, или модифицируя секцию [colors] файла WIN.INI. Секция [colors] использует ключевые слова (отличные от идентификаторов функций *GetSysColor* и *SetSysColors*) для 25 системных цветов, которым соответствуют тройка чисел в диапазоне от 0 до 255, представляющая значения красного, зеленого и голубого. Следующая таблица показывает, как 25 системных цветов идентифицируются с помощью констант, используемых в функциях *GetSysColor* и *SetSysColors*, а также с помощью ключевых слов файла WIN.INI. Таблица построена последовательно в порядке возрастания значений констант COLOR_ от 0 до 24:

GetSysColor и SetSysColors	WIN.INI
COLOR_SCROLLBAR	Scrollbar
COLOR_BACKGROUND	Background
COLOR_ACTIVECAPTION	ActiveTitle
COLOR_INACTIVECAPTION	InactiveTitle
COLOR_MENU	Menu
COLOR_WINDOW	Window
COLOR_WINDOWFRAME	WindowFrame
COLOR_MENUTEXT	MenuText
COLOR_WINDOWTEXT	WindowText
COLOR_CAPTIONTEXT	TitleText

COLOR_ACTIVEBORDER	ActiveBorder
COLOR_INACTIVEBORDER	InactiveBorder
COLOR_APPWORKSPACE	AppWorkspace
COLOR_HIGHLIGHT	Hilight
COLOR_HIGHLIGHTTEXT	HilightText
COLOR_BTNFACE	ButtonFace
COLOR_BTNSHADOW	ButtonShadow
COLOR_GRAYTEXT	GrayText
COLOR_BTNTEXT	ButtonText
COLOR_INACTIVECAPTIONTEXT	InactiveTitleText
COLOR_BTNHIGHLIGHT	ButtonHilight
COLOR_3DDKSHADOW	ButtonDkShadow
COLOR_3DLIGHT	ButtonLight
COLOR_INFOTEXT	InfoText
COLOR_INFOBK	InfoWindow

Задаваемые по умолчанию значения этих 25 цветов обеспечиваются драйвером дисплея. Windows использует эти задаваемые по умолчанию значения до тех пор, пока они не заменяются значениями из секции [colors] файла WIN.INI, которые могут быть изменены через Control Panel.

Теперь плохие новости: хотя смысл многих из этих цветов кажется очевидным (например, COLOR_BACKGROUND — это цвет фона области desktop), использование системных цветов в Windows 95 весьма хаотично. Первые версии Windows выглядели намного проще, чем сейчас. Действительно, до появления Windows 3.0 определялись только первые 13 из приведенных выше системных цветов. В связи с возросшим применением визуально более сложных дочерних окон управления, использующих трехмерные изображения, понадобилось увеличить количество системных цветов.

Цвета кнопок

Эта проблема особенно очевидна для кнопок: цвет COLOR_BTNFACE является основным цветом поверхности нажимаемых кнопок и цветом фона остальных. (Этот же цвет используется в окнах диалога и сообщений.) Цвет COLOR_BTNSHADOW предназначен для затенения правой и нижней сторон нажимаемых кнопок, для внутренней поверхности квадратов флажков и кружков переключателей. Для нажимаемых кнопок COLOR_BTNTEXT используется как цвет текста, а для других кнопок таким цветом является COLOR_WINDOWTEXT. Несколько других системных цветов также используются для различного оформления кнопок.

Поэтому, если мы хотим изображать кнопки на поверхности нашей рабочей области, то один из способов избежать конфликта цветов, это настроиться на системные цвета. Для начала, при определении класса окна для фона рабочей области вашего окна используйте COLOR_BTNFACE:

```
wndclass.hbrBackground =(HBRUSH)(COLOR_BTNFACE + 1);
```

Вы можете проверить это в программе BTNLOOK. Windows понимает, что если значение *hbrBackground* структуры WNDCLASSEX такое низкое, то оно фактически ссылается на системный цвет, а не на реальный описатель. Windows требует, чтобы вы при использовании этих идентификаторов прибавляли 1, когда задаете их в поле *hbrBackground* структуры WNDCLASSEX, только для того, чтобы это значение не стало равным NULL. Если окажется, что системный цвет при выполнении программы изменяется, тогда поверхность вашей рабочей области станет недействительной, и Windows для ее обновления будет использовать новое значение COLOR_BTNFACE.

Но теперь возникла новая проблема. Если вы выводите текст на экран, используя функцию *TextOut*, то Windows для цвета фона текста (цвет, которым стирается фон позади текста) и цвета самого текста использует значения, определенные в контексте устройства. Задаваемыми по умолчанию значениями являются белый (фон) и черный (текст) вне зависимости от системных цветов и от поля *hbrBackground* структуры класса окна. Поэтому, для изменения цвета текста и его фона на системные цвета, вам нужно использовать функции *SetTextColor* и *SetBkColor*. Делайте это после получения описателя контекста устройства:

```
SetBkColor(hdc, GetSysColor(COLOR_BTNFACE));
SetTextColor(hdc, GetSysColor(COLOR_WINDOWTEXT));
```

Теперь цвет фона рабочей области, цвет фона текста и цвет самого текста — все вместе приведены в соответствие с цветом кнопок.

Сообщение WM_CTLCOLORBTN

Мы рассмотрели, как привести в соответствие цвет своей рабочей области и цвет текста к цвету фона кнопок. А можно ли настроить цвета кнопок так, чтобы они соответствовали цветам, которые вы предпочитаете видеть в своей программе? Теоретически можно, но практически нет.

Что вы, вероятно, не захотите делать, так это использовать функцию *SetSysColor* для изменения цветов изображения кнопок. Это повлияет на все работающие в данный момент под Windows программы; немногие пользователи одобрили бы такой подход.

Лучшим подходом (опять же теоретически) могла бы стать обработка сообщения WM_CTLCOLORBTN. Это сообщение, которое кнопка управления посылает оконной процедуре родительского окна, когда дочернее окно собирается рисовать свою рабочую область. Родительское окно на основании этого сообщения может менять цвета, которые будет использовать оконная процедура дочернего окна при рисовании. (В 16-разрядной версии Windows для всех органов управления применялось сообщение WM_CTLCOLOR. Оно было заменено отдельными сообщениями для каждого типа стандартных дочерних окон управления.)

Когда оконная процедура родительского окна получает сообщение WM_CTLCOLORBTN, то параметр *wParam* этого сообщения является описателем контекста устройства кнопки, а параметр *lParam* — описателем окна кнопки. К тому времени, когда оконная процедура родительского окна получает это сообщение, кнопка управления уже получила свой контекст устройства. При обработке сообщения WM_CTLCOLORBTH в вашей оконной процедуре, вы:

- Необязательно устанавливаете цвет текста с помощью функции *SetTextColor*.
- Необязательно устанавливаете цвет фона текста с помощью функции *SetBkColor*.
- Возвращаете описатель кисти дочернему окну.

Теоретически, дочернее окно использует кисть для рисования фона. Вы должны удалить кисть, когда она становится ненужной.

Здесь тоже возникает проблема с сообщением WM_CTLCOLORBTN: только нажимаемые кнопки и кнопки, определяемые пользователем посылают своему родительскому окну сообщение WM_CTLCOLORBTN. Кроме того, только кнопки, определяемые пользователем, реагируют на обработку сообщения родительским окном, используя кисть для закрашивания фона. А это совершенно бесполезно, поскольку за рисование кнопок, определяемых пользователем и так всегда отвечает родительское окно.

Позже в этой главе мы рассмотрим случаи, когда сообщения, похожие на WM_CTLCOLORBTN, но используемые для других типов дочерних окон управления, оказываются более полезными.

Кнопки, определяемые пользователем

Если вы хотите полностью управлять внешним обликом кнопки, но не хотите связываться с логикой обработки клавиатуры и мыши, вы можете создать кнопку стиля BS_OWNERDRAW, как показано в программе OWNERDRW, приведенной на рис. 8.3.

OWNERDRW.MAK

```
#------------------------
# OWNERDRW.MAK make file
#------------------------
ownerdrw.exe : ownerdrw.obj
     $(LINKER) $(GUIFLAGS) -OUT:ownerdrw.exe ownerdrw.obj $(GUILIBS)

ownerdrw.obj : ownerdrw.c
     $(CC) $(CFLAGS) ownerdrw.c
```

OWNERDRW.C

```
/*----------------------------------------------
   OWNERDRW.C -- Owner-Draw Button Demo Program
                 (c) Charles Petzold, 1996
  ----------------------------------------------*/

#include <windows.h>

#define IDC_SMALLER      1
#define IDC_LARGER       2

#define BTN_WIDTH       (8 * cxChar)
#define BTN_HEIGHT      (4 * cyChar)

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

HINSTANCE hInst;
```

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "OwnerDrw";
     MSG         msg;
     HWND        hwnd;
     WNDCLASSEX  wndclass;

     hInst = hInstance;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = szAppName;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Owner-Draw Button Demo",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

void Triangle(HDC hdc, POINT pt[])
     {
     SelectObject(hdc, GetStockObject(BLACK_BRUSH));
     Polygon(hdc, pt, 3);
     SelectObject(hdc, GetStockObject(WHITE_BRUSH));
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HWND      hwndSmaller, hwndLarger;
     static int       cxClient, cyClient, cxChar, cyChar;
     int              cx, cy;
     LPDRAWITEMSTRUCT pdis;
     POINT            pt[3];
     RECT             rc;

     switch(iMsg)
          {
          case WM_CREATE :
               cxChar = LOWORD(GetDialogBaseUnits());
               cyChar = HIWORD(GetDialogBaseUnits());
```

```
               // Create the owner-draw pushbuttons

          hwndSmaller = CreateWindow("button", "",
                              WS_CHILD | WS_VISIBLE | BS_OWNERDRAW,
                              0, 0, BTN_WIDTH, BTN_HEIGHT,
                              hwnd,(HMENU) IDC_SMALLER, hInst, NULL);

          hwndLarger = CreateWindow("button", "",
                              WS_CHILD | WS_VISIBLE | BS_OWNERDRAW,
                              0, 0, BTN_WIDTH, BTN_HEIGHT,
                              hwnd,(HMENU) IDC_LARGER, hInst, NULL);
          return 0;
     case WM_SIZE :
          cxClient = LOWORD(lParam);
          cyClient = HIWORD(lParam);

               // Move the buttons to the new center

          MoveWindow(hwndSmaller, cxClient / 2 - 3 * BTN_WIDTH  / 2,
                                  cyClient / 2 -     BTN_HEIGHT / 2,
                                  BTN_WIDTH, BTN_HEIGHT, TRUE);

          MoveWindow(hwndLarger,  cxClient / 2 +     BTN_WIDTH  / 2,
                                  cyClient / 2 -     BTN_HEIGHT / 2,
                                  BTN_WIDTH, BTN_HEIGHT, TRUE);
          return 0;

     case WM_COMMAND :
          GetWindowRect(hwnd, &rc);

               // Make the window 10% smaller or larger

          switch(wParam)
               {
               case IDC_SMALLER :
                    rc.left   += cxClient / 20;
                    rc.right  -= cxClient / 20;
                    rc.top    += cyClient / 20;
                    rc.bottom -= cyClient / 20;

                    break;

               case IDC_LARGER :
                    rc.left   -= cxClient / 20;
                    rc.right  += cxClient / 20;
                    rc.top    -= cyClient / 20;
                    rc.bottom += cyClient / 20;

                    break;
               }

          MoveWindow(hwnd, rc.left, rc.top, rc.right  - rc.left,
                                     rc.bottom - rc.top, TRUE);
          return 0;

     case WM_DRAWITEM :
          pdis =(LPDRAWITEMSTRUCT) lParam;

               // Fill area with white and frame it black

          FillRect(pdis->hDC, &pdis->rcItem,
                  (HBRUSH) GetStockObject(WHITE_BRUSH));
          FrameRect(pdis->hDC, &pdis->rcItem,
                   (HBRUSH) GetStockObject(BLACK_BRUSH));
```

```
          // Draw inward and outward black triangles

     cx = pdis->rcItem.right  - pdis->rcItem.left;
     cy = pdis->rcItem.bottom - pdis->rcItem.top;

     switch(pdis->CtlID)
          {
          case IDC_SMALLER :
               pt[0].x = 3 * cx / 8;  pt[0].y = 1 * cy / 8;
               pt[1].x = 5 * cx / 8;  pt[1].y = 1 * cy / 8;
               pt[2].x = 4 * cx / 8;  pt[2].y = 3 * cy / 8;

               Triangle(pdis->hDC, pt);

               pt[0].x = 7 * cx / 8;  pt[0].y = 3 * cy / 8;
               pt[1].x = 7 * cx / 8;  pt[1].y = 5 * cy / 8;
               pt[2].x = 5 * cx / 8;  pt[2].y = 4 * cy / 8;

               Triangle(pdis->hDC, pt);

               pt[0].x = 5 * cx / 8;  pt[0].y = 7 * cy / 8;
               pt[1].x = 3 * cx / 8;  pt[1].y = 7 * cy / 8;
               pt[2].x = 4 * cx / 8;  pt[2].y = 5 * cy / 8;

               Triangle(pdis->hDC, pt);

               pt[0].x = 1 * cx / 8;  pt[0].y = 5 * cy / 8;
               pt[1].x = 1 * cx / 8;  pt[1].y = 3 * cy / 8;
               pt[2].x = 3 * cx / 8;  pt[2].y = 4 * cy / 8;

               Triangle(pdis->hDC, pt);

               break;

          case IDC_LARGER :

               pt[0].x = 5 * cx / 8;  pt[0].y = 3 * cy / 8;
               pt[1].x = 3 * cx / 8;  pt[1].y = 3 * cy / 8;
               pt[2].x = 4 * cx / 8;  pt[2].y = 1 * cy / 8;

               Triangle(pdis->hDC, pt);

               pt[0].x = 5 * cx / 8;  pt[0].y = 5 * cy / 8;
               pt[1].x = 5 * cx / 8;  pt[1].y = 3 * cy / 8;
               pt[2].x = 7 * cx / 8;  pt[2].y = 4 * cy / 8;

               Triangle(pdis->hDC, pt);
               pt[0].x = 3 * cx / 8;  pt[0].y = 5 * cy / 8;
               pt[1].x = 5 * cx / 8;  pt[1].y = 5 * cy / 8;
               pt[2].x = 4 * cx / 8;  pt[2].y = 7 * cy / 8;

               Triangle(pdis->hDC, pt);

               pt[0].x = 3 * cx / 8;  pt[0].y = 3 * cy / 8;
               pt[1].x = 3 * cx / 8;  pt[1].y = 5 * cy / 8;
               pt[2].x = 1 * cx / 8;  pt[2].y = 4 * cy / 8;

               Triangle(pdis->hDC, pt);

               break;
          }

               // Invert the rectangle if the button is selected
```

```
            if(pdis->itemState & ODS_SELECTED)
                 InvertRect(pdis->hDC, &pdis->rcItem);

                      // Draw a focus rectangle if the button has the focus

            if(pdis->itemState & ODS_FOCUS)
                 {
                 pdis->rcItem.left   += cx / 16;
                 pdis->rcItem.top    += cy / 16;
                 pdis->rcItem.right  -= cx / 16;
                 pdis->rcItem.bottom -= cy / 16;

                 DrawFocusRect(pdis->hDC, &pdis->rcItem);
                 }

            return 0;

       case WM_DESTROY :
            PostQuitMessage(0);
            return 0;
       }
  return DefWindowProc(hwnd, iMsg, wParam, lParam);
  }
```

Рис. 8.3 Программа OWNERDRW

В этой программе создаются две кнопки в центре рабочей области окна программы, как показано на рис. 8.4. На левой кнопке находятся четыре треугольника, направленные в центр кнопки. Щелчок на этой кнопке вызывает уменьшение размеров окна на 10%. На правой кнопке находятся четыре треугольника, направленные от центра кнопки к ее сторонам, щелчок на этой кнопке вызывает увеличение размеров окна на 10%.

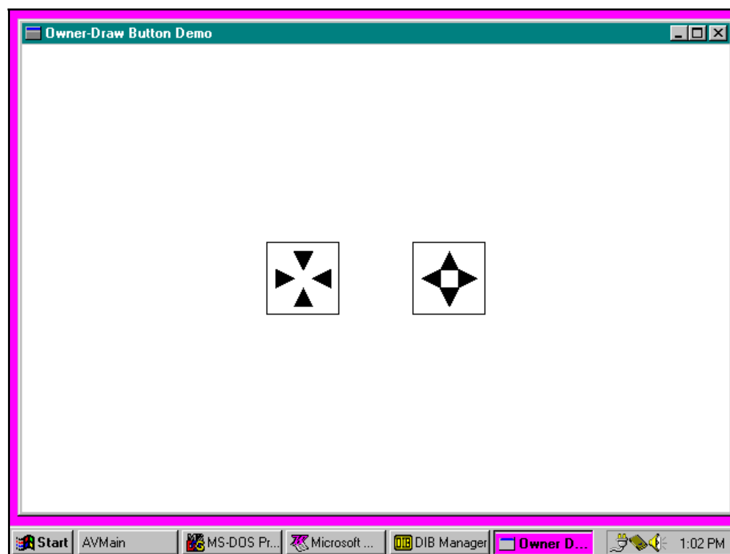


Рис. 8.4 Вид экрана программы OWNERDRW

Большинство программ, в которых для рисования собственных кнопок используется стиль кнопки BS_OWNERDRAW, часто для изображения применяют небольшие растровые образы кнопки. Что же касается программы OWNERDRW, то она просто рисует треугольники на поверхности кнопки.

При обработке сообщения WM_CREATE программа OWNERDRW с помощью вызова функции *GetDialogBaseUnits* получает среднюю высоту и ширину системного шрифта. Эта функция часто очень удобна для получения такой информации. Затем программа OWNERDRW создает две кнопки стиля BS_OWNERDRAW; кнопкам задается ширина, в восемь раз превышающая ширину системного шрифта, и высота, превышающая высоту системного шрифта в четыре раза. (При использовании для рисования кнопок, предопределенных в системе битовых образов, полезно знать, что эти размеры создают кнопки размером 64 на 64 пикселя на VGA.) Но кнопки еще не спозиционированы. При обработке сообщения WM_SIZE, программа OWNERDRW с помощью функции *MoveWindow* позиционирует кнопки в центр рабочей области.

Щелчок на кнопках заставляет их генерировать сообщение WM_COMMAND. Для обработки сообщения WM_COMMAND программа вызывает функцию *GetWindowRect*, чтобы сохранить положение и размер всего окна (а не только рабочей области) в структуре RECT (прямоугольник). Это положение определяется относительно экрана. Затем программа OWNERDRW модифицирует поля этой структуры в соответствии с тем, на левой или на правой кнопке был щелчок. Затем программа с помощью функции *MoveWindow* меняет положение и размер окна, что генерирует другое сообщение WM_SIZE, и кнопки перемещаются в центр рабочей области.

Если бы это было все, что делает программа, она была бы полностью работоспособной, но кнопки были бы невидимы. Кнопки стиля BS_OWNERDRAW при необходимости перерисовки посылают 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.

При обработке сообщения WM_DRAWITEM параметр *lParam* этого сообщения является указателем на структуру типа DRAWITEMSTRUCT. Программа OWNERDRW хранит этот указатель в переменной *pdis*. Эта структура содержит информацию, необходимую программе для рисования кнопки. (Такая же структура используется для создания определяемых пользователем списков.) Полями структуры, которые важны для работы с кнопками, являются *hDC* (контекст устройства для кнопки), *rcItem* (структура RECT с размерами кнопки), *CtlID* (идентификатор окна управления) и *itemState* (которое показывает, нажата ли кнопка и имеет ли она фокус ввода).

Программа OWNERDRW начинает обработку сообщения WM_DRAWITEM с вызова функции *FillRect* для обновления поверхности кнопки белой кистью и с вызова функции *FrameRect* для рисования черной рамки вокруг кнопки. Затем, вызывая функцию *Polygon*, программа OWNERDRW рисует четыре черных треугольника. Это обычный случай.

Если кнопка в данный момент нажимается, то в поле *itemState* структуры DRAWITEMSTRUCT устанавливается бит. Вы можете выяснить, установлен ли этот бит, используя константу ODS_SELECTED. Если бит установлен, программа OWNERDRW, вызывая функцию *InvertRect*, меняет цвет кнопки на обратный. Если кнопка имеет фокус ввода, тогда будет установлен бит ODS_FOCUS поля *itemState*. В этом случае программа OWNERDRW с помощью вызова функции *DrawFocusRect* рисует точечный прямоугольник внутри кнопки точно вдоль ее сторон.

Предупреждение, связанное с использованием кнопок, определяемых пользователем: Windows получает для вас контекст устройства и включает его в виде поля в структуру DRAWITEMSTRUCT. После использования контекста устройства, оставляйте его в том же состоянии, в каком он находился до этого. Все объекты GDI, выбранные в контекст устройства, должны быть удалены из контекста. Кроме того, будьте внимательны, не рисуйте вне прямоугольника, задающего границы кнопки.

Класс статических дочерних окон

Вы можете создать статическое дочернее окно управления, используя класс окна "static" при вызове функции *CreateWindow*. Это совершенно обычные дочерние окна. Они не получают информации от клавиатуры или мыши, и они не посылают сообщений WM_COMMAND обратно родительскому окну. (Когда вы перемещаете мышь или щелкаете мышью над статическим дочерним окном, дочернее окно обрабатывает сообщение WM_NCHITTEST и возвращает в Windows значение HTTRANSPARENT. Это заставляет Windows послать то же сообщение WM_NCHITTEST расположенному внизу окну, которым обычно является родительское окно. Родительское окно, как правило, передает сообщение в *DefWindowProc*, где оно преобразуется в сообщение мыши рабочей области.)

Первые шесть стилей статического окна рисуют прямоугольник или рамку в рабочей области дочернего окна. В приведенной ниже таблице статические стили "RECT" (левый столбец) являются закрашенными прямоугольниками; три статических стиля "FRAME" (правый столбец) представляют из себя прямоугольные рамки без закрашивания прямоугольника:

SS_BLACKRECT	SS_BLACKFRAME
SS_GRAYRECT	SS_GRAYFRAME
SS_WHITERECT	SS_WHITEFRAME

"BLACK", "GRAY" и "WHITE" не означает, что цветами являются соответственно черный, серый и белый. Эти цвета основаны на системных цветах так, как это показано ниже:

Кнопка статического класса	Системный цвет
BLACK	COLOR_3DDKSHADOW
GRAY	COLOR_BTNSHADOW
WHITE	COLOR_BTNHIGHLIGHT

Поле текста окна функции *CreateWindow* для этих стилей игнорируется. Верхний левый угол прямоугольника расположен в точке с координатами *x* и *y* по отношению к родительскому окну. (Вы также можете пользоваться

стилями SS_ETCHEDHORZ, SS_ETCHEDVERT или SS_ETCHEDFRAME для создания рамки с тенью, состоящей из серого и белого цветов.)

Статический класс также включает в себя три стиля текста: SS_LEFT, SS_RIGHT и SS_CENTER. Они предназначены для выравнивания текста соответственно по левому краю, по правому краю и по центру. Текст задается в параметре текста окна функции *CreateWindow*, и позднее он может быть изменен с помощью функции *SetWindowText*. Когда оконная процедура кнопки управления статического класса выводит на экран этот текст, она использует функцию *DrawText* с параметрами DT_WORDBREAK, DT_NOCLIP и DT_EXPANDTABS. Текст помещается внутрь прямоугольника дочернего окна.

Фоном дочерних окон этих трех стилей обычно является COLOR_BTNFACE, а самого текста — COLOR_WINDOWTEXT. Вы можете обрабатывать сообщения WM_CTLCOLORSTATIC для изменения цвета текста с помощью вызова функции *SetTextColor*, а для изменения цвета фона текста — с помощью вызова функции *SetBkColor*, возвращая при этом описатель кисти фона. Это будет вскоре продемонстрировано в программе COLORS1.

И наконец, статический класс также содержит стили окна SS_ICON и SS_USERITEM. Однако, эти стили не имеют смысла при использовании в качестве дочерних окон управления. К этим стилям мы вернемся при изучении окон диалога.

Класс полос прокрутки

Когда в главе 3 при написании программ серии SYSMETS, мы впервые коснулись темы полос прокрутки (scrollbar), там говорилось о некоторых отличиях между "полосами прокрутки окна" и "полосами прокрутки — элементами управления". В SYSMETS использовались полосы прокрутки окна, которые появлялись в правой и нижней частях окна. Вы добавляете к окну при его создании полосы прокрутки путем включения в стиль окна идентификаторов WS_VSCROLL или WS_HSCROLL, или обоих сразу. Теперь мы готовы создать некие полосы прокрутки, которые являются дочерними окнами управления, и которые могут располагаться в любом месте рабочей области родительского окна. Вы создаете полосы прокрутки, являющиеся дочерними окнами управления, используя предопределенный класс окна "scrollbar", и один из двух стилей для полос прокрутки: SBS_VERTS и BS_HORZ.

В отличие от кнопок — элементов управления (и окон редактирования и списков, которые будут обсуждаться позже), полосы прокрутки — элементы управления не посылают родительскому окну сообщений WM_COMMAND. Вместо этого они, также как и полосы прокрутки окна, посылают ему сообщения WM_VSCROLL и WM_HSCROLL. При обработке сообщений полос прокрутки с помощью параметра *lParam* вы можете различать сообщения полос прокрутки окна и полос прокрутки — элементов управления. Для полос прокрутки окна *lParam* равен 0, а для полос прокрутки — элементов управления он является описателем этих полос прокрутки или описателем дочернего окна управления — полосы прокрутки. Что же касается параметра *wParam*, то значения его старшего и младшего слова для полос прокрутки окна и для полос прокрутки — элементов управления имеют одинаковый смысл.

Несмотря на то, что полосы прокрутки окна имеют фиксированную ширину, для задания размеров полос прокрутки — элементов управления в Windows используются размеры всего прямоугольника, задаваемые при вызове функции *CreateWindow* (или позже при вызове функции *MoveWindow*). Вы можете сделать полосы прокрутки управления длинными и узкими, или короткими и широкими.

Если вы захотите создать полосы прокрутки — элементы управления с теми же размерами, что и полосы прокрутки окна, вы можете использовать для получения высоты горизонтальной полосы прокрутки функцию *GetSystemMetrics*:

```
GetSystemMetrics(SM_CYHSCROLL);
```

или для получения ширины вертикальной полосы прокрутки:

```
GetSystemMetrics(SM_CXVSCROLL);
```

(Для задания стандартных размеров полосам прокрутки предназначены идентификаторы стиля окон полос прокрутки SBS_LEFTALIGN, SBS_RIGHTALIGN, SBS_TOPALIGN и SBS_BOTTOMALIGN. Однако, эти стили применимы только для полос прокрутки в окнах диалога.

Вы можете установить диапазон и положение полосы прокрутки — элемента управления с помощью тех же вызовов функций, что и для полос прокрутки окна:

```
SetScrollRange(hwndScroll, SB_CTL, iMin, iMax, bRedraw);
SetScrollPos(hwndScroll, SB_CTL, iPos, bRedraw);
SetScrollInfo(hwndScroll, SB_CTL, &si, bRedraw);
```

Отличие состоит в том, что для полос прокрутки окна в качестве первого параметра используется описатель главного окна, а в качестве второго SB_VERT или SB_HORZ.

Достаточно удивительно то, что системный цвет COLOR_SCROLLBAR более для полос прокрутки не используется. Цвета концевых кнопок и бегунка основаны на COLOR_BTNFACE, COLOR_BTNHIGHLIGHT, COLOR_BTNSHADOW, COLOR_BTNTEXT (для маленьких стрелок), COLOR_BTNSHADOW и COLOR_BTNLIGHT. Цвет большого участка между верхней и нижней концевыми кнопками основан на сочетании цветов COLOR_BTNFACE и COLOR_BTNHIGHLIGHT.

Если вы обрабатываете сообщения WM_CTLCOLORSCROLLBAR, то вы можете возвратить кисть из сообщения для изменения цвета определенной области. Давайте это проделаем.

Программа COLORS1

Для рассмотрения некоторых аспектов использования статических дочерних окон и полос прокрутки, а также для более глубокого изучения цветов, мы будем использовать программу COLORS1, представленную на рис. 8.5. Программа COLORS1 выводит на экран три полосы прокрутки в левой половине рабочей области, помеченные "Red", "Green" и "Blue". Цвет правой половины рабочей области образуется путем сочетания трех исходных цветов, значения которых определяются положением бегунков полос прокрутки. Численные значения этих трех исходных цветов выводятся на экран под тремя полосами прокрутки.

COLORS1.MAK

```
#-----------------------
# COLORS1.MAK make file
#-----------------------

colors1.exe : colors1.obj
     $(LINKER) $(GUIFLAGS) -OUT:colors1.exe colors1.obj $(GUILIBS)

colors1.obj : colors1.c
     $(CC) $(CFLAGS) colors1.c
```

COLORS1.C

```
/*----------------------------------------
   COLORS1.C -- Colors Using Scroll Bars
                (c) Charles Petzold, 1996
  ----------------------------------------*/

#include <windows.h>
#include <stdlib.h>

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

WNDPROC fnOldScr[3];
HWND    hwndScrol[3], hwndLabel[3], hwndValue[3], hwndRect;
int     color[3], iFocus;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char  szAppName[] = "Colors1";
     static char *szColorLabel[] = { "Red", "Green", "Blue" };
     HWND         hwnd;
     int          i;
     MSG          msg;
     WNDCLASSEX   wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground = CreateSolidBrush(0L);
```

```
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Color Scroll",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     hwndRect = CreateWindow("static", NULL,
                             WS_CHILD | WS_VISIBLE | SS_WHITERECT,
                             0, 0, 0, 0,
                             hwnd,(HMENU) 9, hInstance, NULL);

     for(i = 0; i < 3; i++)
          {
          hwndScrol[i] = CreateWindow("scrollbar", NULL,
                             WS_CHILD | WS_VISIBLE | WS_TABSTOP | SBS_VERT,
                             0, 0, 0, 0,
                             hwnd,(HMENU) i, hInstance, NULL);

          hwndLabel[i] = CreateWindow("static", szColorLabel[i],
                             WS_CHILD | WS_VISIBLE | SS_CENTER,
                             0, 0, 0, 0,
                             hwnd,(HMENU)(i + 3), hInstance, NULL);

          hwndValue[i] = CreateWindow("static", "0",
                             WS_CHILD | WS_VISIBLE | SS_CENTER,
                             0, 0, 0, 0,
                             hwnd,(HMENU)(i + 6), hInstance, NULL);

          fnOldScr[i] =(WNDPROC) SetWindowLong(hwndScrol[i], GWL_WNDPROC,
                                          (LONG) ScrollProc);

          SetScrollRange(hwndScrol[i], SB_CTL, 0, 255, FALSE);
          SetScrollPos  (hwndScrol[i], SB_CTL, 0, FALSE);
          }

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static COLORREF crPrim[3] = { RGB(255, 0, 0), RGB(0, 255, 0),
                                   RGB(0, 0, 255) };
     static HBRUSH   hBrush[3], hBrushStatic;
     static int      cyChar;
     static RECT     rcColor;
     char            szbuffer[10];
     int             i, cxClient, cyClient;

     switch(iMsg)
          {
```

```
case WM_CREATE :
     for(i = 0; i < 3; i++)
          hBrush[i] = CreateSolidBrush(crPrim[i]);

     hBrushStatic = CreateSolidBrush(
                         GetSysColor(COLOR_BTNHIGHLIGHT));

     cyChar = HIWORD(GetDialogBaseUnits());
     return 0;

case WM_SIZE :
     cxClient = LOWORD(lParam);
     cyClient = HIWORD(lParam);

     SetRect(&rcColor, cxClient / 2, 0, cxClient, cyClient);

     MoveWindow(hwndRect, 0, 0, cxClient / 2, cyClient, TRUE);

     for(i = 0; i < 3; i++)
          {
          MoveWindow(hwndScrol[i],
              (2 * i + 1) * cxClient / 14, 2 * cyChar,
               cxClient / 14, cyClient - 4 * cyChar, TRUE);

          MoveWindow(hwndLabel[i],
              (4 * i + 1) * cxClient / 28, cyChar / 2,
               cxClient / 7, cyChar, TRUE);

          MoveWindow(hwndValue[i],
              (4 * i + 1) * cxClient / 28, cyClient - 3 * cyChar / 2,
               cxClient / 7, cyChar, TRUE);
          }
     SetFocus(hwnd);
     return 0;

case WM_SETFOCUS :
     SetFocus(hwndScrol[iFocus]);
     return 0;

case WM_VSCROLL :
     i = GetWindowLong((HWND) lParam, GWL_ID);

     switch(LOWORD(wParam))
          {
          case SB_PAGEDOWN :
               color[i] += 15;
                                   // fall through
          case SB_LINEDOWN :
               color[i] = min(255, color[i] + 1);
               break;

          case SB_PAGEUP :
               color[i] -= 15;
                                   // fall through
          case SB_LINEUP :
               color[i] = max(0, color[i] - 1);
               break;

          case SB_TOP :
               color[i] = 0;
               break;

          case SB_BOTTOM :
               color[i] = 255;
```

```
                    break;

                case SB_THUMBPOSITION :
                case SB_THUMBTRACK :
                    color[i] = HIWORD(wParam);
                    break;

                default :
                    break;
                }
            SetScrollPos (hwndScrol[i], SB_CTL, color[i], TRUE);
            SetWindowText(hwndValue[i], itoa(color[i], szbuffer, 10));

            DeleteObject((HBRUSH)
                SetClassLong(hwnd, GCL_HBRBACKGROUND,
                    (LONG) CreateSolidBrush(
                                RGB(color[0], color[1], color[2]))));

            InvalidateRect(hwnd, &rcColor, TRUE);
            return 0;

        case WM_CTLCOLORSCROLLBAR :
            i = GetWindowLong((HWND) lParam, GWL_ID);

            return(LRESULT) hBrush[i];

        case WM_CTLCOLORSTATIC :
            i = GetWindowLong((HWND) lParam, GWL_ID);

            if(i >= 3 && i <= 8)    // static text controls
                {
                SetTextColor((HDC) wParam, crPrim[i % 3]);
                SetBkColor((HDC) wParam, GetSysColor(COLOR_BTNHIGHLIGHT));
                return(LRESULT) hBrushStatic;
                }
            break;

        case WM_SYSCOLORCHANGE :
            DeleteObject(hBrushStatic);

            hBrushStatic = CreateSolidBrush(
                                GetSysColor(COLOR_BTNHIGHLIGHT));
            return 0;

        case WM_DESTROY :
            DeleteObject((HBRUSH)
                SetClassLong(hwnd, GCL_HBRBACKGROUND,
                    (LONG) GetStockObject(WHITE_BRUSH)));

            for(i = 0; i < 3; DeleteObject(hBrush[i++]));

            DeleteObject(hBrushStatic);

            PostQuitMessage(0);
            return 0;
        }
    return DefWindowProc(hwnd, iMsg, wParam, lParam);
    }

LRESULT CALLBACK ScrollProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
    {
    int i = GetWindowLong(hwnd, GWL_ID);

    switch(iMsg)
```

```
        {
        case WM_KEYDOWN :
             if(wParam == VK_TAB)
                  SetFocus(hwndScrol[(i +
                      (GetKeyState(VK_SHIFT) < 0 ? 2 : 1)) % 3]);
             break;

        case WM_SETFOCUS :
             iFocus = i;
             break;
        }
   return CallWindowProc(fnOldScr[i], hwnd, iMsg, wParam, lParam);
   }
```

Рис. 8.5 Программа COLORS1

Программа COLORS1 включает в работу свои дочерние окна. В программе используется 10 дочерних окон управления: три полосы прокрутки, 6 окон статического текста, и один статический прямоугольник. Программа COLORS1 обрабатывает сообщения WM_CTLCOLORSCROLLBAR для закрашивания внутренних участков трех полос прокрутки в красный, зеленый и голубой цвета, а также сообщения WM_CTLCOLORSTATIC для окрашивания статического текста.

Вы можете изменять состояние полос прокрутки либо с помощью мыши, либо с помощью клавиатуры. Вы можете использовать программу COLORS1 как инструмент для экспериментов с цветами и выбора наиболее привлекательного (или, если вам так хочется, самого некрасивого) цвета для ваших Windows-программ. Вид экрана программы COLORS1 показан на рис. 8.6, к сожалению, цвета печатной страницы представляют собой лишь варианты серого.

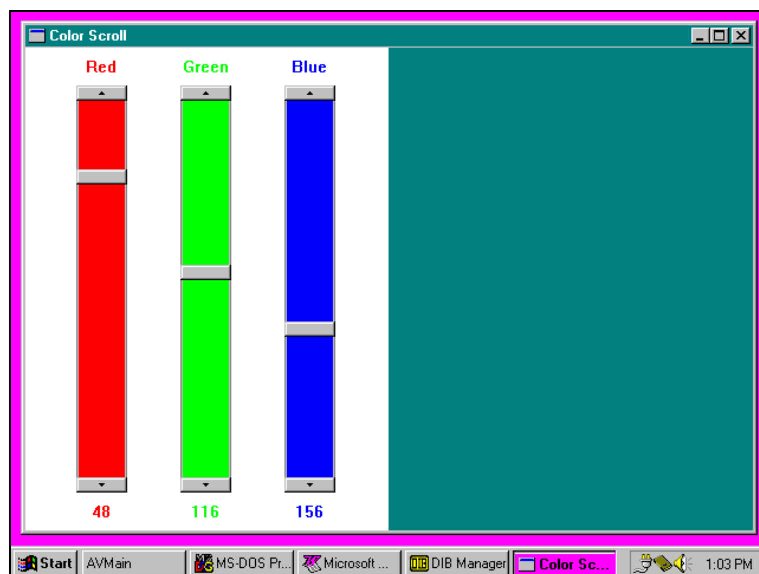


Рис. 8.6 Вид экрана программы COLORS1

Программа COLORS1 не обрабатывает сообщений WM_PAINT. Фактически вся работа в программе COLORS1 выполняется дочерними окнами.

Цвет правой половины рабочей области окна фактически является цветом фона окна. Статическое дочернее окно со стилем SS_WHITERECT занимает левую половину рабочей области. Три полосы прокрутки являются дочерними окнами управления стиля SBS_VERT. Эти полосы прокрутки расположены в верхней части дочернего окна SS_WHITERECT. Еще шесть статических дочерних окон стиля SS_CENTER (выравнивание по центру) обеспечивают индикацию названий цветов и их значений. Программа COLORS1 строит обычное перекрывающееся окно и десять дочерних окон в функции *WinMain* при помощи функции *CreateWindow*. Окна SS_WHITERECT и SS_CENTER используют класс окна "static"; в трех полосах прокрутки используется класс окна "scrollbar".

Первоначально параметры, определяющие положение *x* и *y*, ширину и высоту окна функции *CreateWindow* устанавливаются в 0, поскольку они зависят от размера рабочей области, который еще не известен. Оконная процедура программы COLORS1 с помощью вызова функции *MoveWindow* изменяет размеры всех десяти

дочерних окон, когда получает сообщение WM_SIZE. Поэтому, когда бы вы ни изменили размер окна программы COLORS1, пропорционально меняются и размеры полос прокрутки.

Когда оконная процедура *WndProc* получает сообщение WM_VSCROLL, то старшим словом параметра *lParam* является описатель дочернего окна. Для получения идентификатора дочернего окна можно использовать функцию *GetWindowLong*:

```
i = GetWindowLong(lParam, GWW_ID);
```

Мы задали идентификаторы трех полос прокрутки как три последовательных числа 0, 1 и 2, поэтому *WndProc* может информировать о том, какая из полос является источником сообщения.

Поскольку описатели дочерних окон при создании этих окон были сохранены в массиве, *WndProc* может обработать сообщение полосы прокрутки и установить новое положение соответствующей полосы прокрутки с помощью вызова функции *SetScrollPos*:

```
SetScrollPos(hwndScrol[i], SB_CTL, color[i], TRUE);
```

WndProc также изменяет текст дочернего окна под полосой прокрутки:

```
SetWindowText(hwndValue[i], itoa(color[i], szbuffer, 10));
```

Интерфейс клавиатуры, поддерживаемый автоматически

Полосы прокрутки — элементы управления также могут обрабатывать сообщения от клавиатуры, но только в том случае, если они имеют фокус ввода. В следующей таблице показано, как нажатия клавиш управления курсором преобразуются в сообщения полос прокрутки:

Клавиши управления курсором	**Значение *wParam* сообщения полосы прокрутки**
Home	SB_TOP
End	SB_BOTTOM
Page Up	SB_PAGEUP
Page Down	SB_PAGEDOWN
Стрелка влево или вверх	SB_LINEUP
Стрелка вправо или вниз	SB_LINEDOWN

Фактически, сообщения полос прокрутки SB_TOP и SB_BOTTOM могут вырабатываться только с помощью клавиатуры. Если вы хотите, чтобы полоса прокрутки управления получила фокус ввода, когда на полосе прокрутки происходит щелчок мыши, то вы должны включить идентификатор WS_TABSTOP в параметр стиля окна при вызове функции *CreateWindow*. Если полоса прокрутки имеет фокус ввода, то бегунок полосы прокрутки становится похож на мигающий серый блочок.

Чтобы полностью обеспечить интерфейс клавиатуры для полос прокрутки, требуется затратить несколько больше усилий. Во-первых, оконная процедура *WndProc* должна специально передать полосе прокрутки фокус ввода. Она это делает, обрабатывая сообщение WM_SETFOCUS, которое получает родительское окно при получении фокуса ввода. *WndProc* просто устанавливает фокус ввода на одну из полос прокрутки:

```
SetFocus(hwndScrol[iFocus])
```

Но кроме этого нужно иметь возможность как-то переходить от одной полосы прокрутки к другой, предпочтительнее с помощью клавиши <Tab>. Это более трудно, поскольку, раз полоса прокрутки имеет фокус ввода, она обрабатывает все нажатия клавиш. Но полоса прокрутки отслеживает только клавиши управления курсором; клавиша <Tab> ею игнорируется. Для решения этой задачи существует прием, который называется "введение новой оконной процедуры" (window subclassing). Мы будем пользоваться этим приемом, чтобы в программе COLORS1 получить возможность с помощью клавиши <Tab> переходить с одной полосы прокрутки на другую.

Введение новой оконной процедуры

Оконная процедура для полос прокрутки — элементов управления находится где-то внутри Windows. Однако, вы можете получить адрес этой оконной процедуры с помощью вызова функции *GetWindowLong*, в которой в качестве параметра используется идентификатор GWL_WNDPROC. Более того, вызывая функцию *SetWindowLong*, вы можете задать для полос прокрутки новую оконную процедуру. Это очень мощный прием, который называется "введение новой оконной процедуры". Он позволяет вам "влезть" в существующие внутри Windows оконные процедуры, обработать некоторые сообщения внутри вашей собственной программы, а все остальные сообщения оставить прежней оконной процедуре.

Оконная процедура, которая в программе COLORS1 предварительно обрабатывает сообщения полос прокрутки, называется *ScrollProc*; она находится в конце программы COLORS1.C. Поскольку *ScrollProc* является функцией программы COLORS1, которая вызывается операционной системой Windows, то она должна определяться как функция обратного вызова (CALLBACK).

Для каждой из трех полос прокрутки в программе COLORS1, для установки адреса новой оконной процедуры, а также для получения адреса существующей оконной процедуры полосы прокрутки, используется функция *SetWindowLong*:

```
fnOldScr[i] =(WNDPROC) SetWindowLong(hwndScrol[i], GWL_WNDPROC,(LONG) ScrollProc));
```

Теперь функция *ScrollProc* получает все сообщения, которые Windows посылает оконной процедуре полосы прокрутки для трех полос прокрутки программы COLORS1 (но, конечно, не для полос прокрутки других программ). Оконная процедура *ScrollProc*, при получении сообщения о нажатии клавиши <Tab> или <Shift>+<Tab>, просто передает фокус ввода следующей (или предыдущей) полосе прокрутки. С помощью функции *CallWindowProc* она вызывает прежнюю оконную процедуру полосы прокрутки.

Закрашивание фона

Когда программа COLORS1 определяет свой класс окна, она задает для своей рабочей области сплошную черную кисть:

```
wndclass.hbrBackground = CreateSolidBrush(0L);
```

Если вы изменяете установки полос прокрутки программы COLORS1, то программа должна создать новую кисть и поместить в структуру класса окна новый описатель кисти. Точно также, как мы получали адрес прежней и вводили новую оконную процедуру полос прокрутки с помощью функций *GetWindowLong* и *SetWindowLong*, мы можем получить и ввести новый описатель этой кисти с помощью функций *GetClassWord* и *SetClassWord*.

Вы можете создать новую кисть, ввести ее описатель в структуру класса окна, а затем удалить старую кисть:

```
DeleteObject(
       (HBRUSH)SetClassLong(
                            hwnd,
                            GCL_HBRBACKGROUND,
                            (LONG)CreateSolidBrush(RGB(color[0], color[1], color[2]))
                            )
              );
```

Следующий раз Windows при перерисовке фона окна будет пользоваться новой кистью. Чтобы заставить Windows обновить фон, мы делаем недействительной правую половину рабочей области:

```
InvalidateRect(hwnd, &rcColor, TRUE);
```

Использования в качестве третьего параметра значения TRUE (не равно 0) показывает, что перед рисованием мы хотим обновить фон.

Функция *InvalidateRect* заставляет Windows поместить сообщение WM_PAINT в очередь сообщений оконной процедуры. Поскольку сообщения WM_PAINT имеют низкий приоритет, то такое сообщение, если вы еще перемещаете полосу прокрутки с помощью мыши или клавиш управления курсором, не будет обработано немедленно. В противном случае, если вы хотите, чтобы окно обновилось сразу после того, как его цвет был изменен, в программу, после вызова функции *InvalidateRect* необходимо добавить следующую инструкцию:

```
UpdateWindow(hwnd);
```

Но это может затормозить обработку сообщений клавиатуры и мыши.

Функция *WndProc* программы COLORS1 не обрабатывает сообщения WM_PAINT, а передает его в *DefWindowProc*. Заданный в Windows по умолчанию процесс обработки сообщений WM_PAINT заключается просто в вызовах функций *BeginPaint* и *EndPaint*, которые делают окно действительным. Поскольку мы задали при вызове функции *InvalidateRect*, что фон должен быть обновлен, то вызов функции *BeginPaint* заставляет Windows выработать сообщение WM_ERASEBKGND (обновление фона). *WndProc* игнорирует и это сообщение тоже. Windows обрабатывает его, обновляя фон рабочей области с помощью кисти, заданной в классе окна.

Считается хорошим стилем программирования удалять созданные ресурсы. Поэтому при обработке сообщения WM_DESTROY функция *DeleteObject* вызывается снова:

```
DeleteObject((HBRUSH)SetClassLong(hwnd, GCL_HBRBACKGROUND,(LONG) GetStockObject(WHITE_BRUSH)));
```

Окрашивание полос прокрутки и статического текста

В программе COLORS1 внутренние участки трех полос прокрутки и текст шести текстовых полей окрашиваются красным, зеленым и голубым цветами. Окрашивание полос прокрутки осуществляется путем обработки сообщений WM_CTLCOLORSCROLLBAR.

В *WndProc* мы для кистей определяем статический массив трех описателей:

```
static HBRUSH hBrush[3];
```

При обработке сообщения WM_CREATE мы создаем три кисти:

```
for(i = 0; i < 3; i++)
       hBrush[i] = CreateSolidBrush(crPrim[i]);
```

где в массиве *crPrim* хранятся RGB-значения трех первичных цветов. При обработке сообщений WM_CTLCOLORSCROLLBAR, возвращаемым значением оконной процедуры является одна из этих трех кистей:

```
case WM_CTLCOLORSCROLLBAR:
       i = GetWindowLong((HWND) lParam, GWW_ID);
       return(LRESULT) hBrush[i];
```

При обработке сообщения WM_DESTROY эти три кисти должны быть удалены:

```
for(i = 0; i < 3; DeleteObject(hBrush[i++]));
```

Аналогичным образом, путем обработки сообщения WM_CTLCOLORSTATIC и вызова функции *SetTextColor*, окрашивается текст в статических текстовых полях. Фон текста устанавливается функцией *SetBkColor* с системным цветом COLOR_BTNHIGHLIGHT. Это приводит к тому, что фон текста становится таким же, как цвет статического прямоугольника окна управления, который находится позади полос прокрутки и текста. Для статических дочерних текстовых окон управления этот цвет фона относится только к прямоугольнику позади каждого символа строки, а не ко всей ширине окна управления. Для того, чтобы это реализовать, оконная процедура должна также возвращать описатель кисти цвета COLOR_BTNHIGHLIGHT. Эта кисть называется *hBrushStatic* и создается при обработке сообщения WM_CREATE, а удаляется при обработке сообщения WM_DESTROY.

Создав, при обработке сообщения WM_CREATE, кисть на основе цвета COLOR_BTNHIGHLIGHT, и пользуясь ею на протяжении работы программы, мы создали себе маленькую проблему. Если во время работы программы цвет COLOR_BTNHIGHLIGHT изменяется, то изменится и цвет статического прямоугольника, а также цвет его текстового фона, однако весь фон текстового окна управления останется прежним — COLOR_BTNHIGHLIGHT.

Для решения этой проблемы в программе COLORS1 обрабатывается сообщение WM_SYSCOLORCHANGE путем простого повторного создания кисти *hBrushStatic*, использующей новый цвет.

Класс редактирования

Класс редактирования (edit) является, в некотором смысле, простейшим из предопределенных в Windows классов окна, хотя с других точек зрения он оказывается и более сложным. Когда вы создаете дочернее окно, используя имя класса "edit", вы определяете прямоугольник на основе параметров положения *x* и *y*, ширины и высоты функции *CreateWindow*. В этом прямоугольнике содержится редактируемый текст. Когда дочернее окно управления имеет фокус ввода, вы можете набирать текст, двигать курсор, выбирать группы символов, используя либо мышь, либо клавишу <Shift> и клавиши управления курсором, удалять выбранный текст в папку обмена нажимая комбинацию клавиш <Ctrl>+<X>, копировать текст нажимая комбинацию клавиш <Ctrl>+<C>, и вставлять текст из папки обмена используя комбинацию клавиш <Ctrl>+<V>.

Одним из простейших применений окон редактирования — элементов управления (или управляющих окон редактирования, edit controls) является простое однострочное окно ввода данных. Но окна редактирования не ограничены только одной строкой, что иллюстрируется в программе POPPAD1, представленной на рис. 8.7. Как программы, с которыми мы уже сталкивались в этой книге, программа POPPAD будет модернизироваться с целью использования окон меню, диалога (для открытия и сохранения файлов) и принтеров. Последней версией программы будет простой, но полноценный текстовый редактор с небольшими добавлениями, которые потребуется сделать для его реализации в тексте нашей программы.

POPPAD1.MAK

```
#----------------------
# POPPAD1.MAK make file
#----------------------

poppad1.exe : poppad1.obj
```

```
     $(LINKER) $(GUIFLAGS) -OUT:poppad1.exe poppad1.obj $(GUILIBS)

poppad1.obj : poppad1.c
     $(CC) $(CFLAGS) poppad1.c
```

POPPAD1.C

```
/*-------------------------------------------------------
   POPPAD1.C -- Popup Editor using child window edit box
                (c) Charles Petzold, 1996
  -------------------------------------------------------*/

#include <windows.h>

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

char szAppName[] = "PopPad1";

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     HWND       hwnd;
     MSG        msg;
     WNDCLASSEX wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);
     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, szAppName,
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HWND hwndEdit;

     switch(iMsg)
          {
          case WM_CREATE :
               hwndEdit = CreateWindow("edit", NULL,
                         WS_CHILD | WS_VISIBLE | WS_HSCROLL | WS_VSCROLL |
```

```
                            WS_BORDER | ES_LEFT | ES_MULTILINE |
                            ES_AUTOHSCROLL | ES_AUTOVSCROLL,
                       0, 0, 0, 0,
                       hwnd,(HMENU) 1,
                      ((LPCREATESTRUCT) lParam) -> hInstance, NULL);
             return 0;

        case WM_SETFOCUS :
             SetFocus(hwndEdit);
             return 0;

        case WM_SIZE :
             MoveWindow(hwndEdit, 0, 0, LOWORD(lParam),
                                         HIWORD(lParam), TRUE);
             return 0;

        case WM_COMMAND :
             if(LOWORD(wParam) == 1)
                  if(HIWORD(wParam) == EN_ERRSPACE ||
                      HIWORD(wParam) == EN_MAXTEXT)
                            MessageBox(hwnd, "Edit control out of space.",
                                         szAppName, MB_OK | MB_ICONSTOP);
             return 0;
        case WM_DESTROY :
             PostQuitMessage(0);
             return 0;
        }
   return DefWindowProc(hwnd, iMsg, wParam, lParam);
   }
```

Рис. 8.7 Программа POPPAD1

Программа POPPAD1 — это многострочный текстовый редактор (хотя пока еще и без возможности ввода/вывода файлов), который уместился в менее чем 100 строках текста на языке С. (Один недостаток, тем не менее, имеется. Он состоит в том, что предопределенное многострочное окно редактирования не позволяет редактировать текст размером более 32 килобайт.) Как видите, сама программа POPPAD1 делает немного. Предопределенное окно редактирования делает достаточно много. По этой причине программа позволяет вам узнать о потенциальных возможностях управляющего окна редактирования как такового, без помощи вашей программы.

Стили класса редактирования

Как уже говорилось, вы создаете управляющее окно редактирования, используя в качестве имени класса окна "edit" при вызове функции *CreateWindow*. Стилем окна является WS_CHILD и еще несколько опций. Как и в статических дочерних окнах управления, текст в управляющих окнах редактирования может быть выравнен либо по левому краю, либо по правому, либо по центру. Формат можно задать с помощью стилей окна ES_LEFT, ES_RIGHT и ES_CENTER.

По умолчанию в управляющем окне редактирования имеется одна строка. Вы можете создать многострочное управляющее окно редактирования, используя стиль окна ES_MULTILINE. Для однострочного управляющего окна редактирования обычно можно вводить текст только в конце прямоугольника редактирования. Для создания управляющего окна редактирования с автоматической горизонтальной прокруткой используйте стиль ES_AUTOHSCROOL. Для многострочного управляющего окна редактирования, если не задан стиль ES_AUTOHSCROOL, то текст автоматически переносится на новую строку. При задании этого стиля для перехода на новую строку нужно нажимать клавишу <Enter>. Используя стиль окна ES_AUTOVSCROOL, в многострочное управляющее окно редактирования можно включить полосу вертикальной прокрутки.

Если вы включите эти стили прокрутки в многострочные управляющие окна редактирования, то вы можете добавить полосы прокрутки к управляющему окну редактирования. Это делается путем использования тех же идентификаторов стиля окна, что и для недочерних окон: WS_HSCROLL и WS_VSCROLL.

По умолчанию в управляющем окне редактирования отсутствует рамка окна. Добавить ее можно, используя стиль WS_BORDER.

Когда вы выделяете текст в управляющем окне редактирования, Windows отображает его на экране в инвертированном виде. Однако, при потере управляющим окном редактирования фокуса ввода, выбранный текст

больше не выделяется. Если вы хотите, чтобы выделенный текст подсвечивался даже в том случае, если управляющее окно редактирования не имеет фокуса ввода, вы можете использовать стиль ES_NOHIDESEL.

Когда в программе POPPAD1 создается управляющее окно редактирования, его стиль при вызове функции *CreateWindow* задается равным:

```
WS_CHILD | WS_VISIBLE | WS_HSCROLL | WS_VSCROLL | WS_BORDER | ES_LEFT | ES_MULTILINE |
ES_AUTOHSCROLL | ES_AUTOVSCROLL
```

Далее в программе POPPAD1 определяются размеры управляющего окна редактирования, это делается путем вызова функции *MoveWindow* при получении оконной процедурой *WndProc* сообщения WM_SIZE. Размер управляющего окна просто устанавливается равным размеру главного окна:

```
MoveWindow(hwndEdit, 0, 0, LOWORD(lParam), HIWORD(lParam), TRUE);
```

Для однострочного управляющего окна редактирования высота окна должна соответствовать высоте символа. Если управляющее окно редактирования имеет рамку (как чаще и бывает), используйте полуторную высоту символа (включая межстрочное пространство).

Коды уведомления управляющих окон редактирования

Окна редактирования посылают оконной процедуре родительского окна сообщения WM_COMMAND. Значения переменных *lParam* и *wParam*, являющихся параметрами этих сообщений такие же, как и для кнопок управления:

Параметр	Описание
LOWORD (*wParam*)	Идентификатор дочернего окна
HIWORD (*wParam*)	Код уведомления
lParam	Описатель дочернего окна

Ниже представлены коды уведомления управляющих окон редактирования:

EN_SETFOCUS	Окно получило фокус ввода
EN_KILLFOCUS	Окно потеряло фокус ввода
EN_CHANGE	Содержимое окна будет меняться
EN_UPDATE	Содержимое окна изменилось
EN_ERRSPACE	Произошло переполнение буфера редактирования
EN_MAXTEXT	Произошло переполнение буфера редактирования при вставке
EN_HSCROLL	На горизонтальной полосе прокрутки был щелчок мышью
EN_VSCROLL	На вертикальной полосе прокрутки был щелчок мышью

В программе POPPAD1 обрабатываются только коды уведомления EN_ERRSPACE и EN_MAXTEXT. При получении этих уведомлений на экран выводится окно сообщений.

Управляющие окна редактирования хранят текст в области памяти программы их родительского окна. Как уже отмечалось ранее, содержимое управляющего окна редактирования ограничено примерно 32 килобайтами.

Использование управляющих окон редактирования

Если вы используете несколько однострочных управляющих окон редактирования на поверхности вашего главного окна, то для передачи фокуса ввода от окна к окну вам понадобится вводить новую оконную процедуру. Вы можете это сделать также, как это делается в программе COLORS1, перехватывая нажатия клавиш <Tab> и <Shift>+<Tab>. (Другой пример введения новой оконной процедуры показан далее в этой главе в программе HEAD.) То как использовать клавишу <Enter> — дело ваше. Вы можете управлять ею также, как клавишей <Tab> или использовать ее как сигнал для программы, что все поля редактирования готовы.

Если вы хотите поместить в редактируемое поле текст, вы можете воспользоваться функцией *SetWindowText*. Для получения текста из окна редактирования используются функции *GetWindowTextLength* и *GetWindowText*. Мы рассмотрим примеры таких возможностей в более поздних версиях программы POPPAD.

Сообщения управляющему окну редактирования

Мы не станем обсуждать все сообщения, которые мы можем послать управляющему окну редактирования с помощью функции *SendMessage*, поскольку их достаточно много, а некоторые будут использоваться в следующих версиях программы POPPAD. Здесь дан общий обзор.

Эти сообщения позволяют вам удалять, копировать или очищать текущую выделенную часть текста. Пользователь выделяет текст для обработки, используя мышь или клавишу <Shift> с нужной клавишей управления курсором, выбранный текст подсвечивается в окне редактирования:

```
SendMessage(hwndEdit, WM_CUT, 0, 0);
SendMessage(hwndEdit, WM_COPY, 0, 0);
SendMessage(hwndEdit, WM_CLEAR, 0, 0);
```

Сообщение WM_CUT удаляет выделенный текст из окна редактирования и посылает его в папку обмена. Сообщение WM_COPY копирует выделенный текст в папку обмена, оставляя его неизменным в окне редактирования. Сообщение WM_CLEAR удаляет выделенный текст из окна редактирования без копирования его в папку обмена.

Вы также можете вставить текст из папки обмена в месте, соответствующем позиции курсора в окне редактирования:

```
SendMessage(hwndEdit, WM_PASTE, 0, 0);
```

Вы можете получить начальное и конечное положения текущего выделения:

```
SendMessage(hwndEdit, EM_GETSEL,(WPARAM) &iStart,(LPARAM) &iEnd);
```

Конечным положением фактически является положение последнего выделенного символа плюс 1.

Вы можете выделить текст:

```
SendMessage(hwndEdit, EM_SETSEL, iStart, iEnd);
```

Вы также можете заменить текущий выделенный текст другим текстом:

```
SendMessage(hwndEdit, EM_REPLACESEL, 0,(LPARAM) szString);
```

Для многострочных окон редактирования вы можете получить число строк:

```
iCount = SendMessage(hwndEdit, EM_GETLINECOUNT, 0, 0);
```

Для любой отдельной строки вы можете получить смещение текста от начала буфера редактирования:

```
iOffset = SendMessage(hwndEdit, EM_LINEINDEX, iLine, 0);
```

Строки нумеруются, начиная с 0. При значении *iLine* равном —1 функция возвращает смещение строки, содержащей курсор. Длину строки можно получить из:

```
iLength = SendMessage(hwndEdit, EM_LINELENGTH, iLine, 0);
```

и копировать саму строку в буфер можно таким образом:

```
iLength = SendMessage(hwndEdit, EM_GETLINE, iLine,(LPARAM) szBuffer);
```

Класс окна списка

Последними из предопределенных в Windows дочерних окон управления, о которых будет рассказано в этой главе, являются окна списков (list box). Список — это набор текстовых строк, который выводится на экран в виде прокручиваемого в прямоугольнике столбца текста. Программа может добавлять или удалять строки в списке путем посылки сообщений оконной процедуре списка. Окно списка посылает сообщения WM_COMMAND своему родительскому окну, когда в списке выбирается какой-либо пункт. Родительское окно может определить, какой пункт списка был выбран.

Чаще всего списки используются в окнах диалога, например, когда окно диалога вызывается при выборе опции Open в меню File. В окне списка отображаются имена файлов текущего каталога, а также другие подкаталоги. Список может быть как с одиночным выбором, так и с множественным. Последний позволяет пользователю выбирать более одного пункта списка. Если окно списка имеет фокус ввода, то один из пунктов списка выводится окруженным штриховой линией. Такая индикация не означает, что данный пункт списка выбран. Выбранный пункт индицируется световым выделением, что означает вывод его в инверсном виде.

В списке с единичным выбором пользователь может выбрать пункт списка, на котором находится курсор, путем нажатия клавиши <Spacebar>. Клавиши управления курсором перемещают как курсор, так и текущую выборку, и с помощью них можно прокручивать содержимое списка. Клавиши <Page Up> и <Page Down> также позволяют прокручивать список, при этом перемещается не выборка, а только курсор. Нажатие буквенной клавиши приводит к перемещению курсора и выборки к первому (или следующему) пункту, который начинается с этой буквы. Пункт списка также можно выбрать путем щелчка или двойного щелчка мыши.

В списке со множественным выбором клавиша <Spacebar> переключает состояние выборки того пункта, на котором находится курсор. (Если пункт уже был выбран, то выборка отменяется.) Клавиши управления курсором

отменяют выборку всех ранее выбранных пунктов и перемещают курсор и выборку точно также, как в случае списка с единичной выборкой. Однако клавиша <Ctrl> совместно с клавишами управления курсором позволяет перемещать курсор без перемещения выборки. Клавиша <Shift> совместно с клавишами управления курсором позволяет расширять выборку.

Щелчок и двойной щелчок мыши на элементе списка с множественной выборкой отменяют все ранее выбранные пункты и выбирают тот пункт списка, на котором был щелчок. Однако щелчок мыши при нажатой клавише <Shift> добавляет в выборку указанный пункт без изменения состояния выборки других пунктов.

Стили окна списка

Дочернее окно списка вы создаете с помощью вызова функции *CreateWindow*, используя имя "listbox" в качестве имени класса окна и WS_CHILD в качестве идентификатора стиля. Однако при этом задаваемом по умолчанию стиле сообщения WM_COMMAND родительскому окну не посылаются. Это означает, что программе следует опрашивать окно списка (посредством сообщений к нему) относительно выбранных в списке пунктов. Поэтому окно списка почти всегда включает идентификатор стиля окна LBS_NOTIFY, что позволяет родительскому окну получать от окна списка сообщения WM_COMMAND. Если вы хотите получить возможность сортировки элементов списка, вам необходимо использовать в окне списка и другой часто используемый идентификатор стиля — LBS_SORT.

По умолчанию, в списке допускается выбор только одного пункта. Если вы хотите создать список с возможностью выборки сразу нескольких пунктов, вам необходимо использовать идентификатор стиля LBS_MULTIPLESEL.

Обычно, если к списку добавляется новый элемент, то окно списка обновляется. Вы можете предотвратить это, если включите стиль LBS_NOREDRAW. Скорее всего вы не захотите использовать этот стиль. Вы можете временно запретить перерисовку окна списка, используя сообщение WM_SETREDRAW, о котором будет рассказано чуть позже.

По умолчанию, оконная процедура окна списка выводит на экран только список элементов без какой-либо рамки вокруг него. Рамку окна вы можете добавить с помощью идентификатора стиля окна WS_BORDER. Для прокрутки содержимого списка с помощью мыши и вертикальной полосы прокрутки используйте идентификатор стиля окна WS_VSCROLL.

В заголовочных файлах Windows определяется стиль окна списка, который называется LBS_STANDART и включает в себя наиболее часто употребляемые стили. Он определяется следующим образом:

```
(LBS_NOTIFY | LBS_SORT | WS_VSCROLL | WS_BORDER)
```

Вы также можете пользоваться идентификаторами WS_SIZEBOX и WS_CAPTION, которые дают возможность менять размер окна списка и перемещать его по рабочей области родительского окна.

Ширина окна списка должна подбираться с учетом ширины самой длинной строки списка плюс ширина полосы прокрутки. Ширину вертикальной полосы прокрутки можно получить с помощью вызова функции:

```
GetSystemMetrics(SM_CXVSCROLL);
```

Высоту окна списка можно рассчитать путем перемножения высоты символа на число пунктов, которые вы одновременно хотите видеть в окне. В окне списка при размещении строк текста величина межстрочного пространства не используется.

Добавление строк в окно списка

После того, как вы создали окно списка, следующим шагом должно стать добавление в список строк текста. Вы делаете это, используя функцию *SendMessage* для отправки сообщения окну списка. Ссылка на строки текста обычно осуществляется через индекс, который начинается с 0, что соответствует самому верхнему элементу списка. В приводимых далее примерах *hwndList* — это описатель дочернего окна управления, т. е. окна списка, а *iIndex* — это значение индекса. В тех случаях, когда вы с помощью вызова функции *SendMessage* передаете строку текста, параметр *lParam* является указателем на эту оканчивающуюся нулем строку.

В большинстве приводимых примеров функция *SendMessage* может вернуть значение LB_ERRSPACE (равное —2), если оконной процедуре не хватит памяти для сохранения содержимого списка. Если случится любая другая ошибка, то возвращаемым значением функции *SendMessage* будет LB_ERR (—1), а при ее нормальной работе — LB_OKAY (0). Вы можете проверять функцию *SendMessage* на 0, чтобы определить наличие каждой из этих двух ошибок.

Если вы используете стиль LBS_SORT (или, если вы располагаете строки в списке в том порядке, в котором вы хотите, чтобы они появлялись), то простейшим способом заполнить список будет использование сообщения LB_ADDSTRING:

```
SendMessage(hwndList, LB_ADDSTRING, 0,(LPARAM) szString);
```

Если вы не используете стиль LBS_SORT, то можете вставить строку в ваш список, задав индекс и используя сообщение LB_INSERTSTRING:

```
SendMessage(hwndList, LB_INSERTSTRING, iIndex,(LPARAM) szString);
```

Например, если *iIndex* равен 4, то *szString* становится новой, пятой строкой, начиная от вершины списка (поскольку счет начинается с 0) со значением индекса, равным 4. Все расположенные ниже строки сдвигаются вниз. При *iIndex* равном —1 строка становится последней строкой списка. Сообщение LB_INSERTSTRING можно использовать и со списком стиля LBS_SORT, но тогда содержимое списка не будет пересортировано. (Вы также можете вставить строку с помощью сообщения LB_DIR, о котором более подробно будет рассказано в конце главы.)

Удалить строку из списка можно с помощью сообщения LB_DELETESTRING, указав значение индекса:

```
SendMessage(hwndList, LB_DELETESTRING, iIndex, 0);
```

Полностью очистить список можно с помощью сообщения LB_RESETCONTENT:

```
SendMessage(hwndList, LB_RESETCONTENT, 0, 0);
```

Оконная процедура окна списка обновляет окно, если к списку добавляется или из списка удаляется элемент. Если вам известно число строк, которые необходимо добавить или удалить, то вам может понадобиться временно приостановить это обновление, сбросив флаг обновления окна:

```
SendMessage(hwndList, WM_SETREDRAW, FALSE, 0);
```

После окончания работы по изменению списка, вам следует восстановить флаг обновления окна:

```
SendMessage(hwndList, WM_SETREDRAW, TRUE, 0);
```

Окно списка, созданное со стилем LBS_NOREDRAW, открывается со сброшенным флагом обновления окна.

Выбор и извлечение элементов списка

Вызовы функции *SendMessage*, которые выполняются в представленных ниже задачах, обычно возвращают какое-то значение. Если имеет место ошибка, то это значение устанавливается в LB_ERR (определено как —1).

После того, как вы вставили в список несколько элементов, вы можете определить количество элементов в списке:

```
iCount = SendMessage(hwndList, LB_GETCOUNT, 0, 0);
```

Некоторые другие вызовы различаются для списков с единичной выборкой и для списков с множественной выборкой. Сначала рассмотрим список с единичной выборкой.

Обычно вы разрешаете пользователю выбирать из списка. Но если вы хотите выделить элемент, выбираемый по умолчанию, то можете использовать такой вызов:

```
SendMessage(hwndList, LB_SETCURSEL, iIndex, 0);
```

При установки в качестве *iIndex* значения —1, выборка для всех элементов отменяется.

Вы также можете выбирать элемент списка на основе его первых символов:

```
iIndex = SendMessage(hwndList, LB_SELECTSTRING, iIndex,(LPARAM) szSearchString);
```

Величина *iIndex*, заданная в качестве параметра *lParam* функции *SendMessage*, является номером пункта, с которого начинается поиск пункта, начальные символы которого заданы в *szSearchString*. При значении *iIndex* равном —1, поиск начинается с начала списка. Возвращаемым значением функции *SendMessage* является индекс выбранного элемента или LB_ERR, если в списке нет элементов, с начальными символами из строки *szSearchString*.

Когда вы получаете от окна списка сообщение WM_COMMAND (или в любое другое время), с помощью сообщения LB_GETCURSEL вы можете определить индекс текущего выбранного элемента:

```
iIndex = SendMessage(hwndList, LB_GETCURSEL, 0, 0);
```

Если возвращаемое значение функции *SendMessage* равно LB_ERR, то это означает отсутствие выбранных элементов.

Вы можете определить длину строки любого элемента списка:

```
iLength = SendMessage(hwndList, LB_GETTEXTLEN, iIndex, 0);
```

Копировать выбранную строку в буфер можно следующим образом:

```
iLength = SendMessage(hwndList, LB_GETTEXT, iIndex,(LPARAM) szBuffer);
```

В обоих случаях возвращаемое значение функции *SendMessage*, т. е. *iLength*, это длина строки. Массив *szBuffer* должен быть достаточно большим, чтобы вместить строку и завершающий NULL-символ. Для предварительного выделения памяти для хранения строки, можно использовать сообщение LB_GETTEXTLEN.

В списке с множественным выбором нельзя использовать сообщения LB_SETCURSEL, LB_GETCURSEL или LB_SELECTSTRING. Вместо них вы используете сообщение LB_SETSEL для установки состояния выборки конкретного элемента, при этом не оказывая влияния на другие элементы, которые могли бы быть выбраны:

```
SendMessage(hwndList, LB_SETSEL, wParam, iIndex);
```

Параметр *wParam* не равен 0 для выбора и выделения элемента списка и равен 0 для отмены выбора. Если параметр *lParam* равен —1, то все элементы списка либо выбираются, либо их выбор отменяется. Определить, выбран или нет конкретный элемент списка, можно с помощью вызова:

```
iSelect = SendMessage(hwndList, LB_GETSEL, iIndex, 0);
```

где *iSelect* не равно нулю, если пункт с номером *iIndex* выбран, и равно 0 — в противном случае.

Получение сообщений от окон списка

Когда пользователь щелкает мышью над окном списка, окно списка получает фокус ввода. Родительское окно может предоставить управляющему (listbox control) окну списка фокус ввода с помощью вызова функции:

```
SetFocus(hwndList);
```

Если окно списка имеет фокус ввода, то для выбора пунктов списка также могут использоваться клавиши управления курсором, буквенные клавиши и клавиша <Spacebar>.

Управляющее окно списка посылает сообщения WM_COMMAND своему родительскому окну. Значение параметров сообщения *lParam* и *wParam* то же, что и для кнопок управления и управляющих окон редактирования:

LOWORD (*wParam*)	Идентификатор дочернего окна
HIWORD (*wParam*)	Код уведомления
lParam	Описатель дочернего окна

Ниже перечисляются коды уведомления и их значения:

LBN_ERRSPACE	-2
LBN_SELCHANGE	1
LBN_DBLCLK	2
LBN_SELCANCEL	3
LBN_SELFOCUS	4
LBN_KILLFOCUS	5

Окно списка посылает своему родительскому окну коды LBN_SELCHANGE и LBN_DBLCLK только в том случае, если в стиль дочернего окна включен идентификатор LBS_NOTIFY.

Код LBN_ERRSPACE показывает, что превышен размер памяти, отведенный для списка. Код LBN_SELCHANGE показывает, что был изменен текущий выбор; эти сообщения имеют место, когда пользователь перемещает подсветку по списку, изменяет состояние выборки с помощью клавиши <Spacebar> или выбирает нужный элемент списка с помощью щелчка мыши. Код LBN_DBLCLK показывает, что на данном пункте списка имел место двойной щелчок мыши. (Значение кодов уведомления для LBN_SELCHANGE и LBN_DBLCLK соответствует количеству щелчков мыши.)

В зависимости от вашего приложения, вы можете использовать либо сообщения LBN_SELCHANGE, либо сообщения LBN_DBLCLK, либо оба вместе. Ваша программа будет получать много сообщений LBN_SELCHANGE, что же касается сообщений LBN_DBLCLK, то они будут случаться только при двойных щелчках мышью. Если в вашей программе используются двойные щелчки мышью, то вам понадобиться обеспечить соответствующий интерфейс клавиатуры для дублирования сообщений LBN_DBLCLK.

Простое приложение, использующее окно списка

Теперь, когда вы знаете как создать список, как заполнить его текстовыми элементами, как получать сообщения от окна списка и как извлекать строки, самое время создать приложение, использующее список. Программа ENVIRON, представленная на рис. 8.8, использует окно списка в рабочей области окна для вывода имен текущих

переменных окружения MS-DOS (таких, как PATH, COMSPEC и PROMPT). При выборе переменной, имя и строка окружения выводятся в верхней части рабочей области.

ENVIRON.MAK

```
#-----------------------
# ENVIRON.MAK make file
#-----------------------

environ.exe : environ.obj
     $(LINKER) $(GUIFLAGS) -OUT:environ.exe environ.obj $(GUILIBS)

environ.obj : environ.c
     $(CC) $(CFLAGS) environ.c
```

ENVIRON.C

```
/*----------------------------------------
   ENVIRON.C -- Environment List Box
               (c) Charles Petzold, 1996
  ----------------------------------------*/

#include <windows.h>
#include <stdlib.h>
#include <string.h>

#define  MAXENV  4096

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Environ";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH)(COLOR_WINDOW + 1);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Environment List Box",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);
     while(GetMessage(&msg, NULL, 0, 0))
          {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
          }
```

```
     return msg.wParam;
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char szBuffer[MAXENV + 1];
     static HWND hwndList, hwndText;
     HDC         hdc;
     int         i;
     TEXTMETRIC  tm;

     switch(iMsg)
          {
          case WM_CREATE :
               hdc = GetDC(hwnd);
               GetTextMetrics(hdc, &tm);
               ReleaseDC(hwnd, hdc);

               hwndList = CreateWindow("listbox", NULL,
                               WS_CHILD | WS_VISIBLE | LBS_STANDARD,
                               tm.tmAveCharWidth, tm.tmHeight * 3,
                               tm.tmAveCharWidth * 16 +
                                    GetSystemMetrics(SM_CXVSCROLL),
                               tm.tmHeight * 5,
                               hwnd,(HMENU) 1,
                              (HINSTANCE) GetWindowLong(hwnd, GWL_HINSTANCE),
                               NULL);

               hwndText = CreateWindow("static", NULL,
                               WS_CHILD | WS_VISIBLE | SS_LEFT,
                               tm.tmAveCharWidth,          tm.tmHeight,
                               tm.tmAveCharWidth * MAXENV, tm.tmHeight,
                               hwnd,(HMENU) 2,
                              (HINSTANCE) GetWindowLong(hwnd, GWL_HINSTANCE),
                               NULL);

               for(i = 0; environ[i]; i++)
                    {
                    if(strlen(environ [i]) > MAXENV)
                         continue;
                    *strchr(strcpy(szBuffer, environ [i]), '=') = '\0';
                    SendMessage(hwndList, LB_ADDSTRING, 0,(LPARAM) szBuffer);
                    }
               return 0;

          case WM_SETFOCUS :
               SetFocus(hwndList);
               return 0;

          case WM_COMMAND :
               if(LOWORD(wParam) == 1 && HIWORD(wParam) == LBN_SELCHANGE)
                    {
                    i = SendMessage(hwndList, LB_GETCURSEL, 0, 0);
                    i = SendMessage(hwndList, LB_GETTEXT, i,
                                             (LPARAM) szBuffer);

                    strcpy(szBuffer + i + 1, getenv(szBuffer));
                    *(szBuffer + i) = '=';

                    SetWindowText(hwndText, szBuffer);
                    }
               return 0;

          case WM_DESTROY :
```

```
            PostQuitMessage(0);
            return 0;
        }
    return DefWindowProc(hwnd, iMsg, wParam, lParam);
    }
```

Рис. 8.8 Программа ENVIRON

В программе ENVIRON создается два дочерних окна: окно списка стиля LBS_STANDART и статическое окно стиля SS_LEFT (текст с выравниванием по левому краю). В программе ENVIRON для получения списка строк окружения используется переменная environ (объявленная как внешняя в STDLIB.H). Программа использует сообщение LB_ADDSTRING для размещения каждой строки в окне списка.

Когда вы запускаете программу ENVIRON, то с помощью мыши или клавиатуры можете выбрать переменную окружения. Каждый раз, когда вы изменяете выбор, окно списка посылает своему родительскому окну сообщение WM_COMMAND, которое попадает в процедуру *WndProc*. Когда *WndProc* получает сообщение WM_COMMAND, она проверяет равно ли младшее слово параметра *lParam* (идентификатор дочернего окна) единице, и равно ли старшее слово параметра *wParam* (код уведомления) величине LBN_SELCHANGE. При наличии соответствующих значений она, с помощью сообщения LB_GETCURSEL, получает индекс выбранного элемента, а с помощью сообщения LB_GETTEXT — сам текст выбранного элемента, т. е. имя переменной окружения. Для получения строки, соответствующей этой переменной окружения, в программе ENVIRON используется функция языка С *getenv*, а функция *SetWindowText* используется для передачи этой строки статическому дочернему окну управления, в котором собственно и выводится на экран содержимое строки.

Обратите внимание, что в программе ENVIRON для индексирования переменной *environ* и получения строки нельзя использовать индекс, возвращаемый сообщением LB_GETCURSEL. Поскольку окно списка имеет стиль LBS_SORT (являющийся частью стиля LBS_STANDART), индексы не совпадают.

Список файлов

Лучшее оставлено напоследок: самым мощным сообщением окон списка является сообщение LB_DIR. Следующий оператор заполняет список перечнем файлов каталога, иногда с подкаталогами и именами доступных дисков:

```
SendMessage(hwndList, LB_DIR, iAttr,(LPARAM) szFileSpec);
```

Использование атрибутов файлов

Параметр *iAttr* — это код атрибута файла. Младший значащий байт — обычный атрибут файла, используемый при вызовах функций MS-DOS:

Параметр *iAttr*	**Значение**	**Атрибут**
DDL_READWRITE	0x0000	Обычный файл
DDL_READONLY	0x0001	Файл только для чтения
DDL_HIDDEN	0x0002	Скрытый файл
DDL_SYSTEM	0x0004	Системный файл
DDL_DIRECTORY	0x0010	Подкаталог
DDL_ARCHIVE	0x0020	Файл с установленным архивным битом

Старший байт обеспечивает некоторый дополнительный контроль:

Параметр *iAttr*	**Значение**	**Опция**
DDL_DRIVES	0x4000	включение имен дисков
DDL_EXCLUSIVE	0x8000	включать только файлы с указанными атрибутами

Когда значение *iAttr* сообщения LB_DIR равно DDL_READWRITE, то в списке перечисляются обычные файлы, файлы только для чтения и файлы с установленным архивным битом. Это соответствует логике, используемой функциями MS-DOS для поиска файлов. Если значение *iAttr* сообщения LB_DIR равно DDL_DIRECTORY, то в список, дополнительно к файлам, включаются имена подкаталогов в квадратных скобках. Если значение *iAttr* сообщения LB_DIR равно DDL_DRIVES | DDL_DIRECTORY, то в список добавляются буквенные идентификаторы всех доступных дисков, при этом они расположены между черточками.

При установке в *iAttr* старшего бита перечисляются только файлы с установленными флагами, обычные файлы не включаются. Для программы резервного копирования файлов, например, можно перечислить только те файлы, которые были изменены после последнего резервирования. В таких файлах установлен архивный бит, поэтому можно использовать DDL_EXCLUSIVE | DDL_ARCHIVE.

Упорядочивание списков файлов

Параметр *lParam* — это указатель на строку, задающую спецификацию файлов, например, "*.*". Такая спецификация файлов не влияет на подкаталоги, которые содержатся в списке.

Сообщение LBS_SORT можно использовать для окна списка файлов. Окно списка будет сначала отображать файлы, удовлетворяющие спецификации, а затем, необязательно, список доступных дисков в виде:

```
[-A-]
```

и (тоже необязательно) имена подкаталогов. Первый подкаталог будет выглядеть следующим образом:

```
[..]
```

Эта "двоеточие" — точка входа в подкаталог на уровень выше в сторону корневого каталога. (Точка входа не появится, если вы перечисляете файлы корневого каталога.) И наконец, указанные имена подкаталогов перечисляются в виде:

```
[SUBDIR]
```

Если вы не используете сообщение LBS_SORT, имена файлов и подкаталогов выводятся вперемешку, а имена доступных дисков оказываются в конце списка.

Утилита Head для Windows

Хорошо известная в операционной системе UNIX утилита "head" (заголовок) выводит на экран начальные строки файла. Давайте используем окно списка для написания аналогичной программы для Windows. В программе HEAD, представленной на рис. 8.9, в списке перечисляются все файлы и подкаталоги. Вы можете выбрать файл для отображения с помощью двойного щелчка мыши на имени файла или, если имя файла выделено, нажав клавишу <Enter>. Вы также можете изменить подкаталог, используя любой из этих способов. Программа HEAD выводит на экран в правую часть рабочей области окна до 8 килобайт информации из начала файла.

HEAD.MAK

```
#--------------------
# HEAD.MAK make file
#--------------------

head.exe : head.obj
     $(LINKER) $(GUIFLAGS) -OUT:head.exe head.obj $(GUILIBS)

head.obj : head.c
     $(CC) $(CFLAGS) head.c
```

HEAD.C

```
/*---------------------------------------------
   HEAD.C -- Displays beginning(head) of file
            (c) Charles Petzold, 1996
  ---------------------------------------------*/

#include <windows.h>
#include <string.h>
#include <direct.h>

#define  MAXPATH     256
#define  MAXREAD    8192

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

WNDPROC fnOldList;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Head";
     HWND        hwnd;
     MSG         msg;
```

```
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH)(COLOR_BTNFACE + 1);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);
     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "File Head",
                         WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static BOOL     bValidFile;
     static char     sReadBuffer[MAXREAD], szFile[MAXPATH];
     static HWND     hwndList, hwndText;
     static OFSTRUCT ofs;
     static RECT     rect;
     char            szBuffer[MAXPATH + 1];
     HDC             hdc;
     int             iHandle, i;
     PAINTSTRUCT     ps;
     TEXTMETRIC      tm;

     switch(iMsg)
          {
          case WM_CREATE :
               hdc = GetDC(hwnd);
               SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));
               GetTextMetrics(hdc, &tm);
               ReleaseDC(hwnd, hdc);

               rect.left = 20 * tm.tmAveCharWidth;
               rect.top  =  3 * tm.tmHeight;

               hwndList = CreateWindow("listbox", NULL,
                              WS_CHILDWINDOW | WS_VISIBLE | LBS_STANDARD,
                              tm.tmAveCharWidth, tm.tmHeight * 3,
                              tm.tmAveCharWidth * 13 +
                                   GetSystemMetrics(SM_CXVSCROLL),
                              tm.tmHeight * 10,
                              hwnd,(HMENU) 1,
```

```
                    (HINSTANCE) GetWindowLong(hwnd, GWL_HINSTANCE),
                    NULL);

          hwndText = CreateWindow("static", getcwd(szBuffer, MAXPATH),
                    WS_CHILDWINDOW | WS_VISIBLE | SS_LEFT,
                    tm.tmAveCharWidth,          tm.tmHeight,
                    tm.tmAveCharWidth * MAXPATH, tm.tmHeight,
                    hwnd,(HMENU) 2,
                    (HINSTANCE) GetWindowLong(hwnd, GWL_HINSTANCE),
                    NULL);

          fnOldList =(WNDPROC) SetWindowLong(hwndList, GWL_WNDPROC,
                                        (LPARAM) ListProc);

          SendMessage(hwndList, LB_DIR, 0x37,(LPARAM) "*.*");
          return 0;

     case WM_SIZE :
          rect.right  = LOWORD(lParam);
          rect.bottom = HIWORD(lParam);
          return 0;

     case WM_SETFOCUS :
          SetFocus(hwndList);
          return 0;

     case WM_COMMAND :
          if(LOWORD(wParam) == 1 && HIWORD(wParam) == LBN_DBLCLK)
               {
               if(LB_ERR ==(i = SendMessage(hwndList,
                                        LB_GETCURSEL, 0, 0L)))
                    break;

               SendMessage(hwndList, LB_GETTEXT, i,(LPARAM) szBuffer);

               if(-1 != OpenFile(szBuffer, &ofs, OF_EXIST | OF_READ))
                    {
                    bValidFile = TRUE;
                    strcpy(szFile, szBuffer);
                    getcwd(szBuffer, MAXPATH);
                    if(szBuffer [strlen(szBuffer) - 1] != '\\')
                         strcat(szBuffer, "\\");
                    SetWindowText(hwndText, strcat(szBuffer, szFile));
                    }
               else
                    {
                    bValidFile = FALSE;
                    szBuffer [strlen(szBuffer) - 1] = '\0';
                    chdir(szBuffer + 1);
                    getcwd(szBuffer, MAXPATH);
                    SetWindowText(hwndText, szBuffer);
                    SendMessage(hwndList, LB_RESETCONTENT, 0, 0L);
                    SendMessage(hwndList, LB_DIR, 0x37,(LONG) "*.*");
                    }
               InvalidateRect(hwnd, NULL, TRUE);
               }
          return 0;

     case WM_PAINT :
          hdc = BeginPaint(hwnd, &ps);
          SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));
          SetTextColor(hdc, GetSysColor(COLOR_BTNTEXT));
          SetBkColor  (hdc, GetSysColor(COLOR_BTNFACE));
```

```
            if(bValidFile && -1 !=(iHandle =
                    OpenFile(szFile, &ofs, OF_REOPEN | OF_READ)))
                {
                i = _lread(iHandle, sReadBuffer, MAXREAD);
                _lclose(iHandle);
                DrawText(hdc, sReadBuffer, i, &rect, DT_WORDBREAK |
                            DT_EXPANDTABS | DT_NOCLIP | DT_NOPREFIX);
                }
            else
                bValidFile = FALSE;

            EndPaint(hwnd, &ps);
            return 0;

        case WM_DESTROY :
            PostQuitMessage(0);
            return 0;
        }
    return DefWindowProc(hwnd, iMsg, wParam, lParam);
    }

LRESULT CALLBACK ListProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
    {
    if(iMsg == WM_KEYDOWN && wParam == VK_RETURN)

        SendMessage(GetParent(hwnd), WM_COMMAND, 1,
                    MAKELONG(hwnd, LBN_DBLCLK));

    return CallWindowProc(fnOldList, hwnd, iMsg, wParam, lParam);
    }
```

Рис. 8.9 Программа HEAD

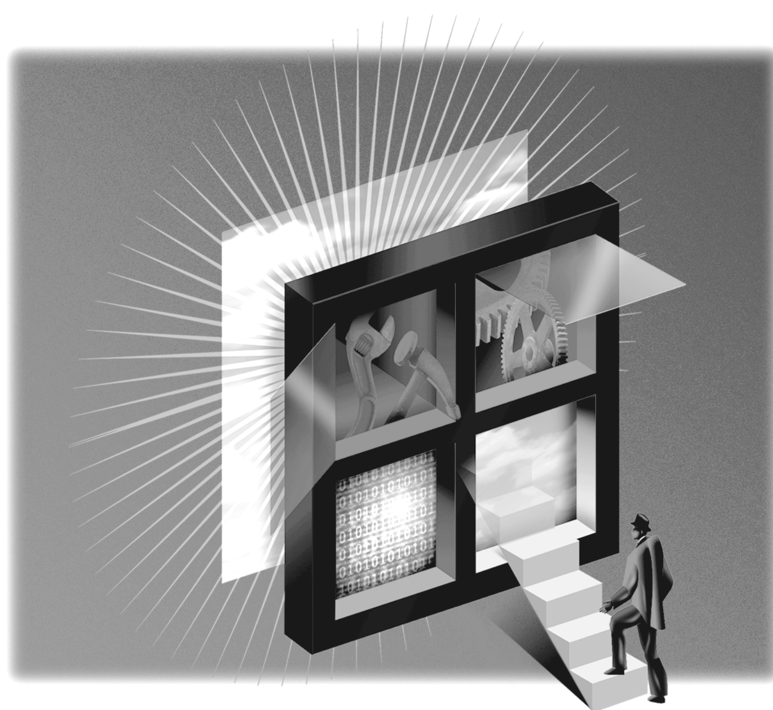
В программе ENVIRON, когда мы выбирали переменную окружения — либо с помощью щелчка мыши, либо с помощью клавиатуры — программа выводила на экран соответствующую строку. Однако, если бы мы использовали аналогичный прием в программе HEAD, программа была бы слишком медленной, поскольку ей пришлось бы долго открывать и закрывать каждый выбранный файл по мере перемещения вашей выборки по списку. Вместо этого в программе HEAD необходим двойной щелчок мыши на имени файла или подкаталога. Это несколько затрудняет задачу, поскольку управляющее окно списка не имеет встроенного интерфейса клавиатуры, который бы соответствовал двойному щелчку мыши. А, как известно, необходимо обеспечивать интерфейс клавиатуры там, где это возможно.

В чем же состоит решение? Конечно, во введении дополнительной оконной процедуры. Новая оконная функция класса списка в программе HEAD называется *ListProc*. Она просто отслеживает сообщение WM_KEYDOWN с параметром *wParam*, равным VK_RETURN, и посылает сообщение WM_COMMAND с кодом уведомления LBN_DBLCLK обратно родительскому окну. При обработке в *WndProc* сообщения WM_COMMAND, для контроля выбранного пункта, используется функция Windows *OpenFile*. Если функция *OpenFile* возвращает ошибку, то значит выбран не файл, а, возможно, подкаталог. Затем для смены подкаталога в программе HEAD используется функция *chdir*. Программа посылает окну списка сообщение LB_RESETCONTEXT для обновления контекста и сообщение LB_DIR для заполнения окна списка перечнем файлов нового подкаталога.

При обработке сообщения WM_PAINT в оконной процедуре открывается файл с использованием функции Windows *OpenFile*. Возвращаемым значением функции является описатель файла MS-DOS, который может быть передан функциям Windows *_lread* и *_lclose*. Содержимое файла выводится на экран с помощью функции *DrawText*.

Часть III

Использование ресурсов



Глава 9 Значки, курсоры, битовые образы и строки

В большинство программ для Windows включаются пользовательские значки, которые Windows выводит на экран в левом верхнем углу строки заголовка окна приложения. Кроме этого Windows выводит на экран значок программы в списках программ меню Start, или в панели задач в нижней части экрана, или в списке программы Windows Explorer. Некоторые программы — наиболее известными из которых являются графические программы для рисования, например Windows Paint, используют собственные курсоры мыши для отражения различных действий программы. В очень многих программах для Windows используются окна меню и диалога. Вместе с полосами прокрутки окна меню и диалога — это основа стандартного пользовательского интерфейса Windows.

Значки, курсоры, окна меню и диалога связаны между собой. Все это виды ресурсов (resources) Windows. Ресурсы являются данными, и они хранятся в .EXE файле программы, но расположены они не в области данных, где обычно хранятся данные исполняемых программ. Таким образом, к ресурсам нет непосредственного доступа через переменные, определенные в исходном тексте программы. Они должны быть явно загружены из файла с расширением .EXE в память.

Когда Windows загружает в память код и данные программы для ее выполнения, она обычно оставляет ресурсы на диске. Только тогда, когда Windows нужен конкретный ресурс, она загружает его в память. Действительно, вы могли обратить внимание на такую динамическую загрузку ресурсов при работе с Windows-программами. Когда вы первый раз вызываете окно диалога программы, Windows обычно обращается к диску для копирования ресурса окна диалога из файла с расширением .EXE программы в оперативную память.

В книге будут рассмотрены следующие ресурсы:

- Значки (icons)
- Курсоры (cursors)
- Битовые образы (bitmaps)
- Символьные строки (character strings)
- Ресурсы, определяемые пользователем (user defined resources)
- Меню (menus)
- Быстрые комбинации клавиш (keyboard accelerators)
- Окна диалога (dialog boxes)

В этой главе рассказывается о первых пяти ресурсах из приведенного списка. О меню и быстрых комбинациях клавиш рассказывается в главе 10, об окнах диалога — в главе 11.

Компиляция ресурсов

При создании программы ресурсы определяются в файле описания ресурсов (resource script), который представляет собой ASCII-файл с расширением .RC. Файл описания ресурсов может содержать представление ресурсов в ASCII-кодах, а также может ссылаться и на другие файлы (ASCII или бинарные файлы), в которых содержатся остальные ресурсы. С помощью компилятора ресурсов (файл RC.EXE) файл описания ресурсов компилируется и становится бинарным файлом с расширением .RES. Задав в командной строке LINK файл с расширением .RES, вы можете заставить компоновщик включить скомпилированный файл описания ресурсов в файл с расширением .EXE программы вместе с обычными кодом и данными программы из файлов с расширением .OBJ и .LIB.

В командной строке можно компилировать файл описания ресурсов с расширением .RC, превращая его, таким образом, в бинарный файл с расширением .RES, путем выполнения команды:

```
RC — r — DWIN32 filename.RC
```

Эта команда создает бинарный файл с именем *filename.RES*. Большинство программистов, пишущих программы для Windows, дают файлу описания ресурсов то же имя, что и самой программе.

В приведенной выше командной строке параметр *-r* заставляет компилятор ресурсов создать файл с расширением .RES и сохранить его на диске. Это необязательно, но делается почти всегда. Параметр -DWIN32 определяет константу (WIN32), которая показывает, что скомпилированный файл описания ресурсов должен храниться в 32-разрядном формате, предназначенном для Windows 95 и Windows NT. Одна из причин определения константы WIN32 состоит в том, что текстовые строки, появляющиеся в окнах меню и диалога, хранятся в формате Unicode, по два байта на каждый символ. Если вы сделаете распечатку файла с расширением .RES в шестнадцатеричном формате, то увидите, что все ASCII-символы в окнах меню и диалога разделены нулями. Не волнуйтесь — это нормально.

До появления Windows 95 компилятор файла описания ресурсов был способен, помимо своей основной задачи, еще и добавлять скомпилированные ресурсы в файл с расширением .EXE, созданный компоновщиком. Компоновщик, поставляемый с Microsoft Visual C++ версии 4.0 для Windows 95 выполняет всю эту работу. Просто укажите файл с расширением .RES рядом с файлом (или файлами) с расширением .OBJ в командной строке LINK.

Процедура компиляции ресурсов отражена в новой секции уже хорошо нам знакомого make-файла. Наряду с компиляцией исходных текстов программы на языке C, вы следующим образом компилируете файл описания ресурсов:

```
progname.res : progname.rc [progname.h] [остальные файлы]
       $(RC) $(RCVARS) progname.rc
```

Макросы RC и RCVARS заданы переменными окружения соответственно как "rc" и "-r -DWIN32". Как уже отмечалось, заголовочный файл .H может быть также определяющим файлом. Действительно, в описание ресурсов вы можете добавить заголовочный файл. Кроме того этот заголовочный файл обычно также включается в файл с исходным текстом программы на C и содержит определения идентификаторов, которые программа использует для ссылки на ресурсы. Также было показано, что в списке зависимостей возможно появление и других файлов. Это те файлы, на которые в описании ресурсов имеется ссылка. Обычно это бинарные файлы со значками, курсорами и битовыми образами.

Компилятор ресурсов RC.EXE использует препроцессор, который может учитывать добавленные и удаленные константы, определять символы ограничителей комментариев /* и */, и директивы препроцессора C *#define, #undef, #ifdef, #ifndef, #include, #if, #elif, #else* и *#endif*. Директива *#include* здесь работает несколько иначе, чем в обычных программах на C. (Более детально мы это исследуем в главе 11.)

Другим изменением make-файла при использовании ресурсов является то, что файл с расширением .RES (наряду со всеми файлами с расширением .OBJ) для целевого файла с расширением .EXE становится определяющим файлом.

Значки и курсоры

Давайте начнем с рассмотрения простой программы, использующей два вида ресурсов — значок и курсор. Программа RESOURC1, приведенная на рис. 9.1, при свертывании программы выводит на экран собственный значок и собственный курсор, когда мышь оказывается в рабочей области окна программы RESOURC1. Программа RESOURC1 также рисует свой значок в нескольких строках и столбцах внутри рабочей области.

RESOURC1.MAK

```
#------------------------
# RESOURC1.MAK make file
#------------------------
resourc1.exe : resourc1.obj resourc1.res
     $(LINKER) $(GUIFLAGS) -OUT:resourc1.exe \
     resourc1.obj resourc1.res $(GUILIBS)

resourc1.obj : resourc1.c
     $(CC) $(CFLAGS) resourc1.c

resourc1.res : resourc1.rc resourc1.ico resourc1.cur
     $(RC) $(RCVARS) resourc1.rc
```

RESOURC1.C

```
/*---------------------------------------------------------
   RESOURC1.C -- Icon and Cursor Demonstration Program No. 1
                 (c) Charles Petzold, 1996
  ---------------------------------------------------------*/
```

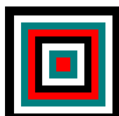
```
#include <windows.h>

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

char      szAppName[] = "Resourc1";
HINSTANCE hInst;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     HWND       hwnd;
     MSG        msg;
     WNDCLASSEX wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(hInstance, szAppName);
     wndclass.hCursor       = LoadCursor(hInstance, szAppName);
     wndclass.hbrBackground =(HBRUSH)(COLOR_WINDOW + 1);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(hInstance, szAppName);

     RegisterClassEx(&wndclass);

     hInst = hInstance;

     hwnd = CreateWindow(szAppName, "Icon and Cursor Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HICON hIcon;
     static int   cxIcon, cyIcon, cxClient, cyClient;
     HDC          hdc;
     PAINTSTRUCT  ps;
     int          x, y;

     switch(iMsg)
          {
          case WM_CREATE :
               hIcon = LoadIcon(hInst, szAppName);
               cxIcon = GetSystemMetrics(SM_CXICON);
               cyIcon = GetSystemMetrics(SM_CYICON);
               return 0;
```

```
          case WM_SIZE :
               cxClient = LOWORD(lParam);
               cyClient = HIWORD(lParam);
               return 0;

          case WM_PAINT :
               hdc = BeginPaint(hwnd, &ps);

               for(y = cyIcon; y < cyClient; y += 2 * cyIcon)
                    for(x = cxIcon; x < cxClient; x += 2 * cxIcon)
                         DrawIcon(hdc, x, y, hIcon);

               EndPaint(hwnd, &ps);
               return 0;

          case WM_DESTROY :
               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

RESOURC1.RC

```
/*-----------------------------
   RESOURC1.RC resource script
  -----------------------------*/

resourc1  ICON    resourc1.ico
resourc1  CURSOR  resourc1.cur
```

RESOURC1.ICO



RESOURC1.CUR

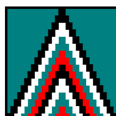


Рис. 9.1 Программа RESOURC1, содержащая курсор и значок

Вы можете создавать значки и курсоры с помощью Microsoft Developer Studio или любой другой интегрированной среды разработки для Windows. Значки и курсоры хранятся в бинарном формате. Файлы значков имеют расширение .ICO, а файлы курсоров — .CUR. Ссылки на эти файлы имеются в файле описания ресурсов RESOURC1.RC.

Редактор изображений

Программу-инструмент, которую вы используете для создания значков, курсоров и битовых образов часто называют редактором изображений (image editor), и он является одной из наиболее важных утилит разработчика в любой интегрированной среде разработки программ для Windows. Как значки, так и курсоры являются разновидностью битовых образов, поэтому будет полезно рассмотреть битовые образы первыми.

Битовый образ (bitmap) — это битовый массив, где один или более битов соответствуют каждому пикселю экрана. В монохромном битовом образе для каждого пикселя требуется один бит. (В простейшем случае 1 соответствует белому цвету, а 0 — черному. Тем не менее, битовые образы часто используются не только для создания простых рисунков, но и в логических операциях.) В цветном битовом образе для отображения цвета каждому пикселю соответствует несколько битов. Редакторы изображений обычно поддерживают создание монохромных и 16-цветных битовых образов. В 16-цветном битовом образе для каждого пикселя требуется 4 бита.

Битовый образ может иметь любое число строк и столбцов, но в некоторых редакторах изображений предельный размер образа, с которым они работают, может быть ограничен. Битовые образы хранятся в файлах с расширением .BMP. Вы тоже можете создавать значки и курсоры с помощью редактора изображений.

Теоретически, Windows выводит значки и курсоры на экран в пиксельном размере, который зависит от разрешающей способности дисплея. Это гарантирует, что они не будут ни слишком большими, ни слишком маленькими. Однако, подходящие для подавляющего большинства видеоадаптеров VGA (Video Graphics Array), с

разрешением экрана 640 пикселей по горизонтали и 480 по вертикали, размеры значков и курсоров изменяются не совсем так, как требуется.

В Windows 95 имеется два размера значков — стандартный и маленький. На дисплеях VGA стандартный значок имеет размер 32 пикселя в ширину и 32 пикселя в высоту, а площадь маленького значка равна 16 квадратных пикселей. Маленький значок используется в левой части строки заголовка приложения для вызова системного меню, в панели задач системы в нижней части экрана, а также в списках программ меню Start. Значки, появляющиеся на рабочем столе, имеют стандартный размер. В программе Windows Explorer и меню Start пользователь может произвольно выбирать стандартный или маленький значок.

В программе можно получить горизонтальный (X) и вертикальный (Y) размеры значков и курсоров, используя функцию *GetSystemMetrics* с параметрами SM_CXICON и SM_CYICON (для стандартного значка), SM_CXSMICON и SM_CYSMICON (для маленького значка) и SM_CXCURSOR и SM_CYCURSOR для курсоров мыши. Для большинства дисплеев размеры стандартных курсоров и значков одинаковы.

Редактор изображений, включенный в состав программы Developer Studio, может создавать файл с расширением .ICO, содержащий один из трех различных образов значка:

- Стандартный: 16-цветный площадью 32 квадратных пикселя
- Монохромный: черно-белый площадью 32 квадратных пикселя
- Маленький: 16-цветный площадью 16 квадратных пикселей

Все курсоры и значки, показанные в программах этой главы, являются монохромными. Вам нет необходимости создавать значки во всех трех форматах. Сама Windows может создать из стандартного значка — маленький, просто исключив каждый второй столбец и строку. Для монохромных дисплеев (которые сейчас почти полностью устарели) Windows может приближенно изобразить значок, используя различные оттенки серого цвета.

Когда вы создаете образ значка в одном из трех представленных форматов, редактор изображений, фактически, сохраняет его в виде двух битовых образов — монохромной маски (mask) битового образа и монохромного или цветного изображения битового образа. Значки всегда прямоугольны, но маска позволяет значку представлять непрямоугольные изображения, т. е. вместо изображения всего значка, некоторые его части могут быть окрашены на экране цветом фона. Кроме этого значки могут содержать области, инвертирующие цвет фона. В следующей таблице показано, как редактор изображений строит два битовых образа, которые описывают монохромный значок:

Цвет:	**Черный**	**Белый**	**Экран**	**Инверсный экран**
Маска битового образа:	0	0	1	1
Изображение битового образа:	0	1	0	1

При выводе значка на экран, Windows сначала использует поразрядную операцию AND экрана и первого битового образа. Пиксели экрана, соответствующие нулевым битам первого битового образа становятся нулевыми, т. е. черными. Пиксели экрана, соответствующие единичным битам остаются без изменения. Эта логика отражена в следующей таблице:

Бит маски	**Пиксели экрана**	
	0	**1**
0	0	0
1	0	1

Далее, Windows выполняет поразрядную операцию исключающего OR изображения битового образа и экрана. 0 во втором битовом образе оставляет пиксель на экране без изменений; 1 во втором битовом образе инвертирует пиксель экрана. Эта логика представлена в следующей таблице:

Бит изображения	**Пиксели экрана**	
	0	**1**
0	0	1
1	1	0

Используя нотацию языка С для этих операций, вывод на экран осуществляется в соответствии со следующей формулой:

```
Display =(Display & Mask) ^ Image
```

Для 16-цветного значка маска битового образа — монохромна и формируется указанным выше способом. Битовый образ изображения содержит 4 бита на пиксель для отображения 16 цветов. Все четыре бита устанавливаются в 1 для той области значка, в которой цвет фона инвертируется.

Раньше при обсуждении битовых образов говорилось, что 0 необязательно означает черный цвет, а 1 необязательно означает белый. Как вы можете теперь видеть, это зависит от того, как Windows использует битовые образы.

В программе RESOURC1 мы определили класс окна так, чтобы фон рабочей области был COLOR_WINDOW. Вы можете изменить цвет окна с помощью программы Control Panel, чтобы увидеть, как меняются цвета значка и курсора.

Получение описателя значков

В файле описания ресурсов ссылка на файл значка выглядит примерно так:

```
myicon ICON iconfile.ico
```

где ICONFILE.ICO — имя файла значка. В этой инструкции значку присваивается имя "myicon". В программе на С для получения описателя значка используется функция *LoadIcon*. В функции *LoadIcon* имеется два параметра. Первым является описатель экземпляра вашей программы, который в *WinMain* обычно называется *hInstance*. Этот описатель требуется для Windows, чтобы определить, в каком файле с расширением .EXE содержится ресурс значка. Вторым параметром является имя значка из описания ресурсов, заданное в виде указателя на оканчивающуюся нулем строку. Возвращаемым значением функции *LoadIcon* является значение типа HICON, которое определяется в WINDOWS.H.

Имеется связь между именем значка в описании ресурсов и в инструкции, содержащей функцию *LoadIcon* вашей программы на С:

Описание ресурсов:	***myicon*** *ICON iconfile.ico*
Исходный текст программы:	*hIcon = LoadIcon (hInstance,* ***"myicon"****);*

Не беспокойтесь о том, в верхнем или нижнем регистре задано имя значка. Компилятор преобразует в файле описания ресурсов имя значка в символы верхнего регистра и вставляет это имя в таблицу ресурсов в заголовке файла программы с расширением .EXE. При первом вызове функции *LoadIcon*, Windows преобразует строку, заданную вторым параметром в символы верхнего регистра и отыскивает в таблице ресурсов файла с расширением .EXE совпадающее с этой строкой имя.

Вместо имени вы также можете использовать число (16-разрядное беззнаковое WORD). Это число называется идентификатором (ID) значка. Ниже показано как это делается:

Описание ресурсов:	***125*** *ICON iconfile.ico*
Исходный текст программы:	*hIcon = LoadIcon (hInstance, MAKEINTRESOURCE (****125****));*

MAKEINTRESOURCE (make an integer into a resourse string — преобразовать целое в строку ресурса) является макросом, определенным в заголовочных файлах Windows, который преобразует число в указатель, но со старшими 16 разрядами, установленными в нуль. Так Windows узнает, что второй параметр функции *LoadIcon* является числом, а не указателем на символьную строку.

В примерах программ, представленных ранее в этой книге, использовались предопределенные значки:

```
LoadIcon(NULL, IDI_APPLICATION);
```

Поскольку параметр *hInstance* установлен в NULL, Windows узнает, что этот значок является предопределенным. IDI_APPLICATION также определяется в заголовочных файлах Windows с использованием макроса MAKEINTRESOURCE:

```
#define IDI_APPLICATION MAKEINTRESOURCE(32512)
```

Предопределенные значки и курсоры являются частью файла драйвера дисплея.

На имя значка можно ссылаться и с помощью третьего метода, который является комбинацией метода, использующего текстовую строку, и метода, использующего число:

Описание ресурсов:	***125*** *ICON iconfile.ico*
Исходный текст программы:	*hIcon = LoadIcon (hInstance,* ***"#125"****);*

По символу # операционная система Windows определяет, что далее следует число в ASCII-коде.

Как насчет четвертого способа? В этом способе используется макроопределение в заголовочном файле, который включается (с помощью директивы *#include*) и в файл описания ресурсов, и в вашу программу:

Заголовочный файл:	*#define* ***myicon*** *125*
Описание ресурсов:	***myicon*** *ICON iconfile.ico*

Исходный текст программы: *hIcon = LoadIcon (hInstance, MAKEINTRESOURCE (**myicon**));*

При использовании этого способа будьте внимательны! Хотя регистр символов при использовании строки в качестве имени значка не имеет значения, он приводит к различию идентификаторов, генерируемых с помощью инструкций *#define*.

Использование идентификаторов вместо имен значков уменьшает размер файла с расширением .EXE и, вероятно, немного ускоряет работу функции *LoadIcon*. Больше того, если в вашей программе используется множество значков, то вы обнаружите, что проще хранить их идентификаторы в массиве.

Использование значков в вашей программе

Хотя для обозначения программ Windows использует значки несколькими способами, во множестве программ для Windows значок задается только при определении класса окна:

```
wndclass.hIcon = LoadIcon(hInstance, "MyIcon");
.
.
.
wndclass.hIconSm = LoadIcon(hInstance, "MySmIcon");
```

Вы можете в обеих инструкциях сослаться на один и тот же значок стандартного размера, и Windows при выводе маленького значка на экран просто приведет его к необходимому размеру. Если в дальнейшем вы захотите изменить значок программы, это можно сделать с помощью функции *SetClassLong*. Предположим, что у вас в описании ресурсов был второй значок:

```
anothericon ICON iconfil2.ico
```

С помощью следующей инструкции вы можете заменить этот значок значком "myicon":

```
SetClassLong( hwnd, GCL_HICON, LoadIcon(hInstance, "anothericon") );
```

Маленький значок можно заменить с помощью GCL_HICONSM. Если вы сохранили описатель значка, возвращенный функцией *LoadIcon*, то вы также можете и нарисовать значок в рабочей области вашего окна:

```
DrawIcon(hdc, x, y, hIcon);
```

Сама Windows использует функцию *DrawIcon* при выводе значка вашей программы в соответствующее место. Windows получает описатель значка из структуры класса окна. Вы можете получить описатель тем же способом:

```
DrawIcon(hdc, x, y, GetClassLong(hwnd, GCL_HICON));
```

В примере программы RESOURC1 в классе окна используется тот же значок, который выводится в рабочей области окна программы. В файле описания ресурсов значку дается такое же имя, как и программе:

```
resourc1 ICON resourc1.ico
```

Поскольку символьная строка "Resourc1" хранится в массиве *szAppName* и уже используется программой в качестве имени класса окна, функция *LoadIcon* вызывается просто:

```
LoadIcon(hInstance, szAppName);
```

Можно заметить, что функция *LoadIcon* для одного и того же значка вызывается трижды, дважды при определении класса окна в *WinMain* и еще раз при получении описателя значка во время обработки сообщения WM_CREATE в *WndProc*. Троекратный вызов функции *LoadIcon* не создает проблем, поскольку возвращаемым значением функции является один и тот же описатель. Фактически, Windows только однажды загружает значок из файла с расширением .EXE.

Использование альтернативных курсоров

Инструкции для задания курсора в файле описания ресурсов и для получения описателя курсора в вашей программе очень похожи на показанные ранее инструкции для значков:

Описание ресурсов: ***mycursor*** *CURSOR cursfile.cur*

Исходный текст программы: *hCursor = LoadCursor (hInstance, "**mycursor**");*

Другие способы, показанные для значков (использование идентификаторов и MAKEINTRESOURCE), также работают и для курсоров. В заголовочные файлы Windows включается определение *typedef* HCURSOR, которое вы можете использовать для хранения описателя курсора.

Вы можете использовать описатель курсора, полученный при вызове функции *LoadCursor*, при задании поля *hCursor* структуры класса окна:

```
wndclass.hCursor = LoadCursor(hInstance, "mycursor");
```

Это заставляет курсор мыши, если он оказывается в рабочей области вашего окна, превращаться в ваш пользовательский курсор.

Если вы используете дочерние окна, можно сделать так, чтобы курсор выглядел по-разному в разных окнах. Если в вашей программе определяются классы этих дочерних окон, то для каждого класса вы можете использовать свой курсор путем соответствующей установки поля *hCursor* в каждом классе окна. А если вы используете предопределенные дочерние элементы управления, то изменить поле *hCursor* класса окна можно с помощью функции:

```
SetClassLong(hwndChild, GCL_HCURSOR, LoadCursor(hInstance, "childcursor"));
```

Если вы разделяете рабочую область окна вашей программы на маленькие логические области без использования дочерних окон, то для изменения курсора мыши вы можете использовать функцию *SetCursor*:

```
SetCursor(hCursor);
```

Функцию *SetCursor* следует вызывать при обработке сообщения WM_MOUSEMOVE. В противном случае для перерисовки курсора при его движении Windows использует курсор, ранее заданный в классе окна.

В программе RESOURC1 для задания имени курсора используется имя программы:

```
resourc1 CURSOR resourc1.cur
```

Когда в исходном тексте программы RESOURC1.C задается класс окна, переменная *szAppName* используется следующим образом в функции *LoadCursor*:

```
wndclass.hCursor = LoadCursor(hInstance, szAppName);
```

Битовые образы: картинки в пикселях

Мы уже говорили об использовании битовых образов в значках и курсорах. В Windows также включен тип ресурсов с именем BITMAP.

Битовые образы используются в двух главных целях. Первая — рисование на экране картинок. Например, файлы драйверов дисплеев в Windows содержат массу крошечных битовых образов, которые используются для рисования стрелок в полосах прокрутки, галочек в раскрывающихся меню, изображений на кнопках изменения размеров окна, флажках и переключателях. Вы можете использовать битовые образы в меню (как показано в главе 10) и панелях инструментов приложений (как показано в главе 12). В таких программах, как Paint, битовые образы используются для изображения графического меню.

Вторая цель использования битовых образов — создание кистей. Кисти, как вы уже знаете, являются шаблонами пикселей, которые Windows использует для закрашивания изображаемых на экране областей.

Использование битовых образов и кистей

Программа RESOURC2, приведенная на рис. 9.2, является модернизированной версией программы RESOURC1, включающей в себя ресурс монохромного битового образа, используемый для создания кисти фона рабочей области. С помощью редактора изображений был создан битовый образ размерами 8 на 8 пикселей, что является минимальным размером для кисти.

RESOURC2.MAK

```
#------------------------
# RESOURC2.MAK make file
#------------------------

resourc2.exe : resourc2.obj resourc2.res
     $(LINKER) $(GUIFLAGS) -OUT:resourc2.exe \
resourc2.obj resourc2.res $(GUILIBS)
resourc2.obj : resourc2.c
     $(CC) $(CFLAGS) resourc2.c

resourc2.res : resourc2.rc resourc2.ico resourc2.cur resourc2.bmp
     $(RC) $(RCVARS) resourc2.rc
```

RESOURC2.C

```
/*---------------------------------------------------------
   RESOURC2.C -- Icon and Cursor Demonstration Program No. 2
                 (c) Charles Petzold, 1996
```

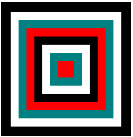
```
  ------------------------------------------------------------*/

#include <windows.h>

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

char      szAppName[] = "Resourc2";
HINSTANCE hInst;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     HBITMAP    hBitmap;
     HBRUSH     hBrush;
     HWND       hwnd;
     MSG        msg;
     WNDCLASSEX wndclass;

     hBitmap = LoadBitmap(hInstance, szAppName);
     hBrush = CreatePatternBrush(hBitmap);

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(hInstance, szAppName);
     wndclass.hCursor       = LoadCursor(hInstance, szAppName);
     wndclass.hbrBackground = hBrush;
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(hInstance, szAppName);

     RegisterClassEx(&wndclass);

     hInst = hInstance;

     hwnd = CreateWindow(szAppName, "Icon and Cursor Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

     DeleteObject((HGDIOBJ) hBrush);       // clean-up
     DeleteObject((HGDIOBJ) hBitmap);

     return msg.wParam;
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HICON hIcon;
     static int   cxIcon, cyIcon, cxClient, cyClient;
     HDC          hdc;
```

```
     PAINTSTRUCT  ps;
     int          x, y;

     switch(iMsg)
          {
          case WM_CREATE :
               hIcon = LoadIcon(hInst, szAppName);
               cxIcon = GetSystemMetrics(SM_CXICON);
               cyIcon = GetSystemMetrics(SM_CYICON);
               return 0;

          case WM_SIZE :
               cxClient = LOWORD(lParam);
               cyClient = HIWORD(lParam);
               return 0;

          case WM_PAINT :
               hdc = BeginPaint(hwnd, &ps);

               for(y = cyIcon; y < cyClient; y += 2 * cyIcon)
                    for(x = cxIcon; x < cxClient; x += 2 * cxIcon)
                         DrawIcon(hdc, x, y, hIcon);

               EndPaint(hwnd, &ps);
               return 0;

          case WM_DESTROY :
               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

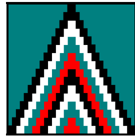
RESOURC2.RC

```
/*-----------------------------
   RESOURC2.RC resource script
  -----------------------------*/

resourc2  ICON    resourc2.ico
resourc2  CURSOR  resourc2.cur
resourc2  BITMAP  resourc2.bmp
```

RESOURC2.ICO



RESOURC2.CUR



RESOURC2.BMP

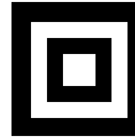


Рис. 9.2 Программа RESOURC2, содержащая значок, курсор и битовый образ

Битовый образ включается в файл описания ресурсов в том же формате, что значок или курсор:

```
resourc2 BITMAP resourc2.bmp
```

Используемая в *WinMain* функция *LoadBitmap* аналогична функциям *LoadIcon* и *LoadCursor*. Ее возвращаемым значением является описатель битового образа:

```
hBitmap = LoadBitmap(hInstance, szAppName);
```

Этот описатель затем используется для создания шаблонной кисти (pattern brush). Основой кисти является битовый образ:

```
hBrush = CreatePatternBrush(hBitmap);
```

Когда Windows закрашивает этой кистью область экрана, битовый образ повторяется по горизонтали и вертикали через каждые восемь пикселей. Эта кисть нам нужна для придания фону рабочей области того цвета, который мы подобрали при определении класса окна:

```
wndclass.hbrBackground = hBrush;
```

Главное отличие между битовыми образами и остальными ресурсами в их практической важности и может быть легко выражено так: битовые образы являются объектами GDI. Хороший стиль составления программ рекомендует, что битовые образы должны быть удалены из программы, если в них отпадает нужда, или если программа завершается. В программе RESOURC2 это происходит по завершении работы функции *WinMain*:

```
DeleteObject((HGDIOBJ) hBrush);
DeleteObject((HGDIOBJ) hBitmap);
```

Символьные строки

Наличие ресурса для символьных строк может вначале показаться странным. Тем более, что не было никаких проблем при использовании привычных символьных строк, определенных в качестве переменных непосредственно в теле исходного текста нашей программы.

Ресурсы-символьные строки предназначены главным образом для облегчения перевода вашей программы на другие языки. Как будет рассказано в следующих двух главах, меню и окна диалога также являются частью описания ресурсов. Если вместо непосредственного использования строк в исходном тексте вашей программы, вы используете ресурсы-символьные строки, то весь текст вашей программы, окажется в одном файле — файле описания ресурсов. Если текст в файле описания ресурсов переводится, то все, что вам нужно сделать для иноязычной версии вашей программы, это перекомпоновать программу и добавить переведенные ресурсы в файл с расширением .EXE. Этот способ намного безопасней, чем возня с исходными кодами вашей программы. (Конечно, можно определить все символьные строки в качестве макросов и хранить их в заголовочном файле. Такой способ также позволяет избежать изменения исходного кода программы при переводе на другие языки.)

Использование ресурсов-символьных строк

Ресурсы-символьные строки определяются в описании ресурсов с помощью ключевого слова STRINGTABLE:

```
STRINGTABLE
{
	id1, "character string 1"
	id2, "character string 2"
	[определения остальных строк]
}
```

В описании ресурсов может содержаться только одна таблица строк. Максимальный размер каждой строки — 255 символов. В строке не может быть управляющих символов языка С, за исключением \t (табуляция). Однако, символьные строки могут содержать восьмеричные константы:

Табуляция (Tab)	\011
Перевод строки (Linefeed)	\012
Возврат каретки (Carriage return)	\015

Эти управляющие символы распознаются функциями *DrawText* и *MessageBox*.

Вы можете использовать функцию *LoadString* для копирования строки из ресурса в буфер в сегменте данных вашей программы:

```
LoadString(hInstance, id, szBuffer, iMaxLength);
```

Параметр *id* соответствует идентификатору, который предшествует каждой строке в файле описания ресурсов; *szBuffer* — это указатель на символьный массив, в который заносится символьная строка; *iMaxLength* — это максимальное число передаваемых в *szBuffer* символов. Идентификаторы строк, которые предшествуют каждой строке, обычно являются идентификаторами макроопределений, которые задаются в заголовочном файле. Многие программисты, программирующие под Windows, для идентификаторов строк используют префикс IDS_. Иногда, при выводе на экран имени файла или другой информации, она должна быть помещена в строку. В этом случае вы помещаете в строку символы форматирования языка С и используете эту строку в качестве форматирующей в функциях *sprintf* или *wsprintf*.

Использование ресурсов-строк в функции *MessageBox*

Давайте рассмотрим пример программы, в которой для вывода на экран трех сообщений об ошибках в окне сообщений используются три символьные строки. В заголовочном файле, который мы назовем PROGRAM.H, для этих сообщений определяются три идентификатора:

```
#define IDS_FILENOTFOUND    1
```

```
#define IDS_FILETOOBIG      2
#define IDS_FILEREADONLY    3
```

Файл описания ресурсов выглядит следующим образом:

```
#include "program.h"

[описание других ресурсов]

STRINGTABLE
{
       IDS_FILENOTFOUND,    "File %s not found."
       IDS_FILETOOBIG,      "File %s too large to edit."
       IDS_FILEREADONLY,    "File %s is read-only."
}
```

Файл с исходным кодом на С также включает этот заголовочный файл и определяет функцию для вывода на экран окна сообщений. (Предполагается, что *szAppName* — это глобальная переменная, в которой содержится имя программы, а *hInst* — это глобальная переменная, в которой содержится описатель экземпляра вашей программы.)

```
#include "program.h"


[другие строки программы]

OkMessage(HWND hwnd, int iErrorNumber, char *szFileName)
{
       char szFormat[40];
       char szBuffer[60];

       LoadString(hInst, iErrorNumber, szFormat, 40);

       sprintf(szBuffer, szFormat, szFileName);

       return MessageBox(hwnd, szBuffer, szAppName, MB_OK | MB_ICONEXCLAMATION);
}
```

Для вывода на экран окна сообщений с сообщением "File not found." программа вызывает функцию:

```
OkMessage(hwnd, IDS_FILENOTFOUND, szFileName);
```

Ресурсы, определяемые пользователем

Ресурсы, определяемые пользователем (user-defined resourse) удобны для включения самых разнообразных данных в ваш файл с расширением .EXE и получения доступа в программе к этим данным. Данные могут содержаться в любом выбранном вами формате — текстовом или бинарном. При загрузке данных в оперативную память возвращаемым значением функций Windows, которые используются для доступа к определяемым пользователем ресурсам, является указатель на данные. С этими данными вы можете делать все, что угодно. Вы вероятно решите, что этот способ хранения и доступа к разнообразным данным удобнее, чем альтернативный, при котором данные хранятся в других файлах и доступ к ним осуществляется через функции файлового ввода.

Например, предположим, что у вас есть файл PROGHELP.TXT, в котором содержится текст "подсказок" для вашей программы. Такой файл не должен быть в чистом виде ASCII-файлом: в нем также могут содержаться бинарные данные, например, указатели, которые могли бы помочь при ссылках на различные части этого файла. Следующим образом опишите ссылку на этот файл в вашем файле описания ресурсов:

```
helptext TEXT proghelp.txt
```

Имена *helptext* (имя ресурса) и TEXT (тип ресурса) в этом выражении могут быть любыми. Слово TEXT написано прописными буквами просто, чтобы сделать его похожим на слова ICON, CURSOR и BITMAP. То, что мы здесь делаем, является созданием ресурса вашего собственного типа, который называется TEXT.

В процессе инициализации программы (например, при обработке сообщения WM_CREATE), можно получить описатель этого ресурса:

```
hResource = LoadResource(hInstance, FindResource(hInstance, "TEXT", "helptext"));
```

Переменная *hResource* определяется как имеющая тип HGLOBAL. Несмотря на свое имя, функция *LoadResource* фактически не загружает сразу ресурс в оперативную память. Используемые вместе, так как это было показано, функции *LoadResource* и *FindResource* по существу эквивалентны функциям *LoadIcon* и *LoadCursor*. Фактически, функции *LoadIcon* и *LoadCursor* используют функции *LoadResource* и *FindResource*.

Вместо имен и типов ресурсов можно использовать числа. Числа могут быть преобразованы в дальние указатели при вызове функции *FindResource* с использованием MAKEINTRESOURCE. Числа, используемые в качестве типа ресурса, должны быть больше 255. (Числа от 1 до 9 при вызове функции *FindResource* используются в Windows для существующих типов ресурсов.)

Когда вам необходимо получить доступ к тексту, вызывайте функцию *LockResource*:

```
pHelpText = LockResource(hResource);
```

Функция *LockResource* загружает ресурс в память (если он еще не был загружен), и возвращает указатель на него. После окончания работы с этим ресурсом, вы можете освободить оперативную память, вызвав функцию *FreeResource*:

```
FreeResource(hResource);
```

Даже если вы не вызовите функцию *FreeResource*, после завершения работы программы оперативная память все равно будет освобождена.

Давайте рассмотрим пример программы, в которой используется три ресурса — значок, таблица строк и ресурс, определяемый пользователем. В программе POEPOEM, представленной на рис. 9.3, на экран, в рабочую область окна программы, выводится текст поэмы Эдгара Алана По "Annabel Lee". Ресурс, определяемый пользователем — это файл POEPOEM.ASC, в котором находится текст поэмы. Текстовый файл заканчивается символом обратной косой черты (\).

POEPOEM.MAK

```
#-----------------------
# POEPOEM.MAK make file
#-----------------------

poepoem.exe : poepoem.obj poepoem.res
     $(LINKER) $(GUIFLAGS) -OUT:poepoem.exe poepoem.obj poepoem.res $(GUILIBS)

poepoem.obj : poepoem.c poepoem.h
     $(CC) $(CFLAGS) poepoem.c

poepoem.res : poepoem.rc poepoem.ico poepoem.asc poepoem.h
     $(RC) $(RCVARS) poepoem.rc
```

POEPOEM.C

```
/*-------------------------------------------------
   POEPOEM.C -- Demonstrates User-Defined Resource
               (c) Charles Petzold, 1996
  -------------------------------------------------*/

#include <windows.h>
#include "poepoem.h"

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

char      szAppName[10];
char      szCaption[35];
HINSTANCE hInst;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     HWND       hwnd;
     MSG        msg;
     WNDCLASSEX wndclass;

     LoadString(hInstance, IDS_APPNAME, szAppName, sizeof(szAppName));
     LoadString(hInstance, IDS_CAPTION, szCaption, sizeof(szCaption));

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
```

```
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(hInstance, szAppName);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(hInstance, szAppName);

     RegisterClassEx(&wndclass);

     hInst = hInstance;

     hwnd = CreateWindow(szAppName, szCaption,
                         WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char   *pText;
     static HGLOBAL hResource;
     static HWND    hScroll;
     static int     iPosition, cxChar, cyChar, cyClient, iNumLines, xScroll;
     char           szPoemRes[15];
     HDC            hdc;
     PAINTSTRUCT    ps;
     RECT           rect;
     TEXTMETRIC     tm;

     switch(iMsg)
          {
          case WM_CREATE :
               hdc = GetDC(hwnd);
               GetTextMetrics(hdc, &tm);
               cxChar = tm.tmAveCharWidth;
               cyChar = tm.tmHeight + tm.tmExternalLeading;
               ReleaseDC(hwnd, hdc);

               xScroll = GetSystemMetrics(SM_CXVSCROLL);

               hScroll = CreateWindow("scrollbar", NULL,
                              WS_CHILD | WS_VISIBLE | SBS_VERT,
                              0, 0, 0, 0,
                              hwnd,(HMENU) 1, hInst, NULL);
               LoadString(hInst, IDS_POEMRES, szPoemRes, sizeof(szPoemRes));

               hResource = LoadResource(hInst,
                           FindResource(hInst, szPoemRes, "TEXT"));

               pText =(char *) LockResource(hResource);
```

```
          iNumLines = 0;

          while(*pText != '\\' && *pText != '\0')
               {
               if(*pText == '\n')
                    iNumLines ++;
               pText = AnsiNext(pText);
               }
          *pText = '\0';

          SetScrollRange(hScroll, SB_CTL, 0, iNumLines, FALSE);
          SetScrollPos  (hScroll, SB_CTL, 0, FALSE);
          return 0;

     case WM_SIZE :
          MoveWindow(hScroll, LOWORD(lParam) - xScroll, 0,
               xScroll, cyClient = HIWORD(lParam), TRUE);
          SetFocus(hwnd);
          return 0;

     case WM_SETFOCUS :
          SetFocus(hScroll);
          return 0;

     case WM_VSCROLL :
          switch(wParam)
               {
               case SB_TOP :
                    iPosition = 0;
                    break;
               case SB_BOTTOM :
                    iPosition = iNumLines;
                    break;
               case SB_LINEUP :
                    iPosition -= 1;
                    break;
               case SB_LINEDOWN :
                    iPosition += 1;
                    break;
               case SB_PAGEUP :
                    iPosition -= cyClient / cyChar;
                    break;
               case SB_PAGEDOWN :
                    iPosition += cyClient / cyChar;
                    break;
               case SB_THUMBPOSITION :
                    iPosition = LOWORD(lParam);
                    break;
               }
          iPosition = max(0, min(iPosition, iNumLines));

          if(iPosition != GetScrollPos(hScroll, SB_CTL))
               {
               SetScrollPos(hScroll, SB_CTL, iPosition, TRUE);
               InvalidateRect(hwnd, NULL, TRUE);
               }
          return 0;

     case WM_PAINT :
          hdc = BeginPaint(hwnd, &ps);

          pText =(char *) LockResource(hResource);

          GetClientRect(hwnd, &rect);
```

```
               rect.left += cxChar;
               rect.top  += cyChar *(1 - iPosition);
               DrawText(hdc, pText, -1, &rect, DT_EXTERNALLEADING);

               EndPaint(hwnd, &ps);
               return 0;

          case WM_DESTROY :
               FreeResource(hResource);
               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

POEPOEM.RC

```
/*----------------------------
   POEPOEM.RC resource script
  ----------------------------*/

#include "poepoem.h"

poepoem     ICON  poepoem.ico
AnnabelLee  TEXT  poepoem.asc

STRINGTABLE
     {
     IDS_APPNAME, "poepoem"
     IDS_CAPTION, """Annabel Lee"" by Edgar Allen Poe"
     IDS_POEMRES, "AnnabelLee"
     }
```

POEPOEM.ICO

POEPOEM.H

```
/*-----------------------
   POEPOEM.H header file
  -----------------------*/

#define IDS_APPNAME 0
#define IDS_CAPTION 1
#define IDS_POEMRES 2
```

POEPOEM.ASC

```
It was many and many a year ago, In a kingdom by the sea,
That a maiden there lived whom you may know
        By the name of Annabel Lee;
And this maiden she lived with no other thought Than to love and be loved by me.
I was a child and she was a child
        In this kingdom by the sea,
But we loved with a love that was more than love -I and my Annabel Lee --
With a love that the winged seraphs of Heaven Coveted her and me.
And this was the reason that, long ago,
        In this kingdom by the sea,
A wind blew out of a cloud, chilling
        My beautiful Annabel Lee;
So that her highborn kinsmen came And bore her away from me,
To shut her up in a sepulchre
        In this kingdom by the sea.
```

```
The angels, not half so happy in Heaven,
        Went envying her and me --
Yes! that was the reason(as all men know,
        In this kingdom by the sea)
That the wind came out of the cloud by night, Chilling and killing my Annabel Lee.
But our love it was stronger by far than the love Of those who were older than we --
        Of many far wiser than we --
And neither the angels in Heaven above
Nor the demons down under the sea
Can ever dissever my soul from the soul Of the beautiful Annabel Lee:
For the moon never beams, without bringing me dreams Of the beautiful Annabel Lee;
And the stars never rise, but I feel the bright eyes Of the beautiful Annabel Lee:
And so, all the night-tide, I lie down by the side Of my darling -- my darling -- my life and my bride,
         In her sepulchre there by the sea --
         In her tomb by the sounding sea.
                                                                      [May, 1849]
```

Рис. 9.3 Программа POEPOEM, содержащая значок и ресурс, определяемый пользователем

В файле описания ресурсов POEPOEM.RC ресурсу, определяемому пользователем, присваивается тип TEXT и имя *AnnabelLee*:

```
AnnabelLee TEXT poepoem.asc
```

При обработке в *WndProc* сообщения WM_CREATE описатель ресурса получается с использованием функций *LoadResource* и *FindResource*. Захватывается ресурс с помощью функции *LockResource*, а небольшая подпрограмма заменяет символ обратной косой черты (\) в конце файла на 0. Это сделано для адаптации текста к функции *DrawText*, которая позже, при обработке сообщения WM_PAINT, будет отображать его на экране.

Обратите внимание на использование полосы прокрутки-элемента управления вместо полосы прокрутки окна: в полосе прокрутки-элемента управления автоматически поддерживается интерфейс клавиатуры, поэтому в программе POEPOEM не требуется обрабатывать сообщение WM_KEYDOWN.

В программе POEPOEM также используются три символьные строки, идентификаторы которых определяются в заголовочном файле POEPOEM.H. Строки IDS_APPNAME и IDS_CAPTION с помощью функции *LoadString* заносятся в глобальные статические переменные:

```
LoadString(hInstance, IDS_APPNAME, szAppName, sizeof(szAppName));
LoadString(hInstance, IDS_CAPTION, szCaption, sizeof(szCaption));
```

Теперь, когда мы задали все символьные строки программы POEPOEM как ресурсы, мы упростили переводчикам их задачу. Конечно, им все равно придется переводить текст поэмы "Annabel Lee", и есть подозрение, что это будет несколько более трудной задачей.

Глава 10 Меню и быстрые клавиши

10

У американцев есть известная юмореска о магазинчике по продаже сыра. Дело было так: некий малый заходит в сырную лавку и просит сыр определенного сорта. Естественно, такого в лавке не оказывается. Тогда он просит сыр другого сорта, потом еще одного и т. д. (всего он успел спросить о 40 сортах), а в ответ повторяется: нет, нет, нет. В конце концов дело заканчивается стрельбой.

При наличии в магазинчике меню, этого неприятного инцидента можно было бы избежать. Меню представляет собой список доступных опций. Голодному меню говорит о том, чем его может накормить кухня, а, применительно к Windows, меню говорит пользователю о том, какие действия с данным приложением он может выполнить.

Меню — это, вероятно, наиболее важная часть пользовательского интерфейса, который предлагают программы для Windows, а добавление меню к вашей программе — это относительно простая задача программирования для Windows. Вы просто определяете структуру меню в вашем описании ресурсов и присваиваете каждому пункту меню уникальный идентификатор. Вы определяете имя меню в структуре класса окна. Когда пользователь выбирает пункт меню, Windows посылает вашей программе сообщение WM_COMMAND, содержащее этот идентификатор. Мы не собираемся останавливаться на этом простом примере. Одной из самых интересных вещей, которую вы можете сделать с помощью меню, является отображение битовых образов, вместо символьных строк, в качестве пунктов меню. Поэтому, давайте подробно рассмотрим, как это сделать.

В этой главе рассказывается также о "быстрых клавишах" (keyboard accelerators). Это комбинации клавиш, которые используются в основном для дублирования функций меню.

Меню

Строка меню выводится на экране непосредственно под строкой заголовка. Эта строка иногда называется главным меню (main menu) или меню верхнего уровня (top-level menu) программы. Выбор элемента главного меню обычно приводит к вызову другого меню, появляющегося под главным, и которое обычно называют всплывающим меню (popup menu) или подменю (submenu). Вы также можете определить несколько уровней вложенности всплывающих меню: т. е. определенный пункт всплывающего меню может вызвать появление другого всплывающего меню. Иногда, с целью увеличения информативности, вызов каких-то пунктов меню приводит к появлению окон диалога. (Об окнах диалога рассказывается в следующей главе.) В большинстве родительских окон, в левой части строки заголовка, имеется маленький значок программы. Щелчок на этом значке приводит к появлению системного меню, которое фактически является всплывающим меню другого типа.

Пункты всплывающих меню могут быть помечены (checked), при этом слева от текста элемента меню Windows ставится специальная метка или "галочка". Галочки позволяют пользователю узнать о том, какие опции программы выбраны из этого меню. Эти опции могут быть взаимоисключающими. Пункты главного меню помечены быть не могут.

Пункты меню в главном и всплывающих меню могут быть "разрешены" (enabled), "запрещены" (disabled) или "недоступны" (grayed). Слова "активно" (active) и "неактивно" (inactive) иногда используются, как синонимы слов "разрешено" и "запрещено". Пункты меню, помеченные как разрешенные или запрещенные, для пользователя выглядят одинаково, а недоступный пункт меню выводится на экран в сером цвете.

С точки зрения пользователя все пункты меню, разрешенные, запрещенные и недоступные, могут быть выбраны. (Отличие только в том, что при выборе запрещенных или недоступных пунктов меню с приложением ничего не происходит.) Пользователь может щелкнуть клавишей мыши на запрещенном пункте меню, или переместить к

запрещенному пункту меню подсветку, или переключиться на пункт меню с помощью буквенной клавиши, соответствующей этому пункту. Однако, с позиции вашей программы разрешенные, запрещенные и недоступные пункты меню функционируют по-разному. Сообщения WM_COMMAND Windows посылает в программу только для разрешенных пунктов меню. Используйте запрещенные и недоступные пункты меню только для тех опций, которые в данный момент не могут быть выбраны. Для того чтобы пользователь знал, какие опции не могут быть выполнены, делайте их недоступными.

Структура меню

Если вы создаете или изменяете меню программы, полезно разделять главное меню и все всплывающие меню. У главного меню имеется описатель меню, у каждого всплывающего меню внутри главного меню имеется собственный описатель меню, и у системного меню (тоже являющегося всплывающим меню) тоже имеется описатель меню.

Каждый пункт меню определяется тремя характеристиками. Первая характеристика определяет то, что будет отображено в меню. Это либо строка текста, либо битовый образ. Вторая характеристика определяет либо идентификатор, который Windows посылает вашей программе в сообщении WM_COMMAND, либо всплывающее меню, которое Windows выводит на экран, когда пользователь выбирает данный пункт меню. Третья характеристика описывает атрибут пункта меню, включая то, является ли данный пункт запрещенным, недоступным или помеченным.

Шаблон меню

Меню можно создать тремя разными способами. Наиболее обычным (и самым простым) является определение меню в файле описания ресурсов в форме шаблона меню, например:

```
MyMenu MENU
{
        [список элементов меню]
}
```

MyMenu — это имя меню. Вы ссылаетесь на это имя в структуре класса окна. Обычно имя меню такое же, как имя приложения.

Внутри квадратных скобок можно использовать либо инструкцию MENUITEM, либо POPUP. Инструкция MENUITEM имеет следующий формат:

```
MENUITEM "&Text", id [, признаки]
```

Формат инструкции POPUP:

```
POPUP "&Text" [, признаки]
{
        [список элементов меню]
}
```

Вместо фигурных скобок можно использовать ключевые слова BEGIN и END. Текст, выводимый для каждого пункта меню должен быть заключен в парные кавычки. Амперсант (&) вызывает подчеркивание следующего за ним символа при выводе на экран. Windows ищет этот же символ, когда пользователь выбирает элемент меню с использованием клавиши <Alt>. Если не включить амперсант в текст, то в тексте меню не будет подчеркнутых символов, и Windows будет использовать первую букву текста при поиске.

Признаками в операторах MENUITEM и POPUP, которые появляются в списке главного меню, являются следующие:

- GRAYED — Данный пункт меню недоступен и не генерирует сообщений WM_COMMAND. Текст изображается недоступным (серым цветом).
- INACTIVE — Данный пункт меню неактивен и не генерирует сообщений WM_COMMAND. Текст изображается обычным образом.
- MENUBREAK — Данный пункт меню, а также все последующие появляются на экране в новой строке меню.
- HELP — Данный пункт меню, а также все последующие появляются на экране, выравненными по правому краю.

Признаки могут комбинироваться с помощью символа поразрядной операции OR языка C (|), но опции GRAYED и INACTIVE нельзя использовать вместе. Опция MENUBREAK не используется в главном меню, поскольку Windows автоматически создает новую строку главного меню, если окно слишком узкое для размещения всех пунктов меню в одну строку.

Следующие за инструкцией POPUP в главном меню квадратные скобки (или ключевые слова BEGIN и END) ограничивают список пунктов всплывающего меню. При определении всплывающего меню допускаются следующие инструкции:

```
MENUITEM "text", id [, признаки]
```

а также:

```
MENUITEM SEPARATOR
```

а также:

```
POPUP "text" [, признаки]
```

MENUITEM SEPARATOR рисует во всплывающем меню горизонтальную черту. Эта черта часто используется для разделения групп, связанных по смыслу и назначению опций.

Для пунктов всплывающего меню в строке символов можно использовать символ табуляции \t для разделения текста по столбцам. Текст, следующий за символом \t располагается в новом столбце, с достаточным отступом вправо для размещения в первом столбце всплывающего меню самой длинной строки текста. То, как это работает, мы рассмотрим при обсуждении быстрых клавиш ближе к окончанию этой главы. Символ \a выравнивает следующий за ним текст по правому краю. Признаками в инструкциях MENUITEM, относящихся к всплывающим меню, являются следующие:

- CHECKED — Слева от текста появляется метка ("галочка").
- GRAYED — Данный пункт меню недоступен и не генерирует сообщений WM_COMMAND. Текст изображается недоступным (серым цветом).
- INACTIVE — Данный пункт меню неактивен и не генерирует сообщений WM_COMMAND. Текст изображается обычным образом.
- MENUBREAK — Данный пункт меню, а также все последующие появляются на экране в новом столбце.
- MENUBARBREAK — Данный пункт меню, а также все последующие появляются на экране в новом столбце. Столбцы разделяются между собой вертикальной чертой.

Опции GRAYED и INACTIVE нельзя использовать вместе. Опции MENUBREAK и MENUBARBREAK нельзя использовать вместе. Если число пунктов всплывающего меню больше, чем можно разместить в одном столбце, необходимо использовать либо MENUBREAK, либо MENUBARBREAK.

Значения идентификаторов в инструкциях MENUITEM — это числа, которые Windows посылает оконной процедуре в сообщениях меню. Значение идентификатора должно быть уникальным в рамках меню. Вместо чисел может оказаться удобнее использовать идентификаторы, определенные в заголовочном файле. По договоренности эти идентификаторы начинаются с символов IDM (ID for a menu, идентификатор меню).

Ссылки на меню в вашей программе

Во многих приложениях Windows в описании ресурсов имеется только одно меню. Ссылка в программе на это меню имеет место в определении класса окна:

```
wndclass.lpszMenuName = "MyMenu";
```

Часто программисты используют строку имени программы в качестве имени меню, имени класса окна, имени значка программы. Однако, вместо имени для меню можно использовать число (или идентификатор). Тогда описание ресурса выглядело бы так:

```
45 MENU
{
        [определение меню]
}
```

В этом случае, оператор присваивания для поля *lpszMenuName* структуры класса окна может быть либо такой:

```
wndclass.lpszMenuName = MAKEINTRESOURCE(45);
```

либо такой:

```
wndclass.lpszMenuName = "#45";
```

Хотя задание имени меню в классе окна является наиболее обычным способом ссылки на ресурс меню, существуют альтернативные варианты. В приложении для Windows ресурс меню можно загрузить в память с помощью функции *LoadMenu*, которая аналогична функциям *LoadIcon* и *LoadCursor*, описанным в главе 9. Если в описании ресурса меню используете имя, то возвращаемым значением функции *LoadMenu* является описатель меню:

```
hMenu = LoadMenu(hInstance, "MyMenu");
```

При использовании числа функция *LoadMenu* принимает либо такой вид:

```
hMenu = LoadMenu(hInstance, MAKEINTRESOURCE(45));
```

либо такой:

```
hMenu = LoadMenu(hInstance, "#45");
```

Затем этот описатель меню можно указать в качестве девятого параметра функции *CreateWindow*:

```
hwnd = CreateWindow("MyClass", "Window Caption",
                    WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    NULL,
                    hMenu,
                    hInstance,
                    NULL);
```

В этом случае меню, указанное при вызове функции *CreateWindow*, перекрывает любое меню, заданное в классе окна. Меню, задаваемое в классе окна, можно считать заданным по умолчанию меню для окон, созданных на основе данного класса окна, если девятый параметр функции *CreateWindow* установлен в NULL. Поэтому можно использовать разные меню для различных окон, созданных на основе одного и того же класса окна.

Можно также указать NULL для меню при регистрации класса окна и NULL для меню при вызове функции *CreateWindow*, а затем присоединить меню к окну следующим образом:

```
SetMenu(hwnd, hMenu);
```

Такая форма позволяет динамически изменять меню окна. В программе NOPOPUPS, которая приведена далее в этой главе, показан пример, иллюстрирующий этот прием.

Любое меню, связанное с окном, удаляется при удалении окна. Любое несвязанное с окном меню перед завершением работы программы должно быть явно удалено при помощи вызова функции *DestroyMenu*.

Меню и сообщения

Когда пользователь выбирает пункт меню, Windows посылает оконной процедуре несколько различных сообщений. Большинство из этих сообщений могут игнорироваться программой, и просто передаваться *DefWindowProc*. Одним из таких сообщений является сообщение WM_INITMENU, которое имеет следующие параметры:

wParam	lParam
Описатель главного меню	0

Значением параметра *wParam* является описатель главного меню, даже если пользователь выбирает пункт системного меню. В программах для Windows сообщение WM_INITMENU обычно игнорируется. Хотя это сообщение существует для того, чтобы дать вам возможность изменить меню перед тем, как будет выбран пункт меню, возможно, что любые изменения главного меню в этот момент могут привести пользователя в замешательство.

Кроме того, программа получает сообщения WM_MENUSELECT. Если пользователь перемещает курсор мыши по пунктам меню, программа может получить множество сообщений WM_MENUSELECT. Как будет рассказано в главе 12, это полезно при использовании строки состояния, содержащей полное описание опции меню. Данное сообщение имеет следующие параметры:

Младшее слово (LOWORD) wParam	**Старшее слово (HIWORD) wParam**	**lParam**
Выбранный пункт: идентификатор меню или описатель всплывающего меню	Флаги выбора	Описатель меню, содержащего выбранный пункт

Сообщение WM_MENUSELECT — это сообщение для отслеживания перемещения по меню. Младшее слово параметра *wParam* говорит о том, какой пункт меню выбран (подсвечен) в данный момент. В старшем слове параметра *wParam* могут быть комбинации из следующих флагов выбора: MF_GRAYED, MF_DISABLED, MF_CHECKED, MF_BITMAP, MF_POPUP, MF_HELP, MF_SYSMENU и MF_MOUSESELECT. Сообщение WM_MENUSELECT может понадобиться для того, чтобы изменить что-нибудь в рабочей области окна на основе информации о перемещении подсветки по пунктам меню. В большинстве программ это сообщение передается в *DefWindowProc*.

Когда Windows готова вывести на экран всплывающее меню, она посылает оконной процедуре сообщение WM_INITMENUPOPUP со следующими параметрами:

wParam	Младшее слово (LOWORD) lParam	Старшее слово (HIWORD) lParam
Описатель всплывающего меню	Индекс всплывающего меню	Для системного меню 1, в противном случае 0

Это сообщение важно, если необходимо разрешать или запрещать пункты меню перед их выводом на экран. Например, предположим, что программа с помощью команды Paste всплывающего меню может копировать текст из буфера обмена. При получении для этого всплывающего меню сообщения WM_INITMENUPOPUP необходимо определить, имеется ли текст в буфере обмена. Если нет, то необходимо сделать этот пункт меню Paste недоступным. Мы рассмотрим такой пример в программе POPPAD, представленной далее в этой главе.

Самым важным сообщением меню является WM_COMMAND. Это сообщение показывает, что пользователь выбрал разрешенный пункт меню окна. Из главы 8 вы помните, что сообщения WM_COMMAND также посылаются дочерними окнами управления. Если оказывается, что для меню и дочерних окон управления используются одни и те же идентификаторы, то различить их можно с помощью значения параметра *lParam*, который для пункта меню будет равен 0:

	Младшее слово (LOWORD) wParam	Старшее слово (HIWORD) wParam	lParam
Меню:	Идентификатор меню	0	0
Элемент управления:	Идентификатор элемента управления	Код уведомления	Описатель дочернего окна

Сообщение WM_SYSCOMMAND похоже на сообщение WM_COMMAND за исключением того, что сообщение WM_SYSCOMMAND сигнализирует, что пользователь выбрал разрешенный пункт системного меню:

	Младшее слово (LOWORD) wParam	Старшее слово (HIWORD) wParam	lParam
Системное меню:	Идентификатор меню	0	0 (Если сообщение WM_SYSCOMMAND является результатом щелчка мыши, тогда старшее и младшее слово lParam являются, соответственно, экранными координатами X и Y курсора мыши.)

Идентификатор меню показывает, какой пункт системного меню выбран. Для предопределенных пунктов системного меню, четыре младших разряда должны быть замаскированы. Результирующая величина будет одной из следующих: SC_SIZE, SC_MOVE, SC_MINIMIZE, SC_MAXIMIZE, SC_NEXTWINDOW, SC_PREVWINDOW, SC_CLOSE, SC_VSCROLL, SC_HSCROLL, SC_ARRANGE, SC_RESTORE и SC_TASKLIST. Кроме того, младшее слово параметра wParam может быть SC_MOUSEMENU или SC_KEYMENU.

Если вы добавите к системному меню новые пункты, то младшее слово параметра wParam станет идентификатором, который вы определите. Для избежания конфликта с предопределенными идентификаторами меню, используйте значения меньше чем 0xF000. Важно, чтобы обычные сообщения WM_SYSCOMMAND передавались в *DefWindowProc*. Если этого не сделать, то обычные команды системного меню будут отключены.

Последним сообщением, которое мы обсудим, является сообщение WM_MENUCHAR, которое, в действительности, не является собственно сообщением меню. Windows посылает это сообщение оконной процедуре в одном из двух случаев: если пользователь нажимает комбинацию клавиши <Alt> и символьной клавиши, несоответствующей пунктам меню, или при выводе на экран всплывающего меню, если пользователь нажимает символьную клавишу, не соответствующую пункту всплывающего меню. Параметры сообщения WM_MENUCHAR являются следующими:

Младшее слово (LOWORD) wParam	Старшее слово (HIWORD) wParam	lParam
Код ASCII	Код выбора	Описатель меню

Код выбора равен:

- 0 — всплывающее меню не отображается.
- MF_POPUP — отображается всплывающее меню.
- MF_SYSMENU — отображается системное меню.

Как правило, в программах для Windows это сообщение передается *DefWindowProc*, которая обычно возвращает 0 в Windows, после чего Windows издает звуковой сигнал (гудок). Мы рассмотрим использование сообщения WM_MENUCHAR в программе GRAFMENU, которая представлена далее в этой главе.

Образец программы

Рассмотрим простой пример. В программе MENUDEMO, представленной на рис. 10.1, в главном меню имеется пять пунктов — File, Edit, Background, Timer и Help. В каждом из этих пунктов имеется всплывающее меню. В программе MENUDEMO демонстрируется простейший и наиболее частый способ обработки сообщений меню, который подразумевает перехват сообщений WM_COMMAND и анализ младшего слова параметра *wParam*.

MENUDEMO.MAK

```
#------------------------
# MENUDEMO.MAK make file
#------------------------

menudemo.exe : menudemo.obj menudemo.res
     $(LINKER) $(GUIFLAGS) -OUT:menudemo.exe menudemo.obj \
     menudemo.res $(GUILIBS)

menudemo.obj : menudemo.c menudemo.h
     $(CC) $(CFLAGS) menudemo.c

menudemo.res : menudemo.rc menudemo.h
     $(RC) $(RCVARS) menudemo.rc
```

MENUDEMO.C

```
/*-----------------------------------------
   MENUDEMO.C -- Menu Demonstration
                 (c) Charles Petzold, 1996
  -----------------------------------------*/

#include <windows.h>
#include "menudemo.h"

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

char szAppName[] = "MenuDemo";

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
     {
     HWND       hwnd;
     MSG        msg;
     WNDCLASSEX wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = szAppName;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Menu Demonstration",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
```

```
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);
     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int  iColorID[5] = { WHITE_BRUSH,  LTGRAY_BRUSH, GRAY_BRUSH,
                                 DKGRAY_BRUSH, BLACK_BRUSH };
     static int  iSelection = IDM_WHITE;
     HMENU       hMenu;

     switch(iMsg)
          {
          case WM_COMMAND :
               hMenu = GetMenu(hwnd);

               switch(LOWORD(wParam))
                    {
                    case IDM_NEW :
                    case IDM_OPEN :
                    case IDM_SAVE :
                    case IDM_SAVEAS :
                         MessageBeep(0);
                         return 0;

                    case IDM_EXIT :
                         SendMessage(hwnd, WM_CLOSE, 0, 0L);
                         return 0;

                    case IDM_UNDO :
                    case IDM_CUT :
                    case IDM_COPY :
                    case IDM_PASTE :
                    case IDM_DEL :
                         MessageBeep(0);
                         return 0;

                    case IDM_WHITE :          // Note: Logic below
                    case IDM_LTGRAY :         //   assumes that IDM_WHITE
                    case IDM_GRAY :           //   through IDM_BLACK are
                    case IDM_DKGRAY :         //   consecutive numbers in
                    case IDM_BLACK :          //   the order shown here.

                         CheckMenuItem(hMenu, iSelection, MF_UNCHECKED);
                         iSelection = LOWORD(wParam);
                         CheckMenuItem(hMenu, iSelection, MF_CHECKED);

                         SetClassLong(hwnd, GCL_HBRBACKGROUND,
                             (LONG) GetStockObject
                       (iColorID[LOWORD(wParam) - IDM_WHITE]));

                         InvalidateRect(hwnd, NULL, TRUE);
                         return 0;
```

```
                    case IDM_START :
                         if(SetTimer(hwnd, 1, 1000, NULL))
                              {
                              EnableMenuItem(hMenu, IDM_START, MF_GRAYED);
                              EnableMenuItem(hMenu, IDM_STOP,  MF_ENABLED);
                              }
                         return 0;

                    case IDM_STOP :
                         KillTimer(hwnd, 1);
                         EnableMenuItem(hMenu, IDM_START, MF_ENABLED);
                         EnableMenuItem(hMenu, IDM_STOP,  MF_GRAYED);
                         return 0;

                    case IDM_HELP :
                         MessageBox(hwnd, "Help not yet implemented!",
                                      szAppName, MB_ICONEXCLAMATION | MB_OK);
                         return 0;

                    case IDM_ABOUT :
                         MessageBox(hwnd, "Menu Demonstration Program.",
                                      szAppName, MB_ICONINFORMATION | MB_OK);
                         return 0;
                    }
               break;

          case WM_TIMER :
               MessageBeep(0);
               return 0;

          case WM_DESTROY :
               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

MENUDEMO.RC

```
/*-----------------------------
   MENUDEMO.RC resource script
  -----------------------------*/
#include "menudemo.h"

MenuDemo MENU
     {
     POPUP "&File"
          {
          MENUITEM "&New",                   IDM_NEW
          MENUITEM "&Open...",               IDM_OPEN
          MENUITEM "&Save",                  IDM_SAVE
          MENUITEM "Save &As...",            IDM_SAVEAS
          MENUITEM SEPARATOR
          MENUITEM "E&xit",                  IDM_EXIT
          }
     POPUP "&Edit"
          {
          MENUITEM "&Undo",                  IDM_UNDO
          MENUITEM SEPARATOR
          MENUITEM "Cu&t",                   IDM_CUT
          MENUITEM "&Copy",                  IDM_COPY
          MENUITEM "&Paste",                 IDM_PASTE
          MENUITEM "De&lete",                IDM_DEL
          }
     POPUP "&Background"
```

```
        {
        MENUITEM "&White",                  IDM_WHITE, CHECKED
        MENUITEM "&Lt Gray",                IDM_LTGRAY
        MENUITEM "&Gray",                   IDM_GRAY
        MENUITEM "&Dk Gray",                IDM_DKGRAY
        MENUITEM "&Black",                  IDM_BLACK
        }
     POPUP "&Timer"
        {
        MENUITEM "&Start"                   IDM_START
        MENUITEM "S&top"                    IDM_STOP,  GRAYED
        }
     POPUP "&Help"
        {
        MENUITEM "&Help...",                IDM_HELP
        MENUITEM "&About MenuDemo...",      IDM_ABOUT
        }
     }
```

MENUDEMO.H

```
/*------------------------
   MENUDEMO.H header file
  ------------------------*/

#define IDM_NEW      1
#define IDM_OPEN     2
#define IDM_SAVE     3
#define IDM_SAVEAS   4
#define IDM_EXIT     5

#define IDM_UNDO    10
#define IDM_CUT     11
#define IDM_COPY    12
#define IDM_PASTE   13
#define IDM_DEL     14

#define IDM_WHITE   20
#define IDM_LTGRAY  21
#define IDM_GRAY    22
#define IDM_DKGRAY  23
#define IDM_BLACK   24

#define IDM_START   30
#define IDM_STOP    31

#define IDM_HELP    40
#define IDM_ABOUT   41
```

Рис. 10.1 Программа MENUDEMO

Все идентификаторы пунктов меню определяются в заголовочном файле MENUDEMO.H. Этот файл должен быть включен (с помощью инструкции *#include*) в файл описания ресурсов и в файл с исходным текстом программы на С. Идентификаторы начинаются с букв IDM. Значения идентификаторов не обязательно должны быть последовательными. Однако, если в программе эти идентификаторы обрабатываются с помощью операторов switch и case, то запомните, что компилятор С гораздо лучше оптимизирует этот код, если значения идентификаторов будут последовательными.

Программа MENUDEMO просто издает звуковой сигнал (гудок) при получении сообщения WM_COMMAND для большинства пунктов всплывающих меню File и Edit. Во всплывающем меню Background перечисляются пять стандартных кистей, которые программа MENUDEMO может использовать для закрашивания фона. В файле описания ресурсов MENUDEMO.RC пункт меню White (с идентификатором меню IDM_WHITE) устанавливается в состояние CHECKED, что вызывает появление метки около этого элемента. В исходном тексте программы MENUDEMO.C начальное значение *iSelection* устанавливается равным IDM_WHITE.

Пять кистей всплывающего меню Background являются взаимоисключающими. Когда программа MENUDEMO.C получает сообщение WM_COMMAND, в котором младшее слово параметра *wParam* является одним из этих пяти пунктов всплывающего меню Background, она должна удалить метку от ранее выбранного цвета фона и поместить ее к новому цвету фона. Для этого, сначала программа получает описатель этого меню:

```
hMenu = GetMenu(hwnd);
```

Функция *CheckMenuItem* используется для снятия метки отмеченного в данный момент пункта меню:

```
CheckMenuItem(hMenu, iSelection, MF_UNCHECKED);
```

Затем значение *iSelection* устанавливается равным младшему слову параметра *wParam*, и новый цвет фона помечается:

```
iSelection = LOWORD(wParam);
CheckMenuItem(hMenu, iSelection, MF_CHECKED);
```

Цвет фона, заданный в классе окна, заменяется новым, и рабочая область окна делается недействительной. Используя новый цвет фона, Windows обновляет окно.

Во всплывающем меню Timer имеется две опции — Start и Stop. В начальный момент времени опция Stop недоступна (так задано в определении меню файла описания ресурсов). При выборе опции Start, программа MENUDEMO пытается запустить таймер и, если это получается, делает недоступной опцию Start, а опцию Stop доступной:

```
EnableMenuItem(hMenu, IDM_START, MF_GRAYED);
EnableMenuItem(hMenu, IDM_STOP, MF_ENABLED);
```

При получении сообщения WM_COMMAND с младшим словом параметра *wParam* равным IDM_STOP, программа MENUDEMO останавливает таймер, делает опцию Start доступной, а опцию Stop — недоступной:

```
EnableMenuItem(hMenu, IDM_START, MF_ENABLED);
EnableMenuItem(hMenu, IDM_STOP, MF_GRAYED);
```

Обратите внимание, что, пока таймер работает, программа MENUDEMO не может получить сообщение WM_COMMAND с младшим словом параметра *wParam* равным IDM_START. Аналогично, если таймер не работает, то программа MENUDEMO не может получить сообщение WM_COMMAND с младшим словом параметра *wParam* равным IDM_STOP.

Когда программа получает сообщение WM_COMMAND с младшим словом параметра *wParam* равным IDM_ABOUT или IDM_HELP, на экран выводится окно сообщения. (В следующей главе мы поменяем его на окно диалога.)

Когда программа получает сообщение WM_COMMAND с младшим словом параметра *wParam* равным IDM_EXIT, она сама себе посылает сообщение WM_CLOSE. Это то самое сообщение, которое *DefWindowProc* посылает оконной процедуре при получении сообщения WM_SYSCOMMAND с параметром *wParam* равным SC_CLOSE. Ближе к окончанию главы, в программе POPPAD2, мы исследуем этот вопрос более подробно.

Этикет при организации меню

Формат всплывающих меню File и Edit в программе MENUDEMO очень напоминает форматы этих меню в других программах для Windows. Одной из задач Windows является обеспечение пользователя таким интерфейсом, при котором для каждой новой программы не требуется изучение базовых концепций. Этому несомненно помогает то, что меню File и Edit выглядят одинаково в любой программе для Windows, а также то, что для выбора используются одни и те же комбинации символьных клавиш и клавиши <Alt>.

За исключением всплывающих меню File и Edit, остальные пункты меню программ для Windows отличаются от программы к программе. При создании меню вам необходимо изучить существующие программы для Windows и стремиться к поддержанию некоторого стандарта. Конечно, если вы считаете эти другие программы неправильными, и знаете лучший путь их создания, никто не посмеет вас остановить. Кроме этого запомните, что для исправления меню обычно требуется исправить только файл описания ресурсов, а не код программы. Позже, без каких бы то ни было проблем, вы сможете изменить пункты меню.

Хотя в главное меню программы могут быть включены инструкции MENUITEM, лучше этого не делать, поскольку в этом случае слишком велика вероятность их ошибочного выбора. Если вы их все-таки ввели, то для индикации того, что данный пункт меню не вызывает появления всплывающего меню, в конце строки текста ставьте восклицательный знак.

Сложный способ определения меню

Определение меню в файле описания ресурсов — это, как правило, простейший способ добавить меню к окну программы, но этот способ — не единственный. Вы можете обойтись без файла описания ресурсов и, с помощью

вызовов функций *CreateMenu* и *AppendMenu*, создать все меню внутри программы. После завершения определения меню, можно передать описатель меню функции *CreateWindow* или использовать функцию *SetMenu* для установки меню окна.

Теперь о том, как это делается. Возвращаемым значением функции *CreateMenu* является просто описатель нового меню:

```
hMenu = CreateMenu();
```

В исходном состоянии меню не содержит ни одного элемента. Элементы в меню вставляются с помощью функции *AppendMenu*. Вам необходимо получить свой описатель меню для каждого пункта главного меню и для каждого всплывающего меню. Всплывающие меню строятся отдельно, а затем их описатели вставляются в меню верхнего уровня. Программа, приведенная на рис. 10.2, именно таким образом создает меню. Получившееся меню аналогично меню программы MENUDEMO.

```
hMenu = CreateMenu();

hMenuPopup = CreateMenu();

AppendMenu(hMenuPopup, MF_STRING, IDM_NEW, "&New");
AppendMenu(hMenuPopup, MF_STRING, IDM_OPEN, "&Open...");
AppendMenu(hMenuPopup, MF_STRING, IDM_SAVE, "&Save");
AppendMenu(hMenuPopup, MF_STRING, IDM_SAVEAS, "Save &As...");
AppendMenu(hMenuPopup, MF_SEPARATOR, 0,  NULL);
AppendMenu(hMenuPopup, MF_STRING, IDM_EXIT, "E&xit");

AppendMenu(hMenu, MF_POPUP,(UINT) hMenuPopup, "&File");

hMenuPopup = CreateMenu();

AppendMenu(hMenuPopup, MF_STRING, IDM_UNDO, "&Undo");
AppendMenu(hMenuPopup, MF_SEPARATOR, 0,  NULL);
AppendMenu(hMenuPopup, MF_STRING, IDM_CUT, "Cu&t");
AppendMenu(hMenuPopup, MF_STRING, IDM_COPY, "&Copy");
AppendMenu(hMenuPopup, MF_STRING, IDM_PASTE, "&Paste");
AppendMenu(hMenuPopup, MF_STRING, IDM_DEL, "De&lete");

AppendMenu(hMenu, MF_POPUP,(UINT) hMenuPopup, "&Edit");

hMenuPopup = CreateMenu();

AppendMenu(hMenuPopup, MF_STRING | MF_CHECKED, IDM_WHITE, "&White");
AppendMenu(hMenuPopup, MF_STRING, IDM_LTGRAY, "&Lt Gray");
AppendMenu(hMenuPopup, MF_STRING, IDM_GRAY, "&Gray");
AppendMenu(hMenuPopup, MF_STRING, IDM_DKGRAY, "&Dk Gray");
AppendMenu(hMenuPopup, MF_STRING, IDM_BLACK, "&Black");

AppendMenu(hMenu, MF_POPUP,(UINT) hMenuPopup, "&Background");

hMenuPopup = CreateMenu();

AppendMenu(hMenuPopup, MF_STRING, IDM_START, "&Start");
AppendMenu(hMenuPopup, MF_STRING | MF_GRAYED, IDM_STOP, "S&top");
AppendMenu(hMenu, MF_POPUP,(UINT) hMenuPopup, "&Timer");

hMenuPopup = CreateMenu();

AppendMenu(hMenuPopup, MF_STRING, IDM_HELP, "&Help...");
AppendMenu(hMenuPopup, MF_STRING, IDM_ABOUT, "&About MenuDemo...");

AppendMenu(hMenu, MF_POPUP,(UINT) hMenuPopup, "&Help");
```

Рис. 10.2 Код программы, создающий такое же меню, как в программе MENUDEMO, но без использования файла описания ресурсов

Согласитесь, что шаблон меню в файле описания ресурсов проще и понятнее. Не рекомендуется определять меню таким образом, а только показано, как это можно сделать. Несомненно, что вы могли бы существенно уменьшить

размер кода, используя массивы структур, содержащие строки символов, идентификаторы и флаги всех пунктов меню. Действуя таким образом, вы получите преимущества при использовании третьего способа определения меню.

Третий подход к определению меню

Функция *LoadMenuIndirect* получает указатель на структуру типа MENUITEMTEMPLATE, а возвращает описатель меню. Эта функция используется в Windows для создания меню после загрузки из файла описания ресурсов обычного шаблона меню. Если вы — решительный человек, то можете попытаться самостоятельно определить меню таким образом.

Независимые всплывающие меню

Вы также можете создать меню без строки главного меню. Вместо нее можно создать всплывающее меню, которое будет появляться в любой части экрана. Один из подходов состоит в том, что всплывающее меню должно появляться при щелчке правой кнопки мыши. Однако, сами пункты меню по-прежнему должны выбираться левой кнопкой мыши. Как это делается, показано в программе POPMENU, представленной на рис. 10.3.

POPMENU.MAK

```
#-----------------------
# POPMENU.MAK make file
#-----------------------

popmenu.exe : popmenu.obj popmenu.res
     $(LINKER) $(GUIFLAGS) -OUT:popmenu.exe popmenu.obj \
     popmenu.res $(GUILIBS)

popmenu.obj : popmenu.c popmenu.h
     $(CC) $(CFLAGS) popmenu.c

popmenu.res : popmenu.rc popmenu.h
     $(RC) $(RCVARS) popmenu.rc
```

POPMENU.C

```
/*----------------------------------------
   POPMENU.C -- Popup Menu Demonstration
               (c) Charles Petzold, 1996
  ----------------------------------------*/

#include <windows.h>
#include "popmenu.h"

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
char      szAppName[] = "PopMenu";
HINSTANCE hInst;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     HWND       hwnd;
     MSG        msg;
     WNDCLASSEX wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
```

```
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hInst = hInstance;

     hwnd = CreateWindow(szAppName, "Popup Menu Demonstration",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HMENU hMenu;
     static int   iColorID[5] = { WHITE_BRUSH,  LTGRAY_BRUSH, GRAY_BRUSH,
                                  DKGRAY_BRUSH, BLACK_BRUSH };
     static int   iSelection = IDM_WHITE;
     POINT        point;

     switch(iMsg)
          {
          case WM_CREATE :
               hMenu = LoadMenu(hInst, szAppName);
               hMenu = GetSubMenu(hMenu, 0);
               return 0;

          case WM_RBUTTONDOWN :
               point.x = LOWORD(lParam);
               point.y = HIWORD(lParam);
               ClientToScreen(hwnd, &point);

               TrackPopupMenu(hMenu, 0, point.x, point.y, 0, hwnd, NULL);
               return 0;

          case WM_COMMAND :
               switch(LOWORD(wParam))
                    {
                    case IDM_NEW :
                    case IDM_OPEN :
                    case IDM_SAVE :
                    case IDM_SAVEAS :
                    case IDM_UNDO :
                    case IDM_CUT :
                    case IDM_COPY :
                    case IDM_PASTE :
                    case IDM_DEL :
                         MessageBeep(0);
                         return 0;

                    case IDM_WHITE :          // Note: Logic below
                    case IDM_LTGRAY :         //   assumes that IDM_WHITE
                    case IDM_GRAY :           //   through IDM_BLACK are
```

```
               case IDM_DKGRAY :         //   consecutive numbers in
               case IDM_BLACK :          //   the order shown here.

                    CheckMenuItem(hMenu, iSelection, MF_UNCHECKED);
                    iSelection = LOWORD(wParam);
                    CheckMenuItem(hMenu, iSelection, MF_CHECKED);

                    SetClassLong(hwnd, GCL_HBRBACKGROUND,
                        (LONG) GetStockObject
                        (iColorID[LOWORD(wParam) - IDM_WHITE]));

                    InvalidateRect(hwnd, NULL, TRUE);
                    return 0;

               case IDM_ABOUT :
                    MessageBox(hwnd, "Popup Menu Demonstration Program.",
                                szAppName, MB_ICONINFORMATION | MB_OK);
                    return 0;

               case IDM_EXIT :
                    SendMessage(hwnd, WM_CLOSE, 0, 0);
                    return 0;

               case IDM_HELP :
                    MessageBox(hwnd, "Help not yet implemented!",
                                szAppName, MB_ICONEXCLAMATION | MB_OK);
                    return 0;
               }
          break;

     case WM_DESTROY :
          PostQuitMessage(0);
          return 0;
     }
return DefWindowProc(hwnd, iMsg, wParam, lParam);
}
```

POPMENU.RC

```
/*----------------------------
   POPMENU.RC resource script
  ----------------------------*/

#include "popmenu.h"

PopMenu MENU
     {
     POPUP ""
          {
          POPUP "&File"
               {
               MENUITEM "&New",               IDM_NEW
               MENUITEM "&Open...",           IDM_OPEN
               MENUITEM "&Save",              IDM_SAVE
               MENUITEM "Save &As...",        IDM_SAVEAS
               MENUITEM SEPARATOR
               MENUITEM "E&xit",              IDM_EXIT
               }
          POPUP "&Edit"
               {
               MENUITEM "&Undo",              IDM_UNDO
               MENUITEM SEPARATOR
               MENUITEM "Cu&t",               IDM_CUT
               MENUITEM "&Copy",              IDM_COPY
               MENUITEM "&Paste",             IDM_PASTE
```

```
            MENUITEM "De&lete",            IDM_DEL
            }
        POPUP "&Background"
            {
            MENUITEM "&White",             IDM_WHITE, CHECKED
            MENUITEM "&Lt Gray",           IDM_LTGRAY
            MENUITEM "&Gray",              IDM_GRAY
            MENUITEM "&Dk Gray",           IDM_DKGRAY
            MENUITEM "&Black",             IDM_BLACK
            }
        POPUP "&Help"
            {
            MENUITEM "&Help...",           IDM_HELP
            MENUITEM "&About PopMenu...",  IDM_ABOUT
            }
        }
    }
```

POPMENU.H

```
/*-----------------------
   POPMENU.H header file
  -----------------------*/

#define IDM_NEW      1
#define IDM_OPEN     2
#define IDM_SAVE     3
#define IDM_SAVEAS   4
#define IDM_EXIT     5

#define IDM_UNDO    10
#define IDM_CUT     11
#define IDM_COPY    12
#define IDM_PASTE   13
#define IDM_DEL     14

#define IDM_WHITE   20
#define IDM_LTGRAY  21
#define IDM_GRAY    22
#define IDM_DKGRAY  23
#define IDM_BLACK   24

#define IDM_HELP    30
#define IDM_ABOUT   31
```

Рис. 10.3 Программа POPMENU

В файле описания ресурсов POPMENU.RC меню определяется почти так, как оно определяется в файле MENUDEMO.RC. Отличие состоит только в том, что главное меню содержит только один пункт — всплывающее меню, содержащее опции File, Edit, Background и Help.

При обработке в *WndProc* сообщения WM_CREATE программа POPMENU получает описатель этого всплывающего меню:

```
hMenu = LoadMenu(hInst, szAppName);
hMenu = GetSubMenu(hMenu, 0);
```

При обработке сообщения WM_RBUTTONDOWN, программа POPMENU получает положение указателя мыши, преобразует это положение в координаты экрана и передает их функции *TrackPopupMenu*:

```
point.x = LOWORD(lParam);
point.y = HIWORD(lParam);
ClientToScreen(hwnd, &point);

TrackPopupMenu(hMenu, 0, point.x, point.y, 0, hwnd, NULL);
```

Затем Windows выводит на экран всплывающее меню с пунктами File, Edit, Background и Help. Выбор любого из этих пунктов приводит к тому, что вложенное всплывающее окно меню появляется на экране правее выбранной опции. Функции этого меню те же, что и у обычного меню.

Использование системного меню

В родительских окнах, стиль которых содержит идентификатор WS_SYSMENU, имеется зона системного меню в левой части строки заголовка. При желании, можно модифицировать это меню. Например, к системному меню можно добавить собственные команды. Хотя это и не рекомендуется, модификация системного меню — это, как правило, быстрый, но некрасивый способ добавления меню к короткой программе, без определения его в файле описания ресурсов. Здесь имеется только одно ограничение: идентификатор, который вы используете для добавления команды к системному меню должен быть меньше, чем 0xF000. В противном случае он будет конфликтовать с идентификаторами, которые Windows использует для команд обычного системного меню. И запомните: при обработке сообщений WM_SYSCOMMAND для этих новых пунктов меню в вашей оконной процедуре, остальные сообщения WM_SYSCOMMAND вы должны передавать в *DefWindowProc*. Если вы этого не сделаете, то гарантировано запретите все обычные опции системного меню.

Программа POORMENU ("Poor Person's Menu"), представленная на рис. 10.4, добавляет к системному меню разделительную линию и три команды. Последняя из этих команд удаляет добавления.

POORMENU.MAK

```
#-----------------------
# POORMENU.MAK make file
#-----------------------

poormenu.exe : poormenu.obj
     $(LINKER) $(GUIFLAGS) -OUT:poormenu.exe poormenu.obj $(GUILIBS)
poormenu.obj : poormenu.c
     $(CC) $(CFLAGS) poormenu.c
```

POORMENU.C

```
/*-----------------------------------------
   POORMENU.C -- The Poor Person's Menu
                 (c) Charles Petzold, 1996
  -----------------------------------------*/

#include <windows.h>

#define IDM_ABOUT   1
#define IDM_HELP    2
#define IDM_REMOVE  3

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

static char szAppName[] = "PoorMenu";

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     HMENU      hMenu;
     HWND       hwnd;
     MSG        msg;
     WNDCLASSEX wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
```

```
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "The Poor-Person's Menu",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     hMenu = GetSystemMenu(hwnd, FALSE);
     AppendMenu(hMenu, MF_SEPARATOR, 0,          NULL);
     AppendMenu(hMenu, MF_STRING,    IDM_ABOUT,  "About...");
     AppendMenu(hMenu, MF_STRING,    IDM_HELP,   "Help...");
     AppendMenu(hMenu, MF_STRING,    IDM_REMOVE, "Remove Additions");

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     switch(iMsg)
          {
          case WM_SYSCOMMAND :
               switch(LOWORD(wParam))
                    {
                    case IDM_ABOUT :
                         MessageBox(hwnd, "A Poor-Person's Menu Program.",
                                    szAppName, MB_OK | MB_ICONINFORMATION);
                         return 0;

                    case IDM_HELP :
                         MessageBox(hwnd, "Help not yet implemented!",
                                    szAppName, MB_OK | MB_ICONEXCLAMATION);
                         return 0;

                    case IDM_REMOVE :
                         GetSystemMenu(hwnd, TRUE);
                         return 0;
                    }
               break;

          case WM_DESTROY :
               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

Рис. 10.4 Программа POORMENU

Три идентификатора меню определяются в самом начале программы POORMENU.C:

```
#define IDM_ABOUT    1
#define IDM_HELP     2
#define IDM_REMOVE   3
```

После создания окна POORMENU получает описатель системного меню:

```
hMenu = GetSystemMenu(hwnd, FALSE);
```

При первом вызове функции *GetSystemMenu* необходимо установить второй параметр в FALSE для подготовки к модификации меню.

Меню изменяется четырьмя вызовами функции *AppendMenu*:

```
AppendMenu(hMenu, MF_SEPARATOR, 0, NULL);
AppendMenu(hMenu, MF_STRING, IDM_ABOUT, "About...");
AppendMenu(hMenu, MF_STRING, IDM_HELP, "Help...");
AppendMenu(hMenu, MF_STRING, IDM_REMOVE, "Remove Additions");
```

Первый вызов функции *AppendMenu* добавляет разделитель. Выбор пункта меню Remove Additions заставляет программу POORMENU удалить эти добавления, что выполняется просто путем повторного вызова функции *GetSystemMenu* со вторым параметром, установленным в TRUE:

```
GetSystemMenu(hwnd, TRUE);
```

В стандартном системном меню имеются опции Restore, Move, Size, Minimize, Maximize, Close и Switch To. Они генерируют сообщения WM_SYSCOMMAND с параметром *wParam*, равным SC_RESTORE, SC_MOVE, SC_SIZE, SC_MINIMIZE, SC_MAXIMIZE, SC_CLOSE и SC_TASKLIST. Хотя в программах для Windows этого обычно не делается, вы сами можете обработать эти сообщения, а не передавать их в *DefWindowProc*. Вы также можете запретить или удалить из системного меню некоторые из этих стандартных опций, используя описанные ниже способы. В документацию Windows также включено несколько стандартных дополнений к системному меню. В них используются идентификаторы SC_NEXTWINDOW, SC_PREVWINDOW, SC_VSCROLL, SC_HSCROLL и SC_ARRANGE. Как вы, наверное, догадались, они предназначены для добавления этих команд к системному меню в некоторых приложениях.

Изменение меню

Мы уже видели, как функция *AppendMenu* может использоваться для определения меню в целом внутри программы и добавления пунктов к системному меню. До появления Windows 3.0 для выполнения этой работы использовалась функция *ChangeMenu*. Функция *ChangeMenu* была столь многогранной по своим задачам, что была одной из наиболее сложных функций в Windows. В Windows 95 эта функция по-прежнему имеется, но ее задачи распределены между пятью новыми функциями:

- *AppendMenu* — добавляет новый элемент в конец меню.
- *DeleteMenu* — удаляет существующий пункт меню и уничтожает его.
- *InsertMenu* — вставляет в меню новый пункт.
- *ModifyMenu* — изменяет существующий пункт меню.
- *RemoveMenu* — удаляет существующий пункт меню.

Отличие между функциями *DeleteMenu* и *RemoveMenu* весьма важно, если указанный пункт меню является всплывающим меню. Функция *DeleteMenu* уничтожает всплывающее меню, а функция *RemoveMenu* — нет.

Другие команды меню

Для работы с меню имеется еще несколько полезных функций.

Если изменяется пункт главного меню, изменения не произойдет, пока Windows не перерисует строку меню. Вызвав функцию *DrawMenuBar*, можно форсировать эту операцию:

```
DrawMenuBar(hwnd);
```

Обратите внимание, что параметром функции *DrawMenuBar* является описатель окна, а не описатель меню.

Описатель всплывающего меню можно получить с помощью функции *GetSubMenu*:

```
hMenuPopup = GetSubMenu(hMenu, iPosition);
```

где *iPosition* — это индекс (отсчитываемый с 0) всплывающего меню внутри главного меню, которое задается параметром *hMenu*. Затем, полученный описатель всплывающего меню *hMenuPopup* можно использовать в других функциях, например, *AppendMenu*.

Текущее число пунктов главного или всплывающего меню можно получить с помощью функции *GetMenuItemCount*:

```
iCount = GetMenuItemCount(hMenu);
```

Идентификатор меню для пункта всплывающего меню можно получить следующим образом:

```
id = GetMenuItemID(hMenuPopup, iPosition);
```

В программе MENUDEMO было показано, как установить или удалить метку пункта всплывающего меню с помощью функции *CheckMenuItem*:

```
CheckMenuItem(hMenu, id, iCheck);
```

В программе MENUDEMO *hMenu* был описателем главного меню, *id* — идентификатором меню, а значение параметра *iCheck* было равно либо MF_CHECKED, либо MF_UNCHECKED. Если *hMenu* является описателем всплывающего меню, то параметр *id* может стать не идентификатором меню, а индексом положения. Если пользоваться этим индексом удобнее, то в третьем параметре указывается флаг MF_BYPOSITION. Например:

```
CheckMenuItem(hMenu, iPosition, MF_CHECKED | MF_BYPOSITION);
```

Работа функции *EnableMenuItem* похожа на работу функции *CheckMenuItem* за исключением того, что третьим параметром может быть MF_ENABLED, MF_DISABLED или MF_GRAYED. Если используется функция *EnableMenuItem* для пункта главного меню, содержащего всплывающее меню, то в качестве третьего параметра следует использовать идентификатор MF_BYPOSITION, поскольку этот пункт меню не имеет идентификатора меню. Мы рассмотрим пример использования функции *EnableMenuItem* в программе POPPAD, представленной далее в этой главе. Функция *HiliteMenuItem* напоминает функции *CheckMenuItem* и *EnableMenuItem*, но использует идентификаторы MF_HILITE и MF_UNHILITE. Эта функция обеспечивает инверсное изображение, которое Windows использует, когда вы перемещаете указатель от одного из пунктов меню к другому. Обычным приложениям нет необходимости использовать функцию *HiliteMenuItem*.

Что еще нужно сделать с меню? Если вы забыли, какие символьные строки использовались в вашем меню, то освежить память можно следующим образом:

```
iByteCount = GetMenuString(hMenu, id, pString, iMaxCount, iFlag);
```

Параметр *iFlag* равен либо MF_BYCOMMAND (при этом *id* — это идентификатор меню), либо MF_BYPOSITION (при этом *id* — это индекс положения). Функция копирует *iMaxCount* байтов строки символов в *pString* и возвращает число скопированных байтов.

Может быть вы хотите узнать, каковы текущие флаги пункта меню:

```
iFlags = GetMenuState(hMenu, id, iFlag);
```

И снова, параметр *iFlag* равен либо MF_BYCOMMAND, либо MF_BYPOSITION. Возвращаемое значение функции *iFlags* — это комбинация всех текущих флагов. Вы можете определить текущие флаги, проверив *iFlags* с помощью идентификаторов MF_DISABLED, MF_GRAYED, MF_CHECKED, MF_MENUBREAK, MF_MENUBARBREAK и MF_SEPARATOR.

А может быть, вам уже слегка надоело меню. В таком случае, если меню вам больше не нужно, его можно удалить:

```
DestroyMenu(hMenu);
```

Эта функция делает недействительным описатель меню.

Нестандартный подход к меню

Теперь, давайте немного сойдем с проторенной дороги. Вместо создания в вашей программе всплывающих меню, попробуем создать нескольких главных меню и переключаться между ними с помощью вызова функции *SetMenu*. Программа NOPOPUPS, представленная на рис. 10.5, показывает, как это сделать. В этой программе, так же как в программе MENUDEMO, имеются пункты меню File и Edit, но вывод их на экран осуществляется по-другому, в виде главного меню.

NOPOPUPS.MAK

```
#-----------------------
# NOPOPUPS.MAK make file
#-----------------------

nopopups.exe : nopopups.obj nopopups.res
     $(LINKER) $(GUIFLAGS) -OUT:nopopups.exe nopopups.obj \
     nopopups.res $(GUILIBS)

nopopups.obj : nopopups.c nopopups.h
     $(CC) $(CFLAGS) nopopups.c

nopopups.res : nopopups.rc nopopups.h
     $(RC) $(RCVARS) nopopups.rc
```

NOPOPUPS.C

```
/*-----------------------------------------------
   NOPOPUPS.C -- Demonstrates No-Popup Nested Menu
                 (c) Charles Petzold, 1996
  -----------------------------------------------*/

#include <windows.h>
#include "nopopups.h"

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "NoPopUps";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "No-Popup Nested Menu Demonstration",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HMENU hMenuMain, hMenuEdit, hMenuFile;
     HINSTANCE    hInstance;

     switch(iMsg)
          {
          case WM_CREATE :
               hInstance =(HINSTANCE) GetWindowLong(hwnd, GWL_HINSTANCE);

               hMenuMain = LoadMenu(hInstance, "MenuMain");
               hMenuFile = LoadMenu(hInstance, "MenuFile");
```

```
               hMenuEdit = LoadMenu(hInstance, "MenuEdit");

               SetMenu(hwnd, hMenuMain);
               return 0;

          case WM_COMMAND :
               switch(LOWORD(wParam))
                    {
                    case IDM_MAIN :
                         SetMenu(hwnd, hMenuMain);
                         return 0;

                    case IDM_FILE :
                         SetMenu(hwnd, hMenuFile);
                         return 0;
                    case IDM_EDIT :
                         SetMenu(hwnd, hMenuEdit);
                         return 0;

                    case IDM_NEW :
                    case IDM_OPEN :
                    case IDM_SAVE :
                    case IDM_SAVEAS :
                    case IDM_UNDO :
                    case IDM_CUT :
                    case IDM_COPY :
                    case IDM_PASTE :
                    case IDM_DEL :
                         MessageBeep(0);
                         return 0;
                    }
               break;

          case WM_DESTROY :
               SetMenu(hwnd, hMenuMain);
               DestroyMenu(hMenuFile);
               DestroyMenu(hMenuEdit);

               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

NOPOPUPS.RC

```
/*-----------------------------
   NOPOPUPS.RC resource script
  -----------------------------*/

#include "nopopups.h"

MenuMain MENU
     {
     MENUITEM "MAIN:",         0,        INACTIVE
     MENUITEM "&File...",      IDM_FILE
     MENUITEM "&Edit...",      IDM_EDIT
     }

MenuFile MENU
     {
     MENUITEM "FILE:",         0,        INACTIVE
     MENUITEM "&New",          IDM_NEW
     MENUITEM "&Open...",      IDM_OPEN
     MENUITEM "&Save",         IDM_SAVE
```

```
     MENUITEM "Save &As...",   IDM_SAVEAS
     MENUITEM "(&Main)",       IDM_MAIN
     }

MenuEdit MENU
     {
     MENUITEM "EDIT:",         0,        INACTIVE
     MENUITEM "&Undo",         IDM_UNDO
     MENUITEM "Cu&t",          IDM_CUT
     MENUITEM "&Copy",         IDM_COPY
     MENUITEM "&Paste",        IDM_PASTE
     MENUITEM "De&lete",       IDM_DEL
     MENUITEM "(&Main)",       IDM_MAIN
     }
```

NOPOPUPS.H

```
/*------------------------
   NOPOPUPS.H header file
  ------------------------*/

#define IDM_NEW     1
#define IDM_OPEN    2
#define IDM_SAVE    3
#define IDM_SAVEAS  4

#define IDM_UNDO    5
#define IDM_CUT     6
#define IDM_COPY    7
#define IDM_PASTE   8
#define IDM_DEL     9

#define IDM_MAIN    10
#define IDM_EDIT    11
#define IDM_FILE    12
```

Рис. 10.5 Программа NOPOPUPS

В файле описания ресурсов вместо одного имеется три меню. Когда оконная процедура обрабатывает сообщение WM_CREATE, Windows загружает в память ресурс каждого меню:

```
hMenuMain = LoadMenu(hInstance, "MenuMain");
hMenuFile = LoadMenu(hInstance, "MenuFile");
hMenuEdit = LoadMenu(hInstance, "MenuEdit");
```

В начале работы программа выводит на экран главное меню:

```
SetMenu(hwnd, hMenuMain);
```

В главном меню тремя символьными строками представлены три опции "MAIN:", "File..." и "Edit...". Однако, опция "MAIN:" запрещена, поэтому она не вызывает посылку оконной процедуре сообщения WM_COMMAND. Для идентификации себя в качестве подменю, меню File и Edit начинаются с "FILE:" и "EDIT:". Последним пунктом меню File и Edit является текстовая строка "(Main)"; эта опция обозначает возвращение в главное меню. Переключение между этими тремя меню достаточно простое:

```
case WM_COMMAND :
       switch(LOWORD(wParam))
              {
              case IDM_MAIN :
                     SetMenu(hwnd, hMenuMain);
                     return 0;

              case IDM_FILE :
                     SetMenu(hwnd, hMenuFile);
                     return 0;

              case IDM_EDIT :
                     SetMenu(hwnd, hMenuEdit);
                     return 0;
```

```
            [другие строки программы]
            }
    break;
```

При обработке сообщения WM_DESTROY программа NOPOPUPS настраивает меню программы на меню Main и, вызывая функцию *DestroyMenu*, удаляет меню File и Edit. Само меню Main удаляется автоматически при удалении окна программы.

Использование в меню битовых образов

Символьные строки — это не единственный способ вывода на экран пунктов меню. Для этой цели можно также использовать битовые образы. Если вы тотчас с ужасом подумали о меню с изображением папок с файлами, банок с кашей и мусорных корзин, то забудьте о картинках. Лучше подумайте о том, какими полезными могли бы быть битовые образы в программах рисования. Подумайте об использовании разных шрифтов и размеров шрифтов, линий разной ширины, штриховых шаблонов и цветов в ваших меню.

Программа, которую мы собираемся исследовать, называется GRAFMENU (graphics menu, графическое меню). Главное меню показано на рис. 10.6. Увеличенные буквы получены из монохромных битовых образов размером 40 на 16 пикселей и сохранены как файлы типа .BMP, которые были созданы с помощью редактора изображений, входящего в состав Developer Studio. Выбор из меню опции FONT вызывает появление всплывающего меню с тремя опциями — Courier New, Arial и Times New Roman. Это стандартные шрифты Windows типа TrueType, и каждая опция представлена на экране соответствующим ей шрифтом (см. рис. 10.7). Эти битовые образы были созданы в программе с использованием контекста памяти.

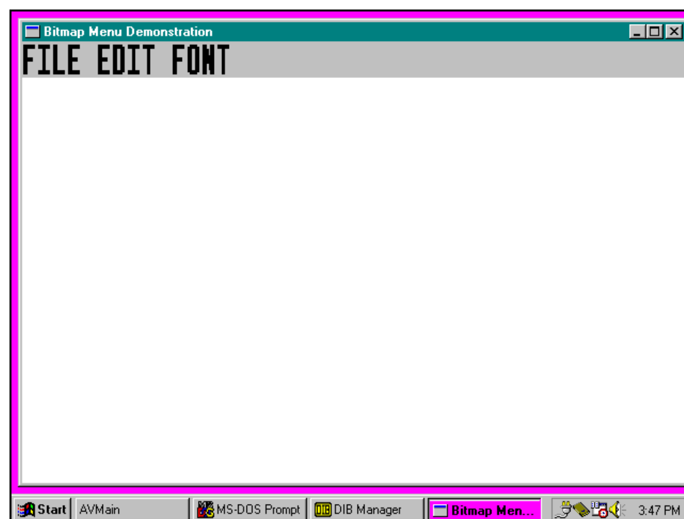


Рис. 10.6 Главное меню программы GRAFMENU

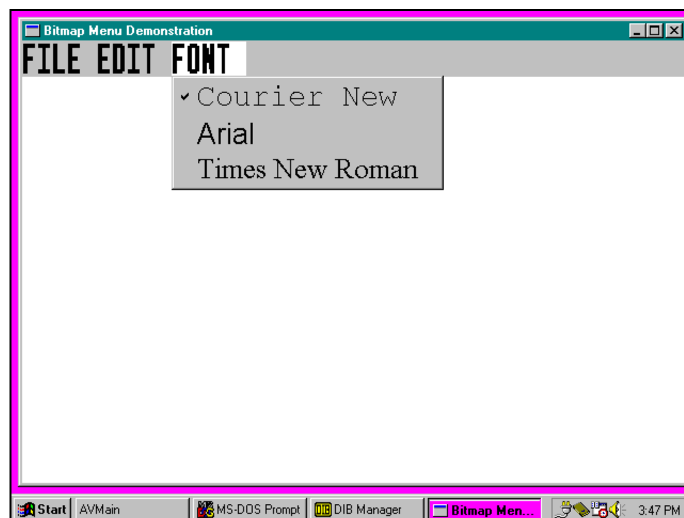


Рис. 10.7 Всплывающее окно меню FONT программы GRAFMENU

И наконец, когда вы открываете системное меню, то получаете доступ к некоторой справочной информации, обозначенной словом "Help", отражающим, вероятно, отчаяние новичка (см. рис. 10.8). Этот монохромный битовый образ размером 64 на 64 пикселя был создан с помощью редактора изображений, входящего в состав Developer Studio.

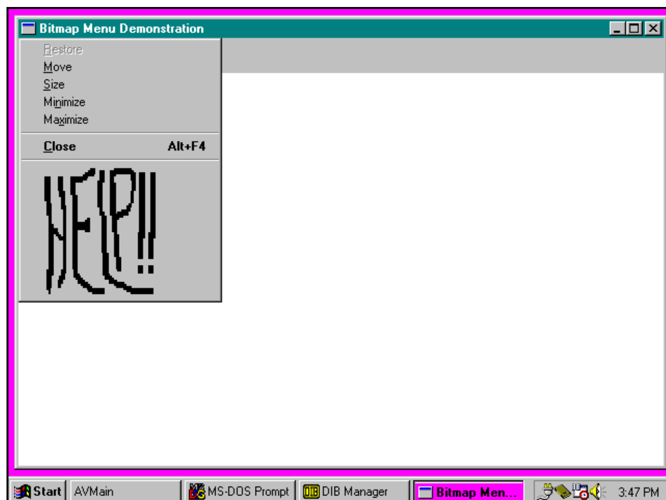


Рис. 10.8 Системное меню программы GRAFMENU

Программа GRAFMENU, а также четыре битовых образа, созданных в редакторе изображений, представлены на рис. 10.9.

GRAFMENU.MAK

```
#------------------------
# GRAFMENU.MAK make file
#------------------------

grafmenu.exe : grafmenu.obj grafmenu.res
     $(LINKER) $(GUIFLAGS) -OUT:grafmenu.exe \
     grafmenu.obj grafmenu.res $(GUILIBS)

grafmenu.obj : grafmenu.c grafmenu.h
     $(CC) $(CFLAGS) grafmenu.c

grafmenu.res : grafmenu.rc grafmenu.h \
     editlabl.bmp filelabl.bmp fontlabl.bmp bighelp.bmp
     $(RC) $(RCVARS) grafmenu.rc
```

GRAFMENU.C

```
/*----------------------------------------------
   GRAFMENU.C -- Demonstrates Bitmap Menu Items
                 (c) Charles Petzold, 1996
  ----------------------------------------------*/
#include <windows.h>
#include <string.h>
#include "grafmenu.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
HBITMAP StretchBitmap(HBITMAP);
HBITMAP GetBitmapFont(int);

char szAppName[] = "GrafMenu";

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     HBITMAP     hBitmapHelp, hBitmapFile, hBitmapEdit,
                 hBitmapFont, hBitmapPopFont[3];
     HMENU       hMenu, hMenuPopup;
```

```
HWND       hwnd;
int        i;
MSG        msg;
WNDCLASSEX wndclass;

wndclass.cbSize        = sizeof(wndclass);
wndclass.style         = CS_HREDRAW | CS_VREDRAW;
wndclass.lpfnWndProc   = WndProc;
wndclass.cbClsExtra    = 0;
wndclass.cbWndExtra    = 0;
wndclass.hInstance     = hInstance;
wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
wndclass.lpszMenuName  = NULL;
wndclass.lpszClassName = szAppName;
wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

RegisterClassEx(&wndclass);

hMenu = CreateMenu();

hMenuPopup = LoadMenu(hInstance, "MenuFile");
hBitmapFile = StretchBitmap(LoadBitmap(hInstance, "BitmapFile"));
AppendMenu(hMenu, MF_BITMAP | MF_POPUP,(int) hMenuPopup,
           (PSTR)(LONG) hBitmapFile);

hMenuPopup = LoadMenu(hInstance, "MenuEdit");
hBitmapEdit = StretchBitmap(LoadBitmap(hInstance, "BitmapEdit"));
AppendMenu(hMenu, MF_BITMAP | MF_POPUP,(int) hMenuPopup,
           (PSTR)(LONG) hBitmapEdit);

hMenuPopup = CreateMenu();
for(i = 0; i < 3; i++)
     {
     hBitmapPopFont[i] = GetBitmapFont(i);
     AppendMenu(hMenuPopup, MF_BITMAP, IDM_COUR + i,
                (PSTR)(LONG) hBitmapPopFont[i]);
     }

hBitmapFont = StretchBitmap(LoadBitmap(hInstance, "BitmapFont"));
AppendMenu(hMenu, MF_BITMAP | MF_POPUP,(int) hMenuPopup,
           (PSTR)(LONG) hBitmapFont);

hwnd = CreateWindow(szAppName, "Bitmap Menu Demonstration",
                    WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    NULL, hMenu, hInstance, NULL);

hMenu = GetSystemMenu(hwnd, FALSE);
hBitmapHelp = StretchBitmap(LoadBitmap(hInstance, "BitmapHelp"));
AppendMenu(hMenu, MF_SEPARATOR, NULL,     NULL);
AppendMenu(hMenu, MF_BITMAP,    IDM_HELP,(PSTR)(LONG) hBitmapHelp);

ShowWindow(hwnd, iCmdShow);
UpdateWindow(hwnd);

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

```
     DeleteObject(hBitmapHelp);
     DeleteObject(hBitmapEdit);
     DeleteObject(hBitmapFile);
     DeleteObject(hBitmapFont);

     for(i = 0; i < 3; i++)
          DeleteObject(hBitmapPopFont[i]);

     return msg.wParam;
     }

HBITMAP StretchBitmap(HBITMAP hBitmap1)
     {
     BITMAP     bm1, bm2;
     HBITMAP    hBitmap2;
     HDC        hdc, hdcMem1, hdcMem2;
     TEXTMETRIC tm;

     hdc = CreateIC("DISPLAY", NULL, NULL, NULL);
     GetTextMetrics(hdc, &tm);
     hdcMem1 = CreateCompatibleDC(hdc);
     hdcMem2 = CreateCompatibleDC(hdc);
     DeleteDC(hdc);

     GetObject(hBitmap1, sizeof(BITMAP),(PSTR) &bm1);

     bm2 = bm1;
     bm2.bmWidth      =(tm.tmAveCharWidth * bm2.bmWidth)  / 4;
     bm2.bmHeight     =(tm.tmHeight       * bm2.bmHeight) / 8;
     bm2.bmWidthBytes =((bm2.bmWidth + 15) / 16) * 2;

     hBitmap2 = CreateBitmapIndirect(&bm2);

     SelectObject(hdcMem1, hBitmap1);
     SelectObject(hdcMem2, hBitmap2);

     StretchBlt(hdcMem2, 0, 0, bm2.bmWidth, bm2.bmHeight,
                hdcMem1, 0, 0, bm1.bmWidth, bm1.bmHeight, SRCCOPY);

     DeleteDC(hdcMem1);
     DeleteDC(hdcMem2);
     DeleteObject(hBitmap1);

     return hBitmap2;
     }

HBITMAP GetBitmapFont(int i)
     {
     static char   *szFaceName[3] = { "Courier New", "Arial",
                                      "Times New Roman" };
     static LOGFONT lf;
     HBITMAP        hBitmap;
     HDC            hdc, hdcMem;
     HFONT          hFont;
     SIZE           size;
     TEXTMETRIC     tm;

     hdc = CreateIC("DISPLAY", NULL, NULL, NULL);
     GetTextMetrics(hdc, &tm);

     lf.lfHeight = 2 * tm.tmHeight;
     strcpy((char *) lf.lfFaceName, szFaceName[i]);

     hdcMem = CreateCompatibleDC(hdc);
```

```
     hFont =(HFONT) SelectObject(hdcMem, CreateFontIndirect(&lf));
     GetTextExtentPoint(hdcMem, szFaceName[i], strlen(szFaceName[i]), &size);

     hBitmap = CreateBitmap(size.cx, size.cy, 1, 1, NULL);
     SelectObject(hdcMem, hBitmap);
     TextOut(hdcMem, 0, 0, szFaceName[i], strlen(szFaceName[i]));

     DeleteObject(SelectObject(hdcMem, hFont));
     DeleteDC(hdcMem);
     DeleteDC(hdc);

     return hBitmap;
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     HMENU      hMenu;
     static int iCurrentFont = IDM_COUR;

     switch(iMsg)
          {
          case WM_CREATE :
               CheckMenuItem(GetMenu(hwnd), iCurrentFont, MF_CHECKED);
               return 0;

          case WM_SYSCOMMAND :
               switch(LOWORD(wParam))
                    {
                    case IDM_HELP :
                         MessageBox(hwnd, "Help not yet implemented!",
                                   szAppName, MB_OK | MB_ICONEXCLAMATION);
                         return 0;
                    }
               break;

          case WM_COMMAND :
               switch(LOWORD(wParam))
                    {
                    case IDM_NEW :
                    case IDM_OPEN :
                    case IDM_SAVE :
                    case IDM_SAVEAS :
                    case IDM_UNDO :
                    case IDM_CUT :
                    case IDM_COPY :
                    case IDM_PASTE :
                    case IDM_DEL :
                         MessageBeep(0);
                         return 0;

                    case IDM_COUR :
                    case IDM_ARIAL :
                    case IDM_TIMES :
                         hMenu = GetMenu(hwnd);
                         CheckMenuItem(hMenu, iCurrentFont, MF_UNCHECKED);
                         iCurrentFont = LOWORD(wParam);
                         CheckMenuItem(hMenu, iCurrentFont, MF_CHECKED);
                         return 0;
                    }
               break;

          case WM_DESTROY :
               PostQuitMessage(0);
               return 0;
```

```
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

GRAFMENU.RC

```
/*-----------------------------
   GRAFMENU.RC resource script
  -----------------------------*/

#include "grafmenu.h"

BitmapEdit BITMAP editlabl.bmp
BitmapFile BITMAP filelabl.bmp
BitmapFont BITMAP fontlabl.bmp
BitmapHelp BITMAP bighelp.bmp

MenuFile MENU
     {
     MENUITEM "&New",        IDM_NEW
     MENUITEM "&Open...",    IDM_OPEN
     MENUITEM "&Save",       IDM_SAVE
     MENUITEM "Save &As...", IDM_SAVEAS
     }

MenuEdit MENU
     {
     MENUITEM "&Undo",       IDM_UNDO
     MENUITEM SEPARATOR
     MENUITEM "Cu&t",        IDM_CUT
     MENUITEM "&Copy",       IDM_COPY
     MENUITEM "&Paste",      IDM_PASTE
     MENUITEM "De&lete",     IDM_DEL
     }
```

GRAFMENU.H

```
/*------------------------
   GRAFMENU.H header file
  ------------------------*/

#define IDM_NEW     1
#define IDM_OPEN    2
#define IDM_SAVE    3
#define IDM_SAVEAS  4

#define IDM_UNDO    5
#define IDM_CUT     6
#define IDM_COPY    7
#define IDM_PASTE   8
#define IDM_DEL     9

#define IDM_COUR    10
#define IDM_ARIAL   11
#define IDM_TIMES   12

#define IDM_HELP    13
```

EDITLABL.BMP

FILELABL.BMP

FONTLABL.BMP

BIGHELP.BMP

Рис. 10.9 Программа GRAFMENU

Два способа создания битовых образов для меню

Чтобы вставить в меню битовый образ используются функции *AppendMenu* и *InsertMenu*. Откуда берется битовый образ? Он может находиться в одном из двух мест. Во-первых, вы можете создать битовый образ с помощью редактора изображений, входящего в состав Developer Studio, и включить файл с битовым образом в описание ресурсов. В программе для загрузки ресурса битового образа в память можно использовать функцию *LoadBitmap*, а для присоединения его к меню — функции *AppendMenu* и *InsertMenu*. Однако, при таком подходе возникает одна проблема. Битовый образ не подойдет ко всем типам разрешения видеомониторов и коэффициентам сжатия; для того чтобы это учесть, необходимо соответствующим образом растянуть или сжать битовый образ. Альтернативой является создание и присоединение битового образа к меню прямо в программе.

Оба эти способа кажутся гораздо более трудными, чем это есть на самом деле. Windows обеспечивает нас функциями, которые позволяют легко работать с битовыми образами при помощи контекста памяти.

Контекст памяти

Когда вы используете вызовы функций GDI (такие как *TextOut*) для записи в рабочую область окна, то фактически запись производится в область памяти (видеопамять), организованную, во многом, как гигантский битовый образ. Ширина и высота этого битового образа равна разрешению дисплея. Способ использования множества битов для определения цветов также определяется видеоадаптером. Если поразмыслить об этом, то окажется, что Windows должна была бы считать определенную область оперативной памяти памятью дисплея и производить запись в эту память точно также, как она пишет на экране. А мы должны были бы использовать эту область памяти в качестве битового образа. Это именно то, чем и является контекст памяти. Он помогает нам рисовать и манипулировать с битовыми образами в программе для Windows. Для этого необходимо сделать следующее:

1. Вызывая функцию *CreateCompatibleDC*, создать контекст памяти. В начальный момент поверхность экрана, соответствующая этому контексту памяти, содержит один монохромный пиксель. Можно считать, что этот контекст памяти имеет ширину и высоту по одному пикселю и два цвета (черный и белый).
2. С помощью функций *CreateBitmap*, *CreateBitmapIndirect* или *CreateCompatibleBitmap* создать неинициализированный битовый образ. При создании битового образа задать его ширину, высоту и цветовую организацию. Однако, пока еще не требуется, чтобы пиксели битового образа что-либо отображали. Сохраните описатель битового образа.
3. С помощью функции *SelectObject* выбрать битовый образ в контекст памяти. Теперь контекст памяти имеет поверхность экрана, которая соответствует размеру битового образа с числом цветов, определенным битовым образом.
4. Использовать функции GDI для рисования в контексте памяти точно так же, как если бы они использовались для рисования в обычном контексте устройства. Все, что рисуется на поверхности экрана контекста памяти, фактически рисуется в битовом образе, выбранном в контекст устройства.
5. Удалить контекст памяти. Остается описатель битового образа, в котором содержится пиксельное представление того, что было нарисовано в контексте памяти.

Создание битового образа, содержащего текст

Функция *GetBitmapFont* в программе GRAFMENU в качестве параметра получает значения 0, 1 или 2, а возвращает описатель битового образа. В этом битовом образе содержится строка "Courier New", "Arial" или "Times New Roman", нарисованная соответствующим шрифтом, почти вдвое большим обычного системного шрифта. Рассмотрим, как работает функция *GetBitmapFont*. (Приводимый далее код не полностью соответствует коду из файла GRAFMENU.C. Для простоты ссылки на массив *szFaceName* заменены соответствующими значениями для шрифта Arial.)

Первым шагом должно быть определение размера системного шрифта с помощью структуры TEXTMETRIC:

```
hdc = CreateIC("DISPLAY", NULL, NULL, NULL);
GetTextMetrics(hdc, &tm);
```

Чтобы сделать больший размер логического шрифта "Arial", необходимо увеличить некоторые значения, полученные из структуры TEXTMETRIC:

```
lf.lfHeight = 2 * tm.tmHeight;
strcpy((char *) lf.lfFaceName, "Arial");
```

Следующим шагом должно быть получение контекста устройства для экрана и создание совместимого с экраном контекста памяти:

```
hdcMem = CreateCompatibleDC(hdc);
```

Описателем контекста памяти является *hdcMem*. На следующем шаге создается шрифт на основе модифицированной структуры *lf*, и этот шрифт выбирается в контекст памяти:

```
hFont =(HFONT) SelectObject(hdcMem, CreateFontIndirect(&lf));
```

Теперь, когда вы записываете в контекст памяти какой-то текст, Windows будет использовать для этого шрифт TrueType Arial, выбранный в контекст устройства.

Но поверхность экрана, представляемая этим контекстом памяти, все еще монохромная и содержит один пиксель. Необходимо сделать битовый образ достаточно большим для вывода в него текста. Размеры текста можно получить с помощью функции *GetTextExtentPoint*, а с помощью функции *CreateBitmap* создать битовый образ на основе полученных размеров:

```
GetTextExtentPoint(hdcMem, "Arial", 5, &size);
hBitmap = CreateBitmap(size.cx, size.cy, 1, 1, NULL);
SelectObject(hdcMem, hBitmap);
```

Теперь контекст устройства имеет монохромную поверхность экрана размер которого точно соответствует размеру текста. Все, что теперь нужно сделать, это вписать туда текст. Вы уже видели эту функцию раньше:

```
TextOut(hdcMem, 0, 0, "Arial", 5);
```

Осталось только очистить память. Для этого с помощью функции *SelectObject* снова выбираем в контекст устройства системный шрифт (он имеет сохраненный нами описатель *hFont*) и удаляем описатель предыдущего шрифта, который возвращает функция *SelectObject*, т. е. описатель шрифта Arial:

```
DeleteObject(SelectObject(hdcMem, hFont));
```

Теперь можно удалить оба контекста устройства:

```
DeleteDC(hdcMem);
DeleteDC(hdc);
```

У нас остался битовый образ, в котором имеется текст "Arial", написанный шрифтом Arial.

Масштабирование битовых образов

Контекст памяти также выручает при необходимости масштабирования шрифтов для дисплеев с другой разрешающей способностью или коэффициентом сжатия. В программе GRAFMENU создаются четыре битовых образа, для коррекции размеров при выводе на экран, который имеет системный шрифт высотой 8 пикселей и шириной 4 пикселя. Для других размеров системного шрифта битовые образы должны быть растянуты. В программе GRAFMENU это делается с помощью функции *StretchBitmap*.

Первым шагом должно стать получение контекста устройства для экрана, получение размеров системного шрифта и создание двух контекстов памяти:

```
hdc = CreateIC("DISPLAY", NULL, NULL, NULL);
GetTextMetrics(hdc, &tm);
hdcMem1 = CreateCompatibleDC(hdc);
hdcMem2 = CreateCompatibleDC(hdc);
DeleteDC(hdc);
```

Описателем битового образа, который передается функции, является *hBitmap1*. С помощью функции *GetObject* программа может получить размеры этого битового образа:

```
GetObject(hBitmap1, sizeof(BITMAP),(PSTR) &bm1);
```

Таким образом размеры копируются в структуру *bm1* типа BITMAP. Структура *bm2* устанавливается равной *bm1*, а затем на основе размеров системного шрифта некоторые размеры в этой структуре модифицируются:

```
bm2 = bm1;
bm2.bmWidth =(tm.tmAveCharWidth * bm2.bmWidth) / 4;
bm2.bmHeight =(tm.tmHeigth * bm2.bmHeight) / 8;
bm2.bmWidthBytes =((bm2.bmWidth + 15) / 16) * 2;
```

Теперь, на основе измененных размеров, можно создать новый битовый образ с описателем *hBitmap2*:

```
hBitmap2 = CreateBitmapIndirect(&bm2);
```

Затем, в двух контекстах памяти можно выбрать эти два битовых образа:

```
SelectObject(hdcMem1 hBitmap1);
SelectObject(hdcMem2, hBitmap2);
```

Необходимо скопировать первый битовый образ во второй и увеличить его. Это требует вызова функции *StretchBlt*:

```
StretchBlt(hdcMem2, 0, 0, bm2.bmWidth, bm2.bmHeight,
            hdcMem1, 0, 0, bm1.bmWidth, bm1.bmHeight, SRCCOPY);
```

Теперь второй битовый образ правильно промасштабирован. Он будет использоваться в меню. Очистка делается просто:

```
DeleteDC(hdcMem1);
DeleteDC(hdcMem2);
DeleteDC(hBitmap1);
```

Соберем все вместе

В функции *WinMain* программы GRAFMENU при создании меню используются функции *StretchBitmap* и *GetBitmapFont*. В программе GRAFMENU имеется два меню, определенных ранее в файле описания ресурсов. Они станут всплывающими меню для опций File и Edit.

Программа GRAFMENU начинается с получения описателя пустого меню:

```
hMenu = CreateMenu();
```

Всплывающее меню File (содержащее четыре опции New, Open, Save и Save As) загружается из файла описания ресурсов:

```
hMenuPopup = LoadMenu(hInstance, "MenuFile");
```

Битовый образ, в котором содержится слово "FILE", также загружается из файла описания ресурсов, а затем с помощью функции *StretchBitmap*, увеличивается его размер:

```
hBitmapFile = StretchBitmap(LoadBitmap(hInstance, "BitmapFile"));
```

Описатели битового образа и всплывающего меню становятся параметрами функции *AppendMenu*:

```
AppendMenu(hMenu, MF_BITMAP | MF_POPUP,(int) hMenuPopup,(PSTR)(LONG) hBitmapFile);
```

Аналогичные действия выполняются для меню Edit:

```
hMenuPopup = LoadMenu(hInstance, "MenuEdit");
hBitmapEdit = StretchBitmap(LoadBitmap(hInstance, "BitmapEdit"));
AppendMenu(hMenu, MF_BITMAP | MF_POPUP,(int) hMenuPopup,
          (PSTR)(LONG) hBitmapEdit );
```

Всплывающее меню трех шрифтов создается при последовательных вызовах функции *GetBitmapFont*:

```
hMenuPopup = CreateMenu();
for(i = 0; i < 3; i++)
{
       hBitmapPopFont[i] = GetBitmapFont(i);
       AppendMenu(hMenuPopup, MF_BITMAP, IDM_COUR + i,
       (PSTR)(LONG) hMenuPopupFont[i]);
}
```

Затем всплывающее меню добавляется к основному меню:

```
hBitmapFont = StretchBitmap(LoadBitmap(hInstance, "BitmapFont"));
AppendMenu(hMenu, MF_BITMAP | MF_POPUP,(int) hMenuPopup,
          (PSTR)(LONG) hBitmapFont);
```

Меню окна готово. Теперь можно включить описатель меню *hMenu* в вызов функции *CreateWindow*:

```
hwnd = CreateWindow(szAppName, "Bitmap Menu Demonstration",
                    WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    NULL, hMenu, hInstance, NULL);
```

После получения *hwnd*, программа GRAFMENU может изменить системное меню. Сначала программа получает его описатель:

```
hMenu = GetSystemMenu(hwnd, FALSE);
```

Следующая инструкция загружает битовый образ "Help" и увеличивает его до соответствующего размера:

```
hBitmapHelp = StretchBitmap(LoadBitmap(hInstance, "BitmapHelp"));
```

Теперь к системному меню добавляется разделитель и увеличенный битовый образ:

```
AppendMenu(hMenu, MF_SEPARATOR, NULL, NULL);
AppendMenu(hMenu, MF_BITMAP, IDM_HELP,(PSTR)(LONG) hBitmapHelp);
```

Запомните, что битовые образы являются объектами GDI и должны быть явно удалены при завершении программы. В программе GRAFMENU эта задача решается после выхода из цикла обработки сообщений:

```
DeleteObject(hBitmapHelp);
DeleteObject(hBitmapEdit);
DeleteObject(hBitmapFile);
DeleteObject(hBitmapFont);

for(i = 0; i < 3; i++)
       DeleteObject(hBitmapPopFont[i]);
```

В заключении раздела приведем несколько общих замечаний:

- Высоту панели главного меню Windows изменяет таким образом, чтобы туда уместился самый высокий битовый образ. Другие битовые образы (или символьные строки) выравниваются по верхней границе панели меню. Размер панели меню, полученный от функции:

  ```
  GetSystemMetrics(SM_CYMENU)
  ```

  изменяется, поскольку в меню добавлены битовые образы.

- При работе с программой GRAFMENU, можно использовать метки пунктов всплывающих меню, заданных в виде битовых образов, но метки при этом имеют обычный размер. Если это вам не нравится, то можно создать свои метки и использовать функцию *SetMenuItemBitmaps*.

- Другим подходом к использованию в меню не текста (или текста, набранного не системным шрифтом) является меню, отображаемое владельцем (owner-draw).

Добавление интерфейса клавиатуры

Теперь появилась новая проблема. Если в меню содержится текст, Windows автоматически добавляет интерфейс клавиатуры. Пункт меню можно выбрать с помощью клавиши <Alt> в комбинации с буквой символьной строки. Но после того как в меню помещен битовый образ, интерфейс клавиатуры отключается.

Это как раз тот самый случай, когда в дело вступает сообщение WM_MENUCHAR. Windows посылает это сообщение оконной процедуре, когда нажимается клавиша <Alt> в комбинации с такой буквой символьной строки, которая не соответствует ни одному пункту меню. Необходимо отслеживать сообщения WM_MENUCHAR и проверять младшее слово параметра *wParam* (ASCII-код символа нажатой клавиши). Если он соответствует пункту меню, то необходимо возвратить обратно в Windows длинное целое, где старшее слово устанавливается в 2, а младшее слово устанавливается равным индексу того пункта меню, который мы хотим связать с этой клавишей. Остальное выполняет Windows.

Быстрые клавиши

Если выразиться максимально кратко, то быстрые клавиши (keyboard accelerators) — это комбинации клавиш, которые генерируют сообщения WM_COMMAND (в некоторых случаях WM_SYSCOMMAND). Чаще всего быстрые клавиши используются в программах для дублирования действий обычных опций меню. (Однако быстрые клавиши могут выполнять и такие функции, которых нет в меню.) Например, в некоторых программах для Windows имеется меню Edit, которое включает в себя опцию Delete; в этих программах для этой опции быстрой клавишей обычно является клавиша <Del>. Пользователь может выбрать из меню опцию Delete, нажимая <Alt>-комбинацию, или может просто нажать быструю клавишу <Del>. Когда оконная процедура получает сообщение WM_COMMAND, то ей не нужно определять, что именно, меню или быстрая клавиша, использовались.

Зачем нужны быстрые клавиши?

Вы можете спросить: "Зачем нужны быстрые клавиши?" Почему нельзя просто отслеживать сообщения WM_KEYDOWN или WM_CHAR и самому дублировать функции меню? В чем преимущество быстрых клавиш? Для простого приложения, имеющего одно окно, несомненно, можно отслеживать сообщения клавиатуры, но, используя быстрые клавиши, вы получаете определенные преимущества: вам не нужно дублировать логику меню и быстрых клавиш.

Для многооконных приложений с множеством оконных процедур быстрые клавиши очень важны. Как известно, Windows посылает сообщения клавиатуры оконной процедуре того окна, которое в данный момент имеет фокус ввода. Однако, в случае быстрых клавиш Windows посылает сообщение WM_COMMAND той оконной процедуре, чей описатель задан в функции *TranslateAccelerator*. Как правило, это будет оконная процедура главного окна вашей программы, т. е. того окна, в котором имеется меню. Следовательно, нет необходимости дублировать логику действия быстрых клавиш в каждой оконной процедуре.

Это преимущество становится особенно важным при использовании немодальных окон диалога (которые будут рассмотрены в следующей главе) или дочерних окон, расположенных в рабочей области вашего главного окна. Если при наличии нескольких окон для перемещения между окнами назначается определенная быстрая клавиша, то только одна оконная процедура должна включать в себя эту логику. Дочерние окна не получают от быстрых клавиш сообщений WM_COMMAND.

Некоторые правила назначения быстрых клавиш

Теоретически можно определить быструю клавиши почти для каждой виртуальной или символьной клавиши в сочетании с клавишами <Shift>, <Ctrl> или <Alt>. Однако надо попытаться добиться какого-то соответствия с другими приложениями и избегать применения тех клавиш, которые использует Windows. Нежелательно назначать быстрыми клавишами клавиши <Tab>, <Enter>, <Esc>, <Spacebar>, поскольку они часто используются для системных функций.

Наиболее часто с быстрыми клавишами работают в меню Edit. Рекомендуемые быстрые клавиши для этих пунктов меню различаются у версий Windows 3.0 и Windows 3.1, поэтому следует обеспечить поддержку как старых, так и новых быстрых клавиш, показанных в следующей таблице:

Функция	Быстрые клавиши старые	Быстрые клавиши новые
Undo (отменить)	<Alt>+<Backspace>	<Ctrl>+<Z>
Cut (вырезать)	<Shift>+<Del>	<Ctrl>+<X>
Copy (копировать)	<Ctrl>+<Ins>	<Ctrl>+<C>
Paste (вставить)	<Shift>+<Ins>	<Ctrl>+<V>
Delete или Clear (удалить или очистить)	<Del>	<Del>

Другой известной быстрой клавишей является клавиша <F1> для вызова подсказки. Избегайте применения клавиш <F4>, <F5> и <F6>, поскольку они часто используются для специальных функций многооконного интерфейса приложений (Multiple Document Interface, MDI), о котором рассказывается в главе 18.

Таблица быстрых клавиш

Быстрые клавиши определяются в файле описания ресурсов (файл с расширением .RC). Здесь показана общая форма определения:

```
MyAccelerators ACCELERATORS
{
       [определения быстрых клавиш]
}
```

Эта таблица быстрых клавиш называется *MyAccelerators*. В таблицу ACCELERATORS не включаются опции загрузки и памяти. В файле описания ресурсов можно иметь несколько таблиц ACCELERATORS.

Для каждой определяемой быстрой клавиши необходима отдельная строка таблицы. Имеется четыре типа определений быстрых клавиш:

```
"char",       id                   [,<SHIFT>] [,CONTROL] [,<ALT>]
"^char",      id                   [,<SHIFT>] [,CONTROL] [,<ALT>]
nCode,        id, ASCII            [,<SHIFT>] [,CONTROL] [,<ALT>]
nCode,        id, VIRTKEY          [,<SHIFT>] [,CONTROL] [,<ALT>]
```

В этих примерах "*char*" означает один символ, заключенный в кавычки, а "^*char*" — это символ ^ и один символ, заключенный в кавычки. Число *id* выполняет ту же функцию, что и идентификатор меню в определении меню. Это значение, которое Windows посылает вашей оконной процедуре в сообщении WM_COMMAND для идентификации быстрой клавиши. Обычно эти идентификаторы определяются в заголовочном файле. Быстрые клавиши почти всегда служат для выбора опций из всплывающих меню. Если быстрая клавиша дублирует команду меню, используйте один и тот же идентификатор для меню и для быстрой клавиши. Если быстрая клавиша не дублирует команду меню, используйте уникальный идентификатор.

В первом типе определения идентификатора быстрая клавиша — это чувствительный к регистру символ в кавычках:

```
"char",       id                   [,<SHIFT>] [,CONTROL] [,<ALT>]
```

Если вы хотите использовать эту клавишу в сочетании с одной или более клавишами <Shift>, <Ctrl> и <Alt>, просто добавьте SHIFT, CONTROL и/или ALT.

В определении второго типа быстрая клавиша — это символ в сочетании с клавишей <Ctrl>:

```
"^char",      id                   [,<SHIFT>] [,CONTROL] [,<ALT>]
```

Этот тип определения эквивалентен первому, если бы в нем за символом было бы указано ключевое слово CONTROL.

В определениях третьего и четвертого типов вместо символа в кавычках используется число (*nCode*):

```
nCode,        id, ASCII            [,<SHIFT>] [,CONTROL] [,<ALT>]
nCode,        id, VIRTKEY          [,<SHIFT>] [,CONTROL] [,<ALT>]
```

Это число интерпретируется либо как чувствительный к регистру ASCII-код символа, либо как код виртуальной клавиши, в зависимости от наличия ключевого слова ASCII или VIRTKEY.

Наиболее часто используются определения быстрых клавиш второго и четвертого типа. В определении второго типа буквенная клавиша используется в сочетании с клавишей <Ctrl>. Например, здесь определяется быстрая клавиша <Ctrl>+<A>:

```
"^A", id
```

Определение четвертого типа используется для виртуальных кодов клавиш, таких как функциональные клавиши. Для комбинации <Ctrl>+<F9> используется следующее определение:

```
VK_F9, wid, VIRTKEY, CONTROL
```

Идентификатор VK_F9 определяется в заголовочных файлах Windows в качестве виртуального кода клавиши <F9>, поэтому, вы должны включить в программу в начале файла описания ресурсов инструкцию:

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

Приведенные выше определения первого и третьего типов применяются редко. Если вы хотите их использовать, внимательно контролируйте чувствительность к регистру. Windows осуществляет чувствительный к регистру контроль совпадения "*char*" или *nCode*, основываясь на нажатой клавише. Если вы добавляете ключевое слово SHIFT, Windows контролирует, нажата или нет клавиша <Shift>. Эта ситуация иногда приводит к неожиданному результату. Например, если "*char*" — это "*A*", то быстрая клавиша срабатывает, если нажата клавиша <A> или при нажатой клавише <Shift> или при включенном режиме Caps Lock. Если вы используете "*A*" с ключевым словом SHIFT, то быстрая клавиша активизируется, если нажата клавиша <A> при нажатой клавише <Shift>, но не должна вызываться при включенном режиме Caps Lock. Точно также "*a*" — сама по себе является быстрой клавишей для клавиши <A> нижнего регистра или для клавиши <A> с нажатой клавишей <Shift> и включенном режиме Caps Lock. Но "*a*" с ключевым словом SHIFT является быстрой клавишей только в одном случае, если клавиша <Shift> нажата, а Caps Lock включен.

При определении быстрых клавиш для пунктов меню необходимо включить описание комбинации быстрых клавиш в текст соответствующего пункта меню. Символ табуляции (\t) разделяет текст и описание быстрой клавиши так, чтобы описание быстрых клавиш располагалось во втором столбце. При наличии слов Shift, Ctrl или Alt после них пишется знак плюс и собственно клавиша. Например:

- F6
- Shift+F6
- Ctrl+F6

Загрузка таблицы быстрых клавиш

Внутри программы для загрузки таблицы быстрых клавиш в память и получения ее описателя используется функция *LoadAccelerators*. Инструкция с функцией *LoadAccelerators* очень похожа на аналогичные инструкции с *LoadIcon*, *LoadCursor*, *LoadBitmap* и *LoadMenu*.

Сначала таблица быстрых клавиш определяется как имеющая тип HACCEL:

```
HACCEL hAccel;
```

Затем производится загрузка таблицы:

```
hAccel = LoadAccelerators(hInstance, "MyAccelerators");
```

Как и в случаях со значками, курсорами, битовыми образами и меню, вместо имени таблицы быстрых клавиш можно вставить число, которое затем будет использоваться в инструкции, содержащей функцию *LoadAccelerators* и макрос MAKEINTRESOURCE. Вместо макроса перед числом можно ставить символ #, тогда весь параметр заключается в кавычки.

Преобразование нажатий клавиш клавиатуры

Мы не будем менять те три строки программы, которые являются общими почти для всех программ Windows, уже приведенных в этой книге. Данная инструкция — стандартный цикл обработки сообщений:

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

Далее приводятся те изменения, которые необходимы, чтобы использовать таблицу быстрых клавиш:

```
while(GetMessage(&msg, NULL, 0, 0))
{
       if(!TranslateAccelerator(hwnd, hAccel, &msg))
       {
              TranslateMessage(&msg);
              DispatchMessage(&msg);
       }
}
```

Функция *TranslateAccelerator* определяет, является ли сообщение, хранящееся в структуре *msg*, сообщением клавиатуры. Если да, функция ищет соответствия в таблице быстрых клавиш, описателем которой является *hAccel*. Если соответствие находится, она вызывает оконную процедуру окна, описателем которого является *hwnd*. Если быстрая клавиша соответствует пункту системного меню, то отправляемым сообщением является WM_SYSCOMMAND. В противном случае — WM_COMMAND.

Возвращаемое значение функции *TranslateAccelerator* не равно нулю, если сообщение уже было преобразовано (и уже отправлено в оконную процедуру), и равно 0, если нет. Если возвращаемое значение функции *TranslateAccelerator* не равно нулю, то нет необходимости вызывать функции *TranslateMessage* и *DispatchMessage*, и поэтому управление вновь передается функции *GetMessage*.

Параметр *hwnd* функции *TranslateAccelerator* кажется несколько излишним, поскольку он не требуется в трех остальных функциях цикла обработки сообщений. Более того, в самой структуре сообщения (переменная структуры *msg*) имеется член с именем *hwnd*, который является описателем окна. Отличия обусловлены следующим:

Поля структуры *msg* заполняются при вызове функции *GetMessage*. Если второй параметр функции *GetMessage* равен NULL, то функция извлекает сообщения всех окон, принадлежащих приложению. При возвращении из функции *GetMessage*, число *hwnd* структуры *msg* является описателем того окна, которое получит сообщение. Однако, когда функция *TranslateAccelerator* преобразует сообщение клавиатуры в сообщение WM_COMMAND или WM_SYSCOMMAND, она заменяет описатель окна *msg.hwnd* описателем *hwnd*, заданным в качестве первого параметра функции. Таким образом, Windows посылает все сообщения быстрых клавиш одной оконной процедуре, даже если в данный момент фокус ввода имеет другое окно приложения. Функция *TranslateAccelerator* не преобразует сообщения клавиатуры, когда модальное окно диалога или окно сообщений имеет фокус ввода, поскольку сообщения для этих окон не проходят через цикл обработки сообщений программы.

В некоторых случаях, когда какое-то другое окно вашей программы (например, немодальное окно диалога) имеет фокус ввода, нажатия быстрых клавиш можно не преобразовывать. В следующей главе вы узнаете, как работать в такой ситуации.

Получение сообщений быстрых клавиш

Если быстрая клавиша связана с пунктом системного меню, функция *TranslateAccelerator* посылает оконной процедуре сообщение WM_SYSCOMMAND. В противном случае функция *TranslateAccelerator* посылает оконной процедуре сообщение WM_COMMAND. В следующей таблице показаны типы сообщений WM_COMMAND, которые можно получить для быстрых клавиш, команд меню и дочерних окон управления:

	Младшее слово (LOWORD) (wParam)	**Старшее слово (HIWORD) (wParam)**	**lParam**
Быстрая клавиша:	Идентификатор быстрой клавиши	1	0
Меню:	Идентификатор меню	0	0
Элемент управления:	Идентификатор элемента управления	Код уведомления	Описатель дочернего окна

Кроме этого, если быстрая клавиша соответствует пункту меню, оконная процедура получает сообщения WM_INITMENU, WM_INITMENUPOPUP и WM_MENUSELECT точно также, как при выборе опции меню. Обычно, в программах при обработке сообщений WM_INITMENUPOPUP делают разрешенными и запрещенными пункты всплывающих меню. При работе с быстрыми клавишами эта возможность по-прежнему имеется. Однако

если быстрая клавиша соответствует запрещенному или недоступному пункту меню, то функция *TranslateAccelerator* не посылает оконной процедуре сообщение WM_COMMAND или WM_SYSCOMMAND.

При сворачивании активного окна для быстрых клавиш, соответствующих разрешенным пунктам системного меню, функция *TranslateAccelerator* посылает оконной процедуре сообщения WM_SYSCOMMAND, а не сообщения WM_COMMAND. Кроме того, функция *TranslateAccelerator* посылает оконной процедуре сообщения WM_COMMAND для быстрых клавиш, которые не соответствуют ни одному из пунктов меню.

Программа POPPAD, имеющая меню и быстрые клавиши

В главе 8 мы создали программу POPPAD1, в которой для реализации простейших функций редактирования текста использовалось дочернее окно редактирования. Теперь мы добавим в программу меню File и Edit, а программу назовем POPPAD2. Все пункты меню Edit будут функционировать; создание функций меню File мы завершим в главе 11, а функцию *Print* — в главе 15. Программа POPPAD2 представлена на рис. 10.10.

POPPAD2.MAK

```
#-----------------------
# POPPAD2.MAK make file
#-----------------------

poppad2.exe : poppad2.obj poppad2.res
     $(LINKER) $(GUIFLAGS) -OUT:poppad2.exe poppad2.obj poppad2.res $(GUILIBS)

poppad2.obj : poppad2.c poppad2.h
     $(CC) $(CFLAGS) poppad2.c

poppad2.res : poppad2.rc poppad2.h poppad2.ico
     $(RC) $(RCVARS) poppad2.rc
```

POPPAD2.C

```
/*-----------------------------------------------------
   POPPAD2.C -- Popup Editor Version 2(includes menu)
               (c) Charles Petzold, 1996
  -----------------------------------------------------*/

#include <windows.h>
#include "poppad2.h"

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

char szAppName[] = "PopPad2";

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     HACCEL     hAccel;
     HWND       hwnd;
     MSG        msg;
     WNDCLASSEX wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(hInstance, szAppName);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = szAppName;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);
```

```
     hwnd = CreateWindow(szAppName, szAppName,
                         WS_OVERLAPPEDWINDOW,
                         GetSystemMetrics(SM_CXSCREEN) / 4,
                         GetSystemMetrics(SM_CYSCREEN) / 4,
                         GetSystemMetrics(SM_CXSCREEN) / 2,
                         GetSystemMetrics(SM_CYSCREEN) / 2,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

     hAccel = LoadAccelerators(hInstance, szAppName);

     while(GetMessage(&msg, NULL, 0, 0))
          {
          if(!TranslateAccelerator(hwnd, hAccel, &msg))
               {
               TranslateMessage(&msg);
               DispatchMessage(&msg);
               }
          }
     return msg.wParam;
     }

AskConfirmation(HWND hwnd)
     {
     return MessageBox(hwnd, "Really want to close PopPad2?",
                       szAppName, MB_YESNO | MB_ICONQUESTION);
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HWND hwndEdit;
     int         iSelect, iEnable;

     switch(iMsg)
          {
          case WM_CREATE :
               hwndEdit = CreateWindow("edit", NULL,
                          WS_CHILD | WS_VISIBLE | WS_HSCROLL | WS_VSCROLL |
                               WS_BORDER | ES_LEFT | ES_MULTILINE |
                               ES_AUTOHSCROLL | ES_AUTOVSCROLL,
                          0, 0, 0, 0,
                          hwnd,(HMENU) 1,
                         ((LPCREATESTRUCT) lParam)->hInstance, NULL);
               return 0;

          case WM_SETFOCUS :
               SetFocus(hwndEdit);
               return 0;

          case WM_SIZE :
               MoveWindow(hwndEdit, 0, 0, LOWORD(lParam),
                                         HIWORD(lParam), TRUE);
               return 0;

          case WM_INITMENUPOPUP :
               if(lParam == 1)
                    {
                    EnableMenuItem((HMENU) wParam, IDM_UNDO,
                         SendMessage(hwndEdit, EM_CANUNDO, 0, 0) ?
                              MF_ENABLED : MF_GRAYED);
```

```
               EnableMenuItem((HMENU) wParam, IDM_PASTE,
                    IsClipboardFormatAvailable(CF_TEXT) ?
                         MF_ENABLED : MF_GRAYED);

               iSelect = SendMessage(hwndEdit, EM_GETSEL, 0, 0);

               if(HIWORD(iSelect) == LOWORD(iSelect))
                    iEnable = MF_GRAYED;
               else
                    iEnable = MF_ENABLED;

               EnableMenuItem((HMENU) wParam, IDM_CUT,   iEnable);
               EnableMenuItem((HMENU) wParam, IDM_COPY,  iEnable);
               EnableMenuItem((HMENU) wParam, IDM_DEL,   iEnable);

               return 0;
               }
          break;

     case WM_COMMAND :
          if(lParam)
               {
               if(LOWORD(lParam) == 1 &&
                   (HIWORD(wParam) == EN_ERRSPACE ||
                     HIWORD(wParam) == EN_MAXTEXT))
                         MessageBox(hwnd, "Edit control out of space.",
                                     szAppName, MB_OK | MB_ICONSTOP);
               return 0;
               }

          else switch(LOWORD(wParam))
                    {
                    case IDM_NEW :
                    case IDM_OPEN :
                    case IDM_SAVE :
                    case IDM_SAVEAS :
                    case IDM_PRINT :
                         MessageBeep(0);
                         return 0;

                    case IDM_EXIT :
                         SendMessage(hwnd, WM_CLOSE, 0, 0);
                         return 0;

                    case IDM_UNDO :
                         SendMessage(hwndEdit, WM_UNDO, 0, 0);
                         return 0;

                    case IDM_CUT :
                         SendMessage(hwndEdit, WM_CUT, 0, 0);
                         return 0;

                    case IDM_COPY :
                         SendMessage(hwndEdit, WM_COPY, 0, 0);
                         return 0;

                    case IDM_PASTE :
                         SendMessage(hwndEdit, WM_PASTE, 0, 0);
                         return 0;

                    case IDM_DEL :
                         SendMessage(hwndEdit, WM_CLEAR, 0, 0);
                         return 0;
```

```
                        case IDM_SELALL :
                             SendMessage(hwndEdit, EM_SETSEL, 0, -1);
                             return 0;

                        case IDM_HELP :
                             MessageBox(hwnd, "Help not yet implemented!",
                                  szAppName, MB_OK | MB_ICONEXCLAMATION);
                             return 0;

                        case IDM_ABOUT :
                             MessageBox(hwnd,
                                  "POPPAD2(c) Charles Petzold, 1996",
                                  szAppName, MB_OK | MB_ICONINFORMATION);
                             return 0;
                        }
               break;

          case WM_CLOSE :
               if(IDYES == AskConfirmation(hwnd))
                    DestroyWindow(hwnd);
               return 0;

          case WM_QUERYENDSESSION :
               if(IDYES == AskConfirmation(hwnd))
                    return 1;
               else
                    return 0;

          case WM_DESTROY :
               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

POPPAD2.RC

```
/*----------------------------
   POPPAD2.RC resource script
  ----------------------------*/

#include <windows.h>
#include "poppad2.h"

PopPad2 ICON poppad2.ico

PopPad2 MENU
     {
     POPUP "&File"
          {
          MENUITEM "&New",              IDM_NEW
          MENUITEM "&Open...",          IDM_OPEN
          MENUITEM "&Save",             IDM_SAVE
          MENUITEM "Save &As...",       IDM_SAVEAS
          MENUITEM SEPARATOR
          MENUITEM "&Print",            IDM_PRINT
          MENUITEM SEPARATOR
          MENUITEM "E&xit",             IDM_EXIT
          }
     POPUP "&Edit"
          {
          MENUITEM "&Undo\tCtrl+Z",     IDM_UNDO
          MENUITEM SEPARATOR
          MENUITEM "Cu&t\tCtrl+X",      IDM_CUT
          MENUITEM "&Copy\tCtrl+C",     IDM_COPY
```

```
        MENUITEM "&Paste\tCtrl+V",    IDM_PASTE
        MENUITEM "De&lete\tDel",      IDM_DEL
        MENUITEM SEPARATOR
        MENUITEM "&Select All",       IDM_SELALL
        }
    POPUP "&Help"
        {
        MENUITEM "&Help...",          IDM_HELP
        MENUITEM "&About PopPad2...", IDM_ABOUT
        }
    }

PopPad2 ACCELERATORS
    {
    "^Z",      IDM_UNDO
    VK_BACK,   IDM_UNDO,  VIRTKEY, ALT
    "^X",      IDM_CUT
    VK_DELETE, IDM_CUT,   VIRTKEY, SHIFT
    "^C",      IDM_COPY
    VK_INSERT, IDM_COPY,  VIRTKEY, CONTROL
    "^V",      IDM_PASTE
    VK_INSERT, IDM_PASTE, VIRTKEY, SHIFT
    VK_DELETE, IDM_DEL,   VIRTKEY
    VK_F1,     IDM_HELP,  VIRTKEY
    }
```

POPPAD2.H

```
/*-----------------------
   POPPAD2.H header file
  -----------------------*/

#define IDM_NEW     1
#define IDM_OPEN    2
#define IDM_SAVE    3
#define IDM_SAVEAS  4
#define IDM_PRINT   5
#define IDM_EXIT    6

#define IDM_UNDO    10
#define IDM_CUT     11
#define IDM_COPY    12
#define IDM_PASTE   13
#define IDM_DEL     14
#define IDM_SELALL  15

#define IDM_HELP    20
#define IDM_ABOUT   22
```

POPPAD2.ICO

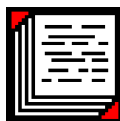


Рис. 10.10 Программа POPPAD2

В файле описания ресурсов POPPAD2.RC содержится меню и таблица быстрых клавиш. Обратите внимание, что все быстрые клавиши показаны после символа табуляции (\t) внутри символьных строк всплывающего меню Edit.

Разрешение пунктов меню

Основная работа оконной процедуры состоит теперь в разрешении и запрещении пунктов меню Edit, что осуществляется при обработке сообщения WM_INITMENUPOPUP. Сначала программа проверяет, отображается

ли всплывающее меню Edit. Поскольку индекс положения пункта Edit в меню (начинающегося с пункта File, чей индекс положения равен 0) равен 1, то *lParam* равен 1 в том случае, если отображается всплывающее меню Edit.

Для того чтобы определить, может ли быть разрешена опция Undo, программа POPPAD2 посылает сообщение EM_CANUNDO дочернему окну редактирования. Возвращаемое значение функции *SendMessage* не равно нулю, если дочернее окно редактирования может выполнить операцию Undo, в этом случае опция Undo делается разрешенной; в противном случае опция делается недоступной:

```
EnableMenuItem((HMENU) wParam, IDM_UNDO,
       SendMessage(hwndEdit, EM_CANUNDO, 0, 0) ? MF_ENABLED : MF_GRAYED);
```

Опцию Paste следует делать разрешенной только в том случае, если в данный момент в папке обмена имеется текст. Определить это можно с помощью функции *IsClipboardFormatAvailable* с идентификатором CF_TEXT:

```
EnableMenuItem((HMENU) wParam, IDM_PASTE,
       IsClipboardFormatAvailable(CF_TEXT) ? MF_ENABLED : MF_GRAYED);
```

Опции Cut, Copy и Delete следует делать разрешенными только в том случае, если в окне редактирования был выделен текст. При посылке окну управления сообщения EM_GETSEL возвращаемым значением функции *SendMessage* является целое, в котором содержится эта информация:

```
iSelect = SendMessage(hwndEdit, EM_GETSEL, 0, 0);
```

Младшим словом *iSelect* является положение первого выбранного символа; старшим словом *iSelect* является положение первого символа после выделения. Если два этих слова равны, то в окне редактирования нет выбранного текста:

```
if(HIWORD(iSelect) == LOWORD(iSelect))
       iEnable = MF_GRAYED;
else
       iEnable = MF_ENABLED;
```

Затем значение *iEnable* используется для опций Cut, Copy и Delete:

```
EnableMenuItem((HMENU) wParam, IDM_CUT, iEnable);
EnableMenuItem((HMENU) wParam, IDM_COPY, iEnable);
EnableMenuItem((HMENU) wParam, IDM_DEL, iEnable);
```

Обработка опций меню

Конечно, если бы мы в программе POPPAD2 не использовали элемент управления — дочернее окно редактирования, то теперь нам пришлось бы заняться проблемами, связанными с фактической реализацией опций Undo, Cut, Copy, Paste, Delete и Select All из меню Edit. Но окно редактирования делает этот процесс элементарным: мы просто посылаем окну редактирования сообщение для каждой из этих опций:

```
case IDM_UNDO :
       SendMessage(hwndEdit, WM_UNDO, 0, 0);
       return 0;

case IDM_CUT :
       SendMessage(hwndEdit, WM_CUT, 0, 0);
       return 0;

case IDM_COPY :
       SendMessage(hwndEdit, WM_COPY, 0, 0);
       return 0;

case IDM_PASTE :
       SendMessage(hwndEdit, WM_PASTE, 0, 0);
       return 0;

case IDM_DEL :
       SendMessage(hwndEdit, WM_CLEAR, 0, 0);
       return 0;

case IDM_SELALL :
       SendMessage(hwndEdit, WM_SETSEL, 0, -1);
       return 0;
```

Обратите внимание, что мы могли бы еще больше все упростить, приравняв значения IDM_UNDO, IDM_CUT и другие — значениям соответствующих сообщений окна WM_UNDO, WM_CUT и т. д.

Опция About всплывающего меню File вызывает простое окно сообщения:

```
case IDM_ABOUT :
       MessageBox(hwnd, "POPPAD2(c) Charles Petzold, 1996",
                     szAppName, MB_OK | MB_ICONINFORMATION);
       return 0;
```

В главе 11 мы создадим здесь окно диалога. Окно сообщения появляется также при выборе опции Help этого меню или при нажатии быстрой клавиши <F1>.

Опция Exit посылает оконной процедуре сообщение WM_CLOSE:

```
case IDM_EXIT :
       SendMessage(hwndEdit, WM_CLOSE, 0, 0);
       return 0;
```

Это именно то, что делает функция *DefWindowProc* при получении сообщения WM_SYSCOMMAND с параметром *wParam*, равным SC_CLOSE.

В предыдущих программах мы не обрабатывали сообщение WM_CLOSE в оконной процедуре, а просто передавали его в *DefWindowProc*. *DefWindowProc* поступает с сообщениями WM_CLOSE очень просто: она вызывает функцию *DestroyWindow*. Однако в программе POPPAD2 сообщение WM_CLOSE обрабатывается, а не пересылается в *DefWindowProc*. В данном случае это не имеет большого значения, но обработка сообщения WM_CLOSE станет важна, когда, как будет показано в главе 11, программа POPPAD будет фактически редактировать файлы:

```
case WM_CLOSE :
       if(IDYES == AskConfirmation(hwnd)) DestroyWindow(hwnd);
       return 0;
```

В программе POPPAD2 функция *AskConfirmation* выводит на экран окно сообщения, запрашивая подтверждение на завершение программы:

```
AskConfirmation(HWND hwnd)
{
       return MessageBox(hwnd, "Do you really want to close Poppad2?",
                     szAppName, MB_YESNO | MB_ICONQUESTION);
}
```

Окно сообщения (как и функция *AskConfirmation*) возвращает IDYES, если нажата кнопка Yes. Только после этого программа POPPAD2 вызывает функцию *DestroyWindow*. В противном случае программа не завершается.

Если перед завершением программы требуется подтверждение, то необходимо также обрабатывать сообщение WM_QUERYENDSESSION. Windows посылает каждой оконной процедуре сообщение WM_QUERYENDSESSION, когда пользователь заканчивает сеанс работы с Windows. Если какая-либо оконная процедура возвращает 0 в ответ на это сообщение, то сеанс работы с Windows не завершается. Мы обрабатываем сообщение WM_QUERYENDSESSION следующим образом:

```
case WM_QUERYENDSESSION :
       if( IDYES == AskConfirmation(hwnd))
              return 1;
       else
              return 0;
```

Сообщения WM_CLOSE и WM_QUERYENDSESSION — это два сообщения, которые требуют обработки, если необходимо запрашивать пользователя о подтверждении завершения программы. По этой причине в программе POPPAD2 опция Exit отправляет оконной процедуре сообщение WM_CLOSE. Это сделано во избежание еще и третьего запроса на подтверждение завершения.

Если вы обрабатываете сообщение WM_QUERYENDSESSION, вас заинтересует и сообщение WM_ENDSESSION. Windows посылает это сообщение каждой оконной процедуре, которая ранее получила сообщение WM_QUERYENDSESSION. Параметр *wParam* этого сообщения равен 0, если сеанс работы не может завершиться из-за того, что другая программа возвратила 0 в ответ на сообщение WM_QUERYENDSESSION. Практически сообщение WM_ENDSESSION отвечает на вопрос: "Я сообщил Windows, что могу закончить свою работу, но завершена ли она на самом деле?"

Хотя в меню File программы POPPAD2 включены обычные опции New, Open, Save и Save As, в этой программе они не работают. Для обработки этих опций необходимо использовать окна диалога. Сейчас мы готовы перейти к их изучению.

Глава 11 Окна диалога

11

Наиболее часто окна диалога или диалоговые окна используются для получения от пользователя дополнительной информации сверх той, которую может обеспечить меню. Программист показывает, что выбор какого-то пункта меню вызывает появления окна диалога, с помощью многоточия (...) после текста этого пункта меню.

Окно диалога обычно имеет вид всплывающего окна с разнообразными дочерними окнами элементов управления внутри. Размер и расположение этих дочерних окон задается в шаблоне окна диалога (dialog box template) в файле описания ресурсов программы. Microsoft Windows 95 обеспечивает возможность создания всплывающих окон диалога и дочерних окон элементов управления в нем, и возможность обработки оконной процедурой сообщений окна диалога (включая все сообщения клавиатуры и мыши). Тот код внутри Windows, который дает возможность все это сделать, иногда называют менеджером окна диалога (dialog box manager).

Многие сообщения, которые обрабатываются оконной процедурой окна диалога внутри Windows, также передаются и в вашу собственную программу в функцию, называемую процедурой окна диалога (dialog box procedure) или просто процедурой диалога (dialog procedure). Эта функция похожа на обычную оконную процедуру, но она имеет некоторые важные особенности. Как правило, внутри процедуры диалога не реализуется слишком много функций. Исключение составляют лишь инициализация дочерних окон элементов управления при создании окна диалога, обработка сообщений от дочерних окон элементов управления и завершение работы с окном диалога.

Рассмотрение окон диалога обычно бывает одним из самых объемных, поскольку оно включает в себя и описание дочерних окон элементов управления. Однако, материал о дочерних окнах элементов управления был представлен в главе 8. При использовании дочерних окон элементов управления, менеджер окна диалога Windows берет на себя решение многих из тех задач, с которыми нам пришлось столкнуться в главе 8. Например, проблемы, имевшие место в программе COLOR1 при передаче фокуса ввода от одной полосы прокрутки к другой, в окнах диалога не возникают. Windows управляет всей логикой переключения фокуса ввода между дочерними окнами элементов управления в окне диалога.

Тем не менее добавление в программу окна диалога – это непростая задача. Она требует внесения изменений в несколько файлов: шаблон окна диалога помещается в файл описания ресурсов, процедура окна диалога – в файл с исходными кодами программы, а идентификаторы, которые используются в окне диалога, часто вносятся в заголовочный файл программы. Для того чтобы почувствовать взаимозависимость всех этих составляющих, начнем с простого окна диалога.

Модальные окна диалога

Окна диалога бывают модальными (modal) или немодальными (modeless). Чаще всего встречаются модальные окна диалога. Если программа выводит на экран модальное окно диалога, то пользователь программы не может переключаться между окном диалога и другими окнами программы. Пользователь должен сначала закончить работу с окном диалога, это обычно делается путем щелчка на кнопке, помеченной либо OK, либо Cancel. Но, несмотря на наличие на экране окна диалога, пользователь может переключаться на другие программы. Некоторые окна диалога (называемые системными модальными окнами) этого делать не позволяют. Системное модальное окно диалога вынуждает пользователя, перед тем как он получит возможность сделать что-либо другое в Windows, завершить работу с ним.

Создание окна диалога About

Даже в тех программах для Windows, в которых ввод не требуется, окно диалога будет появляться достаточно часто, оно вызывается опцией About меню большинства программ. В этом окне диалога на экран выводится имя и значок программы, сведения об авторских правах, кнопка с надписью OK и, может быть, еще какая-то

информация. Предлагаемая вашему вниманию программа не делает ничего, кроме отображения окна диалога About. Программа ABOUT1 представлена на рис. 11.1.

ABOUT1.MAK

```
#----------------------
# ABOUT1.MAK make file
#----------------------

about1.exe : about1.obj about1.res
     $(LINKER) $(GUIFLAGS) -OUT:about1.exe about1.obj about1.res $(GUILIBS)

about1.obj : about1.c about1.h
     $(CC) $(CFLAGS) about1.c

about1.res : about1.rc about1.h about1.ico
     $(RC) $(RCVARS) about1.rc
```

ABOUT1.C

```
/*------------------------------------------
   ABOUT1.C -- About Box Demo Program No. 1
              (c) Charles Petzold, 1996
  ------------------------------------------*/

#include <windows.h>
#include "about1.h"

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
     {
     static char  szAppName[] = "About1";
     MSG          msg;
     HWND         hwnd;
     WNDCLASSEX   wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(hInstance, szAppName);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = szAppName;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(hInstance, szAppName);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "About Box Demo Program",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

```
          DispatchMessage(&msg);
          }
     return msg.wParam;
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static WNDPROC   lpfnAboutDlgProc;
     static HINSTANCE hInstance;

     switch(iMsg)
          {
          case WM_CREATE :
               hInstance =((LPCREATESTRUCT) lParam)->hInstance;
               return 0;

          case WM_COMMAND :
               switch(LOWORD(wParam))
                    {
                    case IDM_ABOUT :
                         DialogBox(hInstance, "AboutBox", hwnd, AboutDlgProc);
                         return 0;
                    }
               break;

          case WM_DESTROY :
               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }

BOOL CALLBACK AboutDlgProc(HWND hDlg, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     switch(iMsg)
          {
          case WM_INITDIALOG :
               return TRUE;

          case WM_COMMAND :
               switch(LOWORD(wParam))
                    {
                    case IDOK :
                    case IDCANCEL :
               EndDialog(hDlg, 0);
               return TRUE;
                    }
               break;
          }
     return FALSE;
     }
```

ABOUT1.RC

```
/*---------------------------
   ABOUT1.RC resource script
  ---------------------------*/

#include <windows.h>
#include "about1.h"

About1 ICON about1.ico

About1 MENU
     {
```

```
     POPUP "&Help"
          {
          MENUITEM "&About About1...",       IDM_ABOUT
          }
     }

AboutBox DIALOG  20, 20, 160, 80
     STYLE WS_POPUP | WS_DLGFRAME
     {
     CTEXT "About1"                     -1,   0, 12, 160,  8
     ICON  "About1"                     -1,   8,  8,   0,  0
     CTEXT "About Box Demo Program"     -1,   0, 36, 160,  8
     CTEXT "(c) Charles Petzold, 1996"  -1,   0, 48, 160,  8
     DEFPUSHBUTTON "OK"               IDOK,  64, 60,  32, 14, WS_GROUP
     }
```

ABOUT1.H

```
/*----------------------
   ABOUT1.H header file
  ----------------------*/

#define IDM_ABOUT      1
```

ABOUT1.ICO

Рис. 11.1 Программа ABOUT1

Шаблон окна диалога

Первой задачей, которую нужно решить для добавления в программу окна диалога, является создание шаблона окна диалога. Этот шаблон может быть помещен прямо в файл описания ресурсов, или он может быть создан в отдельном файле, для которого по договоренности используется расширение .DLG (dialog). При создании для шаблона отдельного файла, в файл описания ресурсов включается строка:

```
rcinclude filename.dlg
```

Шаблон окна диалога можно создавать вручную с помощью текстового редактора или можно использовать какой-нибудь инструмент для автоматизации этого процесса. Поскольку результат работы таких инструментов не может быть приведен здесь, то те шаблоны окон диалога, которые здесь показаны, будут выглядеть так, как будто они создавались вручную.

Шаблон окна диалога программ ABOUT1 выглядит следующим образом:

```
AboutBox DIALOG 20, 20, 160, 80
STYLE WS_POPUP | WS_DLGFRAME
{
       CTEXT "About1"                      -1,  0, 12, 160, 8
       ICON "About1"                       -1,  8, 8,  0,    0
       CTEXT "About Box Demo Program"      -1,  0, 36, 160, 8
       CTEXT "(c) Charles Petzold, 1996"  -1,  0, 48, 160, 8
       DEFPUSHBUTTON "OK"                  IDOK,64,60, 32,  14, WS_GROUP
}
```

В первой строке окну диалога дается имя (в данном случае *AboutBox*). Как и для других ресурсов, вместо имени можно использовать число. За именем следует ключевое слово DIALOG и четыре числа. Первые два — являются координатами *x* и *y* верхнего левого угла окна диалога относительно рабочей области родительского окна при вызове окна диалога программой. Вторые два числа — это ширина и высота окна диалога.

Эти координаты и размеры даются не в пикселях. Значения координат и размеров базируются на специальной системе координат, используемой только для шаблонов окон диалога. Числа основываются на размере символа системного шрифта: координата *x* и ширина выражены в единицах, равных 1/4 средней ширины символа; координата *y* и высота выражены в единицах, равных 1/8 высоты символа. Таким образом, для данного приведенного окна диалога верхний левый угол окна диалога находится на расстоянии 5 символов от левого края

рабочей области родительского окна и на расстоянии 2,5 символов от ее верхнего края. Ширина окна диалога равна 40 символам, а высота – 10 символам.

Такая система координат дает возможность использовать в окне диалога такие координаты и размеры, которые сохранят его общий вид и расположение, независимо от разрешающей способности дисплея. Поскольку высота символов системного шрифта обычно примерно вдвое больше его ширины, то размеры деления по осям *x* и *y* примерно одинаковы.

Функция *GetDialogBaseUnits* позволяет определять размеры системного шрифта, которые используются менеджером окна диалога. Для стандартного монитора VGA (самого часто используемого видеоадаптера в Windows) функция *GetDialogBaseUnits* возвращает ширину символа равную 8 и высоту равную 16 единицам. Поскольку единицы окна диалога соответствуют 1/4 средней ширины символа и 1/8 его высоты, то каждая единица соответствует 2 пикселям монитора VGA. Если идея использования таких единиц измерения окна диалога временами кажется слишком абстрактной, то лучше просто запомнить это правило.

Инструкция STYLE шаблона напоминает поле стиля функции *CreateWindow*. Использование WS_POPUP и WS_DLGFRAME вполне обычно для модальных окон диалога, хотя в дальнейшем мы изучим несколько альтернативных идентификаторов.

Внутри фигурных скобок определяются те дочерние окна элементов управления, которые появятся в окне диалога. В нашем окне диалога используются дочерние окна элементов управления трех типов: CTEXT (текст, выровненный по центру), ICON (значок) и DEFPUSHBUTTON (кнопка, выбираемая по умолчанию). Формат инструкций описания следующий:

```
control-type "text" id, xPos, yPos, xWidth, yHeight [, iStyle]
```

Значение *iStyle* в конце инструкции не является обязательным; оно задает дополнительные стили окна, используя идентификаторы, заданные в заголовочных файлах Windows.

Идентификаторы CTEXT, ICON и DEFPUSHBUTTON используются исключительно в окнах диалога. Они являются сокращенной формой записи идентификаторов класса окна и стиля окна. Например, CTEXT показывает, что класс дочернего окна элементов управления — это "static", а стиль такой:

```
WS_CHILD | SS_CENTER | WS_VISIBLE | WS_GROUP
```

Если с идентификатором WS_GROUP мы сталкиваемся впервые, то стили окна WS_CHILD, SS_CENTER и WS_VISIBLE уже встречались нам в главе 8 при создании статических дочерних окон элементов управления в программе COLORS1.

Что касается значка, то текстовое поле — это имя ресурса значка в программе, который также определен в файле описания ресурсов программы ABOUT1. Текстовое поле кнопки — это тот текст, который появится внутри кнопки на экране. Этот текст аналогичен тексту, заданному во втором параметре функции *CreateWindow*, которая вызывается при создании в программе дочернего окна элемента управления.

Поле *id* представляет собой число, с помощью которого дочернее окно идентифицирует себя при посылке сообщений (обычно эти сообщения WM_COMMAND) своему родительскому окну. Родительским окном этих дочерних окон элементов управления является само окно диалога, которое посылает эти сообщения оконной процедуре, находящейся внутри Windows. Эта оконная процедура посылает эти сообщения процедуре диалогового окна, которая включается в вашу программу. Значения *id* аналогичны идентификаторам дочерних окон, которые использовались в функции *CreateWindow* при создании в главе 8 дочерних окон. Поскольку дочерние окна с текстом и значком не посылают сообщений обратно родительскому окну, эти значения для них устанавливаются равными —1. Значение *id* для кнопки устанавливается равным IDOK, который в заголовочных файлах Windows определяется как 1.

Следующие четыре числа задают положение дочернего окна элемента управления (относительно верхнего левого угла рабочей области окна диалога) и его размер. Положение и размер выражены в единицах, равных 1/4 средней ширины и 1/8 высоты символа системного шрифта. В инструкции ICON высота и ширина игнорируются.

Инструкция DEFPUSHBUTTON шаблона окна диалога в дополнение к стилю окна, заданному ключевым словом DEFPUSHBUTTON, включает в себя стиль окна WS_GROUP. О стиле WS_GROUP (и стиле WS_TABSTOP) будет рассказано несколько позже, при обсуждении второй версии рассмотренной ранее программы — ABOUT2.

Диалоговая процедура

Диалоговая процедура или процедура диалога программы обрабатывает сообщения, получаемые окном диалога. Хотя она очень сильно напоминает оконную процедуру, это не настоящая оконная процедура. Оконная процедура окна диалога находится в Windows. Эта оконная процедура вызывает вашу диалоговую процедуру, передавая ей многие из сообщений, которые получает сама. Здесь представлена процедура диалога программы ABOUT1:

```
BOOL CALLBACK AboutDlgProc(HWND hDlg, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
	switch(iMsg)
	{
	case WM_INITDIALOG :
		return TRUE;

	case WM_COMMAND :
		switch(LOWORD(wParam))
		{
		case IDOK :
		case IDCANCEL :
			EndDialog(hDlg, 0);
			return TRUE;
		}
		break;
	}
	return FALSE;
}
```

Параметры этой функции те же, что и параметры обычной оконной процедуры; как и оконная процедура, процедура диалога должна быть определена как функция типа CALLBACK. Хотя в качестве описателя окна диалога использовался описатель *hDlg*, вместо него можно при желании использовать *hwnd*. Обратите внимание на отличия между этой функцией и оконной процедурой:

- Оконная процедура возвращает значение типа LRESULT; а процедура диалогового окна — значение типа BOOL (определяемое в заголовочных файлах Windows как *int*).
- Если оконная процедура не обрабатывает какое-то сообщение, она вызывает *DefWindowProc*; процедура диалога, если она не обрабатывает сообщение, возвращает FALSE (0), а если обрабатывает, то TRUE (ненулевое значение).
- Процедура диалога не обрабатывает сообщения WM_PAINT и WM_DESTROY. Процедура диалога не получит сообщения WM_CREATE; вместо этого она выполняет инициализацию при обработке специального сообщения WM_INITDIALOG.

Сообщение WM_INITDIALOG является первым сообщением, которое получает процедура диалога. Это сообщение посылается только процедурам диалога. Если процедура диалога возвращает TRUE, то Windows помещает фокус ввода на первое дочернее окно элемента управления, которое имеет стиль WS_TABSTOP (о котором будет рассказано при изучении программы ABOUT2). В нашем окне диалога первым дочерним окном элемента управления, которое имеет стиль WS_TABSTOP, является кнопка. С другой стороны, при обработке сообщения WM_INITDIALOG процедура диалога может использовать функцию *SetFocus* для того, чтобы установить фокус на одно из дочерних окон управления окна диалога, и тогда она должна вернуть значение FALSE.

Единственным оставшимся сообщением, которое обрабатывает процедура окна диалога, является WM_COMMAND. Это то сообщение, которое элемент управления кнопка посылает своему родительскому окну тогда, когда либо на ней производится щелчок мышью, либо нажата клавиша <Spacebar> (пробел) и кнопка имеет фокус ввода. Идентификатор дочернего окна элемента управления (который в шаблоне окна диалога равен IDOK) находится в младшем слове параметра *wParam*. Для этого сообщения процедура диалога вызывает функцию *EndDialog*, которая сообщает Windows о необходимости закрытия окна диалога. Для всех остальных сообщений процедура диалога возвращает FALSE, сообщая оконной процедуре окна диалога внутри Windows, что процедура диалога программы не обрабатывает сообщение.

Сообщения для модального окна диалога не проходят через очередь сообщений программы, поэтому не беспокойтесь о влиянии быстрых клавиш на работу окна диалога.

Вызов окна диалога

При обработке в *WndProc* сообщения WM_CREATE, программа ABOUT1 получает описатель экземпляра программы и сохраняет его в статической переменной:

```
hInstance =((LPCREATESTRUCT) lParam) -> hInstance;
```

Программа ABOUT1 обрабатывает те сообщения WM_COMMAND, в которых младшее слово параметра *wParam* равно IDM_ABOUT. Когда программа его получает, она вызывает функцию *DialogBox*:

```
DialogBox(hInstance, "AboutBox", hwnd, AboutDlgProc);
```

Для этой функции требуется описатель экземпляра (сохраненный при обработке сообщения WM_CREATE), имя окна диалога (как оно определено в файле описания ресурсов), описатель родительского окна окна диалога

(которым является главное окно программы) и адрес процедуры диалога. Если вместо имени шаблона окна диалога используется число, то с помощью макрокоманды MAKEINTRESOURCE его можно преобразовать в строку.

Выбор из меню пункта "About About1..." приводит к выводу на экран окна диалога, показанного на рис. 11.2. Закрыть это окно диалога можно, щелкнув на кнопке OK мышью, нажав клавишу <Spacebar> или <Enter>. При нажатии клавиш <Spacebar> или <Enter> в любом окне диалога, в котором имеется кнопка по умолчанию, Windows посылает диалоговой процедуре сообщение WM_COMMAND, в котором младшее слово параметра *wParam* равно идентификатору заданной по умолчанию кнопки.

Функция *DialogBox*, которая вызывается для вывода на экран окна диалога, не возвращает управление в *WndProc* до тех пор, пока окно диалога не будет закрыто. Возвращаемым значением функции *DialogBox* является второй параметр функции *EndDialog*, которая вызывается в процедуре диалога. (Это значение не используется в программе ABOUT1, но используется в программе ABOUT2.) Затем *WndProc* может передать управление Windows.

Даже при выводе на экран окна диалога, *WndProc* может продолжать получать сообщения. Вы даже можете посылать в *WndProc* сообщения из процедуры диалога. Поскольку главным окном программы ABOUT1 является родительское окно всплывающего окна диалога, то вызов функции *SendMessage* в *AboutDlgProc* должен начинаться следующим образом:

```
SendMessage(GetParent(hDlg), ...);
```

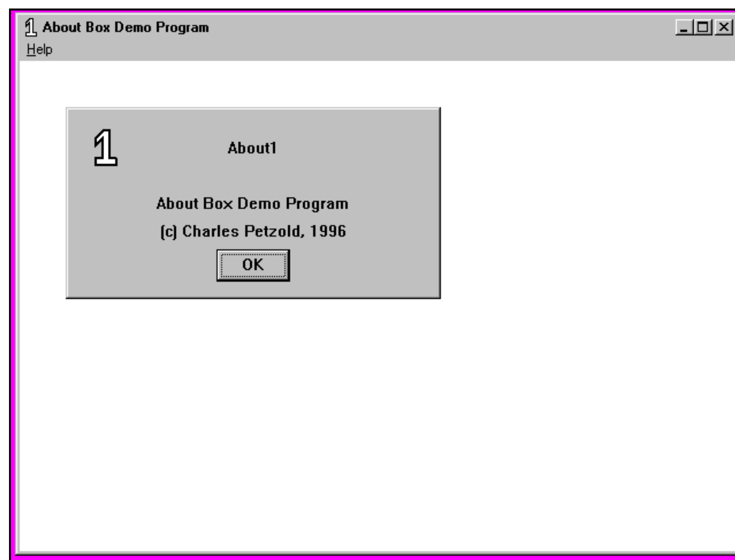


Рис. 11.2 Окно диалога программы ABOUT1

Дополнительная информация о стиле окна диалога

Стиль окна диалога задается в строке STYLE шаблона окна диалога. Для программы ABOUT1 использовался стиль, который наиболее часто используется для модальных окон диалога:

```
STYLE WS_POPUP | WS_DLGFRAME
```

Однако вы можете также поэкспериментировать с другими стилями. Например, можно попытаться использовать такой стиль:

```
STYLE WS_POPUP | WS_CAPTION
```

Такой стиль позволяет создать окно диалога со строкой заголовка и обычной для окна рамкой. Строка заголовка дает возможность пользователю с помощью мыши перемещать окно диалога по экрану. Если используется стиль WS_CAPTION, то координаты *x* и *y*, задаваемые в инструкции DIALOG, являются координатами рабочей области окна диалога относительно верхнего левого угла рабочей области родительского окна. Строка заголовка будет располагаться выше координаты *y*.

При наличии строки заголовка следом за инструкцией STYLE в шаблоне окна диалога можно задать текст заголовка с помощью инструкции CAPTION:

```
CAPTION "Dialog Box Caption"
```

Можно сделать то же самое с помощью функции *SetWindowText* при обработке сообщения WM_INITDIALOG в процедуре диалога:

```
SetWindowText(hDlg, "Dialog Box Caption");
```

Кроме этого, при условии использования стиля WS_CAPTION с помощью стиля WS_SYSMENU к окну диалога можно добавить системное меню:

```
STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU
```

Такой стиль позволяет пользователю выбрать из системного меню опции Move или Close.

Добавление к стилю идентификатора WS_THICKFRAME дает возможность пользователю изменять размер окна диалога, хотя такое изменение размера для окон диалога является делом необычным. Аналогично обстоит дело и с идентификатором WS_MAXIMIZEBOX.

Инструкция STYLE не является необходимой. Если в шаблон не включать инструкцию STYLE или CAPTION, то по умолчанию задается следующий стиль:

```
WS_POPUP | WS_BORDER
```

Но окно такого стиля смотрится хуже. Идентификатор WS_DLGFRAME обеспечивает гораздо более привлекательные результаты. Если к инструкции STYLE добавить инструкцию CAPTION, то по умолчанию задается следующий стиль:

```
WS_POPUP | WS_CAPTION | WS_SYSMENU
```

Кроме этого, меню к окну диалога можно добавить с помощью следующей инструкции в шаблоне окна диалога:

```
MENU menu-name
```

Аргументом является либо имя, либо номер меню в файле описания ресурсов. Меню в модальных окнах диалога – вещь очень необычная. И если оно используется, то необходима уверенность в том, что все идентификаторы меню и дочерних окон элементов управления окна диалога являются уникальными.

Инструкция FONT позволяет использовать в тексте окна диалога какой-либо иной шрифт, отличный от системного.

Хотя оконная процедура окна диалога обычно находится внутри Windows, для обработки сообщений окна диалога можно использовать одну из собственных оконных процедур. Для этого в шаблоне окна диалога необходимо задать имя класса окна:

```
CLASS "class-name"
```

Это делается редко, но тем не менее именно такой подход применяется в представленной далее в этой главе программе HEXCALC.

Когда вызывается функция *DialogBox* с указанием имени шаблона окна диалога, Windows уже имеет почти все необходимое для создания всплывающего окна с помощью обычной функции *CreateWindow*. Windows получает координаты и размеры окна, стиль окна, заголовок и меню из шаблона окна диалога. Описатель экземпляра и описатель родительского окна Windows получает из параметров функции *DialogBox*. Единственной недостающей частью информации является класс окна (если он не задан в шаблоне окна диалога). Для окон диалога Windows регистрирует особый класс окна. Оконная процедура для такого класса окна имеет доступ к указателю на процедуру диалога приложения (который передается в функцию *DialogBox*), таким образом Windows может информировать программу о сообщениях, получаемых этим всплывающим окном. Конечно, вы можете сами создать и поддерживать собственное диалоговое окно путем создания всплывающего окна. Использование функции *DialogBox* значительно облегчает дело.

Дополнительная информация об определении дочерних окон элементов управления

В шаблоне окна управления файла ABOUT1.RC, для определения трех типов дочерних окон элементов управления, которые появляются в окне диалога, использовалась сокращенная запись: CTEXT, ICON и DEFPUSHBUTTON. Можно использовать и другие идентификаторы. Каждый тип соответствует конкретному предопределенному классу окна и стилю окна. В представленной ниже таблице показаны соответствующие каждому типу дочерних окон элементов управления класс окна и стиль окна:

Тип элемента управления	Класс окна	Стиль окна
PUSHBUTTON	button	BS_PUSHBUTTON | WS_TABSTOP
DEFPUSHBUTTON	button	BS_DEFPUSHBUTTON | WS_TABSTOP
CHECKBOX	button	BS_CHECKBOX | WS_TABSTOP
RADIOBUTTON	button	BS_RADIOBUTTON | WS_TABSTOP
GROUPBOX	button	BS_GROUPBOX | WS_TABSTOP
LTEXT	static	SS_LEFT | WS_GROUP
CTEXT	static	SS_CENTER | WS_GROUP
RTEXT	static	SS_RIGHT | WS_GROUP

Тип элемента управления	Класс окна	Стиль окна
ICON	static	SS_ICON
EDITTEXT	edit	ES_LEFT | WS_BORDER | WS_TABSTOP
SCROLLBAR	scrollbar	SBS_HORZ
LISTBOX	listbox	LBS_NOTIFY | WS_BORDER | WS_VSCROLL
COMBOBOX	combobox	CBS_SIMPLE | WS_TABSTOP

Единственной программой, которая понимает эту краткую запись, является компилятор ресурсов (RC). Кроме показанных выше стилей окна, каждое из представленных дочерних окон элементов управления имеет стиль:

```
WS_CHILD | WS_VISIBLE
```

Для всех типов дочерних окон элементов управления, за исключением EDITTEXT, SCROLLBAR, LISTBOX и COMBOBOX, используется следующий формат инструкций, описывающих элементы управления:

```
control-type "text", id, xPos, yPos, xWidth, yHeight [, iStyle]
```

А для типов элементов управления EDITTEXT, SCROLLBAR, LISTBOX и COMBOBOX в формат инструкций определения не входит текстовое поле:

```
control-type id, xPos, yPos, xWidth, yHeight [, iStyle]
```

В обеих этих инструкциях поле *iStyle* не является обязательным.

В главе 8 рассказывалось о правилах задания ширины и высоты предопределенных дочерних окон элементов управления. Полезно было бы вернуться к этой главе и к этим правилам, учитывая, что размеры, которые указываются в шаблонах окон диалога, всегда задаются в единицах 1/4 средней ширины символа и 1/8 его высоты.

Поле стиля (style) инструкций определения окон элементов управления не является обязательным. Оно дает возможность включить в инструкцию другие идентификаторы стиля окна. Например, если необходимо создать флажок с текстом, находящимся слева от квадратного окна флажка, то можно было бы воспользоваться такой инструкцией:

```
CHECKBOX "text", id, xPos, yPos, xWidth, yHeight, BS_LEFTTEXT
```

Хотя сокращенная запись для дочерних окон управления весьма удобна, но она не является исчерпывающей. Например, нельзя создать дочернее окно управления без рамки. По этой причине в компиляторе файла описания ресурсов также определяется и обобщенная форма инструкции окна управления, которая выглядит так:

```
CONTROL "text", id, "class", iStyle, xPos, yPos, xWidth, yHeight
```

Эта инструкция, где можно задать класс окна и полностью определить стиль окна, дает возможность создать любой тип дочернего окна управления. Например, вместо инструкции:

```
PUSHBUTTON    "OK",  IDOK, 10, 20, 32, 14
```

можно использовать инструкцию:

```
CONTROL "OK", IDOK, "button", WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP, 10, 20, 32, 14
```

При компиляции файла описания ресурсов две этих инструкции кодируются одинаково, как в файле с расширением .RES, так и в файле с расширением .EXE.

Если инструкция CONTROL используется в шаблоне окна диалога, нет необходимости включать в нее стили WS_CHILD и WS_VISIBLE. Windows включает их в стиль окна при создании дочерних окон. Кроме этого формат инструкции CONTROL облегчает понимание того, что делает менеджер окна диалога Windows, когда он создает окно диалога. Во-первых, как уже ранее говорилось, он создает всплывающее окно, родительское окно которого определяется описателем окна, заданным в функции *DialogBox*. Затем для каждой инструкции элемента управления в шаблоне диалога, менеджер окна диалога создает дочернее окно. Родительским окном каждого из этих дочерних окон управления является всплывающее окно диалога. Приведенная выше инструкция CONTROL преобразуется в вызов функции *CreateWindow*, которая выглядит следующим образом:

```
CreateWindow("button", "OK",
       WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP,
       10 * cxChar / 4, 20 * cyChar / 8,
       32 * cxChar / 4,14 * cyChar / 8,
       hDlg, IDOK, hInstance, NULL);
```

где *cxChar* и *cyChar* — это ширина и высота символа системного шрифта в пикселях. Параметр *hDlg* является возвращаемым значением функции *CreateWindow*, которая создает окно диалога. Параметр *hInstance* получен при первом вызове функции *DialogBox*.

Более сложное окно диалога

В программе ABOUT1 проиллюстрированы основы работы с простым окном диалога; теперь попытаемся сделать нечто более сложное. Программа ABOUT2, представленная на рис. 11.3, показывает, как работать с дочерними окнами элементов управления (в данном случае с группой переключателей) внутри процедуры окна диалога, и как рисовать в рабочей области окна диалога.

ABOUT2.MAK

```
#----------------------
# ABOUT2.MAK make file
#----------------------

about2.exe : about2.obj about2.res
     $(LINKER) $(GUIFLAGS) -OUT:about2.exe about2.obj about2.res $(GUILIBS)

about2.obj : about2.c about2.h
     $(CC) $(CFLAGS) about2.c

about2.res : about2.rc about2.h about2.ico
     $(RC) $(RCVARS) about2.rc
```

ABOUT2.C

```
/*------------------------------------------
   ABOUT2.C -- About Box Demo Program No. 2
               (c) Charles Petzold, 1996
  ------------------------------------------*/

#include <windows.h>
#include "about2.h"

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

int iCurrentColor  = IDD_BLACK,
    iCurrentFigure = IDD_RECT;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char  szAppName[] = "About2";
     MSG          msg;
     HWND         hwnd;
     WNDCLASSEX   wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(hInstance, szAppName);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = szAppName;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(hInstance, szAppName);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "About Box Demo Program",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);
```

```
     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

void PaintWindow(HWND hwnd, int iColor, int iFigure)
     {
     static COLORREF crColor[8] = { RGB(0,     0, 0), RGB(  0,   0, 255),
                                    RGB(0,   255, 0), RGB(  0, 255, 255),
                                    RGB(255,   0, 0), RGB(255,   0, 255),
                                    RGB(255, 255, 0), RGB(255, 255, 255) };
     HBRUSH          hBrush;
     HDC             hdc;
     RECT            rect;

     hdc = GetDC(hwnd);
     GetClientRect(hwnd, &rect);
     hBrush = CreateSolidBrush(crColor[iColor - IDD_BLACK]);
     hBrush =(HBRUSH) SelectObject(hdc, hBrush);

     if(iFigure == IDD_RECT)
          Rectangle(hdc, rect.left, rect.top, rect.right, rect.bottom);
     else
          Ellipse  (hdc, rect.left, rect.top, rect.right, rect.bottom);

     DeleteObject(SelectObject(hdc, hBrush));
     ReleaseDC(hwnd, hdc);
     }

void PaintTheBlock(HWND hCtrl, int iColor, int iFigure)
     {
     InvalidateRect(hCtrl, NULL, TRUE);
     UpdateWindow(hCtrl);
     PaintWindow(hCtrl, iColor, iFigure);
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HINSTANCE hInstance;
     PAINTSTRUCT      ps;

     switch(iMsg)
          {
          case WM_CREATE :
               hInstance =((LPCREATESTRUCT) lParam)->hInstance;
               return 0;

          case WM_COMMAND :
               switch(LOWORD(wParam))
                    {
                    case IDM_ABOUT :
                         if(DialogBox(hInstance, "AboutBox", hwnd,
                                        AboutDlgProc))
                              InvalidateRect(hwnd, NULL, TRUE);
                         return 0;
                    }
               break;
```

```
          case WM_PAINT :
               BeginPaint(hwnd, &ps);
               EndPaint(hwnd, &ps);

               PaintWindow(hwnd, iCurrentColor, iCurrentFigure);
               return 0;

          case WM_DESTROY :
               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }

BOOL CALLBACK AboutDlgProc(HWND hDlg, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HWND hCtrlBlock;
     static int  iColor, iFigure;

     switch(iMsg)
          {
          case WM_INITDIALOG :
               iColor  = iCurrentColor;
               iFigure = iCurrentFigure;

               CheckRadioButton(hDlg, IDD_BLACK, IDD_WHITE, iColor);
               CheckRadioButton(hDlg, IDD_RECT,  IDD_ELL,   iFigure);

               hCtrlBlock = GetDlgItem(hDlg, IDD_PAINT);

               SetFocus(GetDlgItem(hDlg, iColor));
               return FALSE;

          case WM_COMMAND :
               switch(LOWORD(wParam))
                    {
                    case IDOK :
                         iCurrentColor  = iColor;
                         iCurrentFigure = iFigure;
                         EndDialog(hDlg, TRUE);
                         return TRUE;
                    case IDCANCEL :
                         EndDialog(hDlg, FALSE);
                         return TRUE;

                    case IDD_BLACK :
                    case IDD_RED :
                    case IDD_GREEN :
                    case IDD_YELLOW :
                    case IDD_BLUE :
                    case IDD_MAGENTA :
                    case IDD_CYAN :
                    case IDD_WHITE :
                         iColor = LOWORD(wParam);
                         CheckRadioButton(hDlg, IDD_BLACK, IDD_WHITE, LOWORD
                                          (wParam));
                         PaintTheBlock(hCtrlBlock, iColor, iFigure);
                         return TRUE;

                    case IDD_RECT :
                    case IDD_ELL :
                         iFigure = LOWORD(wParam);
```

```
                        CheckRadioButton(hDlg, IDD_RECT, IDD_ELL, LOWORD
                                   (wParam));
                        PaintTheBlock(hCtrlBlock, iColor, iFigure);
                        return TRUE;
                   }
              break;

         case WM_PAINT :
              PaintTheBlock(hCtrlBlock, iColor, iFigure);
              break;
         }
     return FALSE;
     }
```

ABOUT2.RC

```
/*---------------------------
   ABOUT2.RC resource script
  ---------------------------*/

#include <windows.h>
#include "about2.h"

about2 ICON about2.ico

About2 MENU
     {
     POPUP "&Help"
          {
          MENUITEM "&About About2...",       IDM_ABOUT
          }
     }

#define TABGRP(WS_TABSTOP | WS_GROUP)

AboutBox DIALOG 20, 20, 140, 188
     STYLE WS_POPUP | WS_DLGFRAME
     {
     CTEXT       "About2"       -1,           0,  12, 140,   8
     ICON        "About2"       -1,           8,   8,   0,   0
     CTEXT       "About Box Demo Program" -1, 4,  36, 130,   8
     CTEXT       ""             IDD_PAINT,   68,  54,  60,  60
     GROUPBOX    "&Color"       -1,           4,  50,  54, 112
     RADIOBUTTON "&Black"       IDD_BLACK,    8,  60,  40,  12, TABGRP
     RADIOBUTTON "B&lue"        IDD_BLUE,     8,  72,  40,  12
     RADIOBUTTON "&Green"       IDD_GREEN,    8,  84,  40,  12
     RADIOBUTTON "Cya&n"        IDD_CYAN,     8,  96,  40,  12
     RADIOBUTTON "&Red"         IDD_RED,      8, 108,  40,  12
     RADIOBUTTON "&Magenta"     IDD_MAGENTA,  8, 120,  40,  12
     RADIOBUTTON "&Yellow"      IDD_YELLOW,   8, 132,  40,  12
     RADIOBUTTON "&White"       IDD_WHITE,    8, 144,  40,  12
     GROUPBOX    "&Figure"      -1,          68, 120,  60,  40, WS_GROUP
     RADIOBUTTON "Rec&tangle"   IDD_RECT,    72, 134,  50,  12, TABGRP
     RADIOBUTTON "&Ellipse"     IDD_ELL,     72, 146,  50,  12
     DEFPUSHBUTTON "OK"         IDOK,        20, 168,  40,  14, WS_GROUP
     PUSHBUTTON  "Cancel"       IDCANCEL,    80, 168,  40,  14, WS_GROUP
     }
```

ABOUT2.H

```
/*----------------------
   ABOUT2.H header file
  ----------------------*/

#define IDM_ABOUT      1
```

```
#define IDD_BLACK     10
#define IDD_BLUE      11
#define IDD_GREEN     12
#define IDD_CYAN      13
#define IDD_RED       14
#define IDD_MAGENTA   15
#define IDD_YELLOW    16
#define IDD_WHITE     17

#define IDD_RECT      20
#define IDD_ELL       21

#define IDD_PAINT     30
```

ABOUT2.ICO

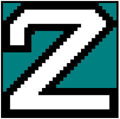


Рис. 11.3 Программа ABOUT2

В окне About программы ABOUT2 имеются две группы переключателей. Одна группа используется для выбора цвета, а другая — для выбора либо прямоугольника, либо эллипса. Прямоугольник или эллипс выводятся в окне диалога, окрашенные в соответствии с выбранным цветом. При нажатии кнопки OK окно диалога закрывается, и оконная процедура программы рисует в своей рабочей области выбранную фигуру. При нажатии кнопки Cancel рабочая область главного окна остается без изменений. Окно диалога показано на рис. 11.4. Хотя для двух кнопок в окне диалога программы ABOUT2 используются предопределенные идентификаторы IDOK и IDCANCEL, для каждой группы переключателей имеется свой идентификатор, который начинается с букв IDD (идентификатор дочернего окна управления окна диалога, ID dialog box control). Эти идентификаторы определяются в заголовочном файле ABOUT2.H.

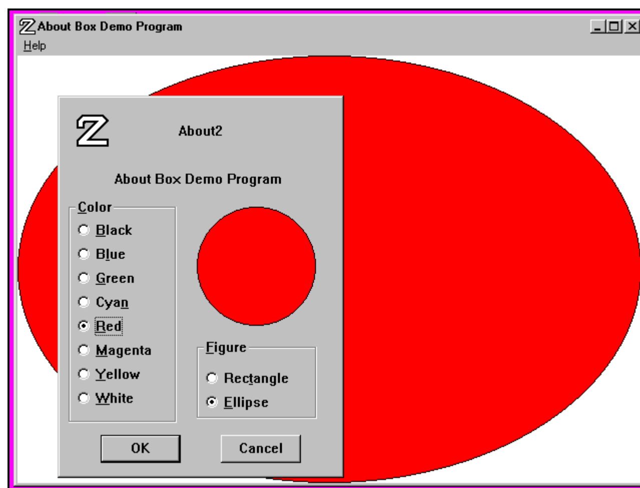


Рис. 11.4 Окно диалога программы ABOUT2

Работа с дочерними элементами управления окна диалога

В главе 8 было показано, что большинство дочерних окон элементов управления посылают своему родительскому окну сообщения WM_COMMAND. (Исключение составляют полосы прокрутки.) Кроме этого было показано, что родительское окно может изменять состояние своих дочерних окон элементов управления (например, включать или выключать переключатели или флажки), посылая дочерним окнам управления сообщения. Аналогичным образом можно изменять состояние дочерних окон управления в процедуре диалога. Например, при наличии нескольких групп переключателей с помощью сообщений можно включать и выключать переключатели в каждой группе. Кроме этого для работы с окнами управления в окнах диалога Windows обеспечивает нас еще несколькими возможностями. Рассмотрим один из способов взаимодействия процедуры диалога и дочерних окон управления.

Шаблон окна диалога программы ABOUT2, представленной на рис. 11.3, находится в файле описания ресурсов ABOUT2.RC. Элемент управления GROUPBOX — это просто рамка с заголовком (Color или Figure), окружающая каждую из двух групп переключателей. Восемь переключателей первой группы, как и два переключателя второй, являются взаимоисключающими.

При щелчке мышью на одном из переключателей (или при нажатии клавиши <Spacebar>, когда переключатель имеет фокус ввода) дочернее окно посылает своему родительскому окну сообщение WM_COMMAND, в котором младшее слово параметра *wParam* равно идентификатору окна элемента управления. Старшим словом параметра *wParam* является код опознавания, а значением параметра *lParam* — описатель окна управления. Для переключателя код уведомления равен либо BN_CLICKED, либо 0. Затем процедура окна диалога внутри Windows передает это сообщение WM_COMMAND в процедуру диалога программы ABOUT2.C. Если для одного из переключателей процедура диалога получает сообщение WM_COMMAND, она устанавливает контрольную метку на этом переключателе и снимает контрольные метки с остальных переключателей данной группы.

В главе 8 было показано, что для включения и выключения переключателя необходимо, чтобы дочернему окну управления было отправлено сообщение BM_SETCHECK. Поэтому для установки переключателя используется следующий оператор:

```
SendMessage(hwndCtrl, BM_SETCHECK, 1, 0);
```

А для снятия выключения, такой:

```
SendMessage(hwndCtrl, BM_SETCHECK, 0, 0);
```

Параметр *hwndCtrl* является описателем дочернего окна элемента управления.

Но этот способ приводит к небольшой проблеме, поскольку в процедуре окна диалога неизвестны описатели всех окон-переключателей. Известен только тот идентификатор переключателя, от которого приходит сообщение. К счастью, в Windows имеется функция для получения описателя окна элемента управления окна диалога, в которой используются описатель окна диалога и идентификатор элемента управления:

```
hwndCtrl = GetDlgItem(hDlg, id);
```

(Кроме этого, с помощью функции *GetWindowLong* можно получить значение идентификатора окна управления, зная описатель этого окна:

```
id = GetWindowLong(hwndCtrl, GWL_ID);
```

но это выражение используется редко.)

Обратите внимание, что в представленном на рис. 11.3 заголовочном файле ABOUT2.H, значения идентификаторов для восьми цветов заданы последовательно от IDD_BLACK до IDD_WHITE. Такая упорядоченность помогает обрабатывать сообщения WM_COMMAND от переключателей. Для включения и выключения переключателей в процедуре окна диалога можно использовать примерно такой код:

```
static int iColor;

[другие строки программы]

case WM_COMMAND:
       switch(LOWORD(wParam))
       {

       [другие строки программы]

       case IDD_BLACK:
       case IDD_RED:
       case IDD_GREEN:
       case IDD_YELLOW:
       case IDD_BLUE:
       case IDD_MAGENTA:
       case IDD_CYAN:
       case IDD_WHITE:
              iColor = LOWORD(wParam);

              for(i = IDD_BLACK; i <= IDD_WHITE; i++)
                     SendMessage(GetDlgItem(hDlg, i), BM_SETCHECK, i == LOWORD(wParam), 0);
              return TRUE;

       [другие строки программы]
```

Такой подход работает вполне удовлетворительно. В *iColor* сохраняется новое значение цвета, а также запускается цикл по идентификаторам всех восьми цветов. Для каждого переключателя получают описатель окна, и каждому

описателю с помощью функции *SendMessage* посылается сообщение BM_SETCHECK. Значение *wParam* этого сообщения устанавливается в 1 для того переключателя, который стал источником сообщения WM_COMMAND, отправленного оконной процедуре окна диалога.

Первым усовершенствованием является специальная функция *SendDlgItemMessage*:

```
SendDlgItemMessage(hDlg, id, iMsg, wParam, lParam);
```

Эта функция эквивалентна следующей:

```
SendMessage(GetDlgItem(hDlg, id), id, wParam, lParam);
```

Теперь цикл будет выглядеть так:

```
for(i = IDD_BLACK; i <= IDD_WHITE; i++)
    SendDlgItemMessage(hDlg, i, BM_SETCHECK, i == LOWORD(wParam), 0);
```

Это несколько лучше. Но реальное улучшение появляется при использовании функции *CheckRadioButton*:

```
CheckRadioButton(hDlg, idFirst, idLast, idCheck);
```

Эта функция снимает контрольные метки со всех переключателей с идентификаторами от *idFirst* до *idLast*, за исключением переключателя с идентификатором *idCheck*, который, наоборот, включается. Идентификаторы должны быть заданы последовательно. Теперь, используя эту функцию, можно избавиться от целого цикла:

```
CheckRadioButton(hDlg, IDD_BLACK, IDD_WHITE, LOWORD(wParam));
```

Именно так сделано в процедуре окна диалога программы ABOUT2.

Похожая функция имеется и для работы с флажками. Если в окне диалога создается элемент управления CHECKBOX, то снять или установить контрольную метку можно с помощью следующей функции:

```
CheckDlgButton(hDlg, idCheckbox, iCheck);
```

Если *iCheck* устанавливается в 1, флажок включается, если в 0 – выключается. Чтобы получить состояние флажка в окне диалога, можно использовать такую функцию:

```
iCheck = IsDlgButtonChecked(hDlg, idCheckbox);
```

Вы можете либо сохранить текущее состояние флажка в статической переменной внутри диалоговой процедуры, либо использовать такую конструкцию про обработке сообщения WM_COMMAND:

```
CheckDlgButton(hDlg, idCheckbox, !IsDlgButtonChecked(hDlg, idCheckbox));
```

Если элемент управления определен с флагом BS_AUTOCHECKBOX, то нет необходимости обрабатывать сообщение WM_COMMAND. Текущее состояние кнопки перед закрытием окна диалога можно получить просто с помощью функции *IsDlgButtonChecked*.

Кнопки OK и Cancel

В программе ABOUT2 имеется две кнопки, помеченные как OK и Cancel. В шаблоне окна диалога файла описания ресурсов ABOUT2.RC кнопка OK имеет идентификатор IDOK (определенный в заголовочных файлах Windows как 1), а кнопка Cancel имеет идентификатор IDCANCEL (определенный как 2). При этом кнопкой по умолчанию является кнопка OK:

```
DEFPUSHBUTTON "OK"   IDOK,       20,   168,   40,   14,   WS_GROUP
PUSHBUTTON    "Cancel"IDCANCEL,  80,   168,   40,   14,   WS_GROUP
```

Такое соглашение для кнопок OK и Cancel в окнах диалога вполне обычно; наличие выбираемой по умолчанию кнопки OK помогает работе с интерфейсом клавиатуры. И вот почему: обычно окно диалога закрывается с помощью либо щелчка мыши на одной из этих кнопок, либо нажатия клавиши <Spacebar>, когда нужная кнопка имеет фокус ввода. Кроме этого оконная процедура окна диалога генерирует сообщение WM_COMMAND при нажатии клавиши <Enter>, независимо от того, какое из окон элементов управления имеет фокус ввода. Младшее слово параметра *wParam* соответствует значению идентификатора выбираемой по умолчанию кнопки окна диалога до тех пор, пока другая кнопка не получит фокус ввода. В этом случае младшее слово параметра *wParam* получает значение идентификатора той кнопки, которая имеет фокус ввода. Если в диалоговом окне нет кнопки, выбираемой по умолчанию, то Windows посылает диалоговой процедуре сообщение WM_COMMAND с младшим словом параметра *wParam* равным IDOK. А при нажатии клавиши <Esc> или <Ctrl>+<Break>, Windows посылает процедуре окна диалога сообщение WM_COMMAND с младшим словом параметра *wParam* равным IDCANCEL. Таким образом нет необходимости добавлять в процедуру окна диалога отдельную логику работы с клавиатурой, поскольку те нажатия клавиш, которые обычно приводят к закрытию окна диалога, преобразуются Windows в сообщения WM_COMMAND для этих двух кнопок.

Функция *AboutDlgProc*, вызывая функцию *EndDialog*, обрабатывает эти два сообщения WM_COMMAND:

```
switch(LOWORD(wParam))
{
case IDOK:
        iCurrentColor = iColor;
        iCurrentFigure = iFigure;
        EndDialog(hDlg, TRUE);
        return TRUE;

case IDCANCEL:
        EndDialog(hDlg, FALSE);
        return TRUE;
```

Когда оконная процедура программы ABOUT2 рисует в рабочей области своей программы прямоугольник или эллипс, она использует глобальные переменные *iCurrentColor* и *iCurrentFigure*. Для рисования фигуры в окне диалога в функции *AboutDlgProc* используются локальные статические переменные *iColor* и *iFigure*.

Обратите внимание на отличие второго параметра у двух функций *EndDialog*. Это значение передается обратно в *WndProc* в качестве возвращаемого значения функции *DialogBox*:

```
case IDM_ABOUT:
        if(DialogBox(hInstance, "AboutBox", hwnd, AboutDlgProc))
                InvalidateRect(hwnd, NULL, TRUE);
        return 0;
```

Если функция *DialogBox* возвращает TRUE (т. е. ненулевое значение), что означает нажатие кнопки OK, то *WndProc* должна обновить рабочую область, нарисовав новую фигуру новым цветом. Эти фигура и цвет запоминаются в глобальных переменных *iCurrentColor* и *iCurrentFigure* в функции *AboutDlgProc*, когда она получает сообщение WM_COMMAND с младшим словом параметра *wParam* равным IDOK. Если функция *DialogBox* возвращает FALSE, то родительское окно продолжает использовать прежние значения глобальных переменных *iCurrentColor* и *iCurrentFigure*.

Величины TRUE и FALSE, как правило, используются в функции *EndDialog* для того, чтобы просигнализировать оконной процедуре родительского окна о том, какой из кнопок (OK или Cancel) пользователь закрывает окно диалога. Однако параметром функции *EndDialog* фактически является *int*, поэтому таким образом можно передавать гораздо больше информации, чем просто значения TRUE или FALSE.

Позиции табуляции и группы

В главе 8 для того, чтобы добавить в программу COLORS1 возможность переходить с одной полосы прокрутки на другую с помощью клавиши <Tab>, использовалась техника введения новой оконной процедуры. В окне диалога необходимость применения этой техники отпадает: Windows обеспечивает всю логику, необходимую для перемещения фокуса ввода с одного окна элемента управления на другое. Однако для этого необходимо включить в шаблон окна диалога стили WS_TABSTOP и WS_GROUP. Для всех дочерних окон элементов управления, к которым необходим доступ с помощью клавиши <Tab>, задается стиль окна WS_TABSTOP. Если вернуться к приведенной в этой главе таблице стилей элементов управления окон диалога, то стоит обратить внимание, что многие дочерние элементы управления по умолчанию содержат стиль WS_TABSTOP, в то время как другие нет. Обычно дочерние элементы управления, которые не содержат стиль WS_TABSTOP (а конкретно статические элементы управления) не должны получать фокус ввода, поскольку он там не нужен. До тех пор, пока фокус ввода не установлен на определенное окно элемента управления, при обработке сообщения WM_INITDIALOG в ответ на него приходит FALSE, и Windows устанавливает фокус ввода на первое окно управления в окне диалога, которое имеет стиль WS_TABSTOP.

Вторая возможность работы с клавиатурой, которую Windows предоставляет в окне диалога, включает в себя использование клавиш управления курсором. Эта возможность особенно важна для групп переключателей. После того как для перемещения к помеченному в данный момент контрольной меткой переключателю внутри группы использовалась клавиша <Tab>, для передачи фокуса ввода от одного переключателя внутри группы к другому необходимо использовать клавиши управления курсором. Этого можно добиться, если использовать стиль окна WS_GROUP. Для конкретных последовательностей дочерних элементов управления в шаблоне окна диалога Windows будет использовать клавиши управления курсором для передачи фокуса ввода с первого элемента управления, имеющего стиль WS_GROUP, на следующие элементы управления группы (но до следующего элемента управления, имеющего стиль WS_GROUP). При достижении последнего элемента управления группы, Windows будет циклически переходить снова на первый и т. д.

По умолчанию дочерние окна управления LTEXT, CTEXT, RTEXT и ICON включают стиль WS_GROUP, который обычно помечает конец группы. Для дочерних окон управления других типов часто необходимо добавлять стиль WS_GROUP.

Рассмотрим шаблон окна диалога в файле ABOUT2.RC:

```
AboutBox DIALOG 20, 20, 140, 188
STYLE WS_POPUP | WS_DLGFRAME
{
       CTEXT          "About2"                       -1,             0,  12, 140,   8
       ICON           "About2"                       -1,             8,   8,   0,   0
       CTEXT          "About Box Demo Program"       -1,             4,  36, 130, 8
       CTEXT          ""                             IDD_PAINT,     68,  54,  60,  60
       GROUPBOX       "&Color"                       -1,             4,  50,  54, 112
       RADIOBUTTON    "&Black"                       IDD_BLACK,      8,  60,  40,  12, TABGRP
       RADIOBUTTON    "B&lue"                        IDD_BLUE,       8,  72,  40,  12
       RADIOBUTTON    "&Green"                       IDD_GREEN,      8,  84,  40,  12
       RADIOBUTTON    "Cya&n"                        IDD_CYAN,       8,  96,  40,  12
       RADIOBUTTON    "&Red"                         IDD_RED,        8, 108,  40,  12
       RADIOBUTTON    "&Magenta"                     IDD_MAGENTA,    8, 120,  40,  12
       RADIOBUTTON    "&Yellow"                      IDD_YELLOW,     8, 132,  40,  12
       RADIOBUTTON    "&White"                       IDD_WHITE,      8, 144,  40,  12
       GROUPBOX       "&Figure"                      -1,            68, 120,  60,  40, WS_GROUP
       RADIOBUTTON    "Rec&tangle"                   IDD_RECT,      72, 134,  50,  12, TABGRP
       RADIOBUTTON    "&Ellipse"                     IDD_ELL,       72, 146,  50,  12
       DEFPUSHBUTTON  "OK"                           IDOK,          20, 168,  40,  14, WS_GROUP
       PUSHBUTTON     "Cancel"                       IDCANCEL,      80, 168,  40,  14, WS_GROUP
}
```

Для лучшего восприятия шаблона, в файле ABOUT2.RC идентификатор TABGRP определяется как сочетание идентификаторов WS_TABSTOP и WS_GROUP:

```
#define TABGRP       (WS_TABSTOP | WS_GROUP)
```

Четыре дочерних окна управления, которые имеют стиль WS_TABSTOP — это первые переключатели каждой группы (стиль задан явно) и две кнопки (стиль задан по умолчанию). При появлении окна диалога на экране перемещаться от одного из этих четырех окон управления к другому можно с помощью клавиши <Tab>.

Внутри каждой группы переключателей для смены фокуса ввода и контрольной отметки используются клавиши управления курсором. Например, первый переключатель (Black) группы Color и группы Figure имеют стиль WS_GROUP. Это означает, что клавишами управления курсором можно перемещать фокус ввода с переключателя Black по всем переключателям группы вплоть до группы Figure (но не переходя в нее). Аналогично, поскольку первый переключатель (Rectangle) группы Figure и кнопка DEFPUSHBUTTON имеют стиль WS_GROUP, можно использовать клавиши управления курсором для перемещения от одного переключателя группы (Rectangle) к другому (Ellipse). Обе кнопки имеют стиль WS_GROUP для предотвращения перемещения фокуса ввода клавишами управления курсором, если какая-то из этих кнопок его имеет.

При работе программы ABOUT2 менеджер окна диалога в Windows совершает с двумя группами переключателей нечто таинственное. Как и ожидалось, клавиши управления курсором внутри группы переключателей перемещают фокус ввода и посылают процедуре окна диалога сообщение WM_COMMAND. Но, когда внутри группы включается переключатель, Windows устанавливает ему стиль WS_TABSTOP. Если в следующий раз с помощью клавиши <Tab> переключиться в эту группу, Windows установит фокус ввода на этот выбранный переключатель.

Знак амперсанта (&) в поле текста приводит к подчеркиванию следующей за ним буквы и добавляет к интерфейсу клавиатуры еще одну возможность. Фокус ввода можно передать любому выбранному переключателю, нажав клавишу с подчеркнутой буквой. Нажав клавишу <C> (для группы Color) или <F> (для группы Figure), можно передать фокус ввода текущему переключателю данной группы.

Хотя, как правило, программисты позволяют менеджеру окна диалога брать все это на себя, в Windows имеются две функции, которые дают возможность определить следующую или предыдущую позицию табуляции или окна группы. Этими функциями являются:

```
hwndCtrl = GetNextDlgTabItem(hDlg, hwndCtrl, bPrevious);
```

и

```
hwndCtrl = GetNextDlgGroupItem(hDlg, hwndCtrl, bPrevious);
```

Если параметр *bPrevious* равен TRUE, то функции возвращают предыдущую позицию табуляции или окна группы, если FALSE — то следующую.

Рисование в окне диалога

Помимо всего прочего, программа ABOUT2 делает и кое-что необычное: рисует в окне диалога. Давайте разберемся как. Внутри шаблона окна диалога в файле ABOUT2.RC определяется пустое текстовое окно управления с размерами и положением, определяющими область, в которой предполагается рисование:

```
CTEXT ""  IDD_PAINT, 68, 54, 60, 60
```

Ширина этой области равна 15 символов, а высота 7,5 символов. Поскольку в этом окне управления текста нет, все, что делает оконная процедура класса "static" состоит в обновлении фона при перерисовке дочернего окна управления.

Когда текущий цвет или фигура изменяется, или когда окно диалога получает сообщение WM_PAINT, процедура диалога вызывает функцию *PaintTheBlock*, вызов которой в программе ABOUT2.C выглядит так:

```
PaintTheBlock(hCtrlBlock, iColor, iFigure);
```

Описатель окна *hCtrlBlock*, являющийся первым параметром функции, был получен при обработке сообщения WM_INITDIALOG:

```
hCtrlBlock = GetDlgItem(hDlg, IDD_PAINT);
```

Далее представлено тело функции *PaintTheBlock*:

```
void PaintTheBlock(HWND hCtrl, int iColor, int iFigure)
{
       InvalidateRect(hCtrl, NULL, TRUE);
       UpdateWindow(hCtrl);
       PaintWindow(hCtrl, iColor, iFigure);
}
```

Таким образом, функция делает недействительным дочернее окно управления, помечает его как обновленное, а затем вызывает другую функцию программы ABOUT2 с именем *PaintWindow*.

Функция *PaintWindow* получает описатель контекста устройства окна *hCtrl* и рисует выбранную фигуру, закрашивая ее кистью выбранным пользователем цветом. Размер дочернего окна управления получен с помощью функции *GetClientRect*. Хотя в шаблоне окна диалога размер дочернего окна управления определяется в символах, *GetClientRect* возвращает размеры в пикселях. Кроме нее, можно использовать функцию *MapDialogRect*, которая преобразует символьные координаты окна диалога в пиксельные координаты рабочей области.

Фактически, мы рисуем не в рабочей области окна диалога, а в рабочей области дочернего окна элемента управления. Когда бы окно диалога ни получило сообщение WM_PAINT, дочернее окно управления является недействительным, затем обновляется и его рабочая область становится действительной, что позволяет рисовать в нем.

Использование с окном диалога других функций

Большинство функций, которые используются с дочерними окнами, также можно применять и с дочерними окнами элементов управления в окне диалога. Например, используя функцию *MoveWindow*, можно двигать окно управления по окну диалога и заставлять пользователя гоняться за ним с мышью.

Иногда возникает необходимость в зависимости от установок других окон управления, в динамическом режиме делать доступными или недоступными определенные элементы управления в окне диалога. Вызов следующей функции:

```
EnableWindow(hwndCtrl, bEnable);
```

делает окно элемента управления доступным, когда параметр *bEnable* равен TRUE (ненулевое значение) и недоступным, когда *bEnable* равен FALSE (0). Если окно элемента управления недоступно, то оно не реагирует ни на клавиатуру, ни на мышь. Нельзя делать недоступным окно элемента управления, имеющее фокус ввода.

Определение собственных окон управления

Хотя Windows предлагает массу возможностей для поддержки окна диалога и дочерних окон элементов управления, имеются различные методы, позволяющие вставлять в этот процесс фрагменты собственного кода. Только что рассматривался метод, дающий возможность рисовать в окне диалога. Можно также использовать технику введения новой оконной процедуры (о котором рассказывалось в главе 8) для изменения поведения дочерних окон элементов управления.

Также можно определять собственные дочерние окна элементов управления и использовать их в окне диалога. Например, предположим, что пользователь предпочитает не прямоугольные кнопки, а эллипсоидные. Создать их

можно путем регистрации класса окна и использования собственной оконной процедуры для обработки сообщений от такого специального дочернего окна. Затем этот класс окна задается в инструкции CONTROL шаблона окна диалога. Именно это делается в программе ABOUT3, представленной на рис. 11.5.

ABOUT3.MAK

```
#----------------------
# ABOUT3.MAK make file
#----------------------

about3.exe : about3.obj about3.res
     $(LINKER) $(GUIFLAGS) -OUT:about3.exe about3.obj about3.res $(GUILIBS)

about3.obj : about3.c about3.h
     $(CC) $(CFLAGS) about3.c

about3.res : about3.rc about3.h about3.ico
     $(RC) $(RCVARS) about3.rc
```

ABOUT3.C

```
/*------------------------------------------
   ABOUT3.C -- About Box Demo Program No. 3
              (c) Charles Petzold, 1996
  ------------------------------------------*/

#include <windows.h>
#include "about3.h"

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
BOOL    CALLBACK AboutDlgProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK EllipPushWndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char  szAppName[] = "About3";
     MSG          msg;
     HWND         hwnd;
     WNDCLASSEX   wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(hInstance, szAppName);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = szAppName;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(hInstance, szAppName);

     RegisterClassEx(&wndclass);

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = EllipPushWndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = NULL;
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH)(COLOR_BTNFACE + 1);
     wndclass.lpszMenuName  = NULL;
```

```
     wndclass.lpszClassName = "EllipPush";
     wndclass.hIconSm       = NULL;

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "About Box Demo Program",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HINSTANCE hInstance;

     switch(iMsg)
          {
          case WM_CREATE :
               hInstance =((LPCREATESTRUCT) lParam)->hInstance;
               return 0;
          case WM_COMMAND :
               switch(LOWORD(wParam))
                    {
                    case IDM_ABOUT :
                         DialogBox(hInstance, "AboutBox", hwnd,
                                   AboutDlgProc);
                         return 0;
                    }
               break;

          case WM_DESTROY :
               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }

BOOL CALLBACK AboutDlgProc(HWND hDlg, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     switch(iMsg)
          {
          case WM_INITDIALOG :
               return TRUE;

          case WM_COMMAND :
               switch(LOWORD(wParam))
                    {
                    case IDOK :
                         EndDialog(hDlg, 0);
                         return TRUE;
                    }
               break;
          }
```

```
     return FALSE;
     }

LRESULT CALLBACK EllipPushWndProc(HWND hwnd, UINT iMsg, WPARAM wParam,
                                                         LPARAM lParam)
     {
     char        szText[40];
     HBRUSH      hBrush;
     HDC         hdc;
     PAINTSTRUCT ps;
     RECT        rect;

     switch(iMsg)
          {
          case WM_PAINT :
               GetClientRect(hwnd, &rect);
               GetWindowText(hwnd, szText, sizeof(szText));
               hdc = BeginPaint(hwnd, &ps);

               hBrush = CreateSolidBrush(GetSysColor(COLOR_WINDOW));
               hBrush =(HBRUSH) SelectObject(hdc, hBrush);
               SetBkColor(hdc, GetSysColor(COLOR_WINDOW));
               SetTextColor(hdc, GetSysColor(COLOR_WINDOWTEXT));

               Ellipse(hdc, rect.left, rect.top, rect.right, rect.bottom);
               DrawText(hdc, szText, -1, &rect,
                              DT_SINGLELINE | DT_CENTER | DT_VCENTER);

               DeleteObject(SelectObject(hdc, hBrush));

               EndPaint(hwnd, &ps);
               return 0;

          case WM_KEYUP :
               if(wParam != VK_SPACE)
                    break;
                                        // fall through
          case WM_LBUTTONUP :
               SendMessage(GetParent(hwnd), WM_COMMAND,
                    GetWindowLong(hwnd, GWL_ID),(LPARAM) hwnd);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

ABOUT3.RC

```
/*---------------------------
   ABOUT3.RC resource script
  ---------------------------*/

#include <windows.h>
#include "about3.h"

about3 ICON about3.ico

About3 MENU
     {
     POPUP "&Help"
          {
          MENUITEM "&About About3...",       IDM_ABOUT
          }
     }

#define TABGRP(WS_TABSTOP | WS_GROUP)
```

```
AboutBox DIALOG  20, 20, 160, 80
     STYLE WS_POPUP | WS_DLGFRAME
     {
     CTEXT   "About3"                    -1,  0, 12, 160,  8
     ICON    "About3"                    -1,  8,  8,   0,  0
     CTEXT   "About Box Demo Program"    -1,  0, 36, 160,  8
     CTEXT   "(c) Charles Petzold, 1996"  -1,  0, 48, 160,  8
     CONTROL "OK" IDOK, "EllipPush",   TABGRP,    64, 60,  32, 14
     }
```

ABOUT3.H

```
/*----------------------
   ABOUT3.H header file
  ----------------------*/

#define IDM_ABOUT      1
```

ABOUT3.ICO

Рис. 11.5 Программа ABOUT3

Класс окна, который мы будем регистрировать называется "EllipPush" эллипсоидная кнопка (elliptical push button). В шаблоне окна диалога вместо инструкции DEFPUSHBUTTON для задания класса окна используется инструкция CONTROL:

```
CONTROL "OK" IDOK, "EllipPush", TABGRP, 64, 60, 32, 14
```

Менеджер окна диалога использует этот класс окна при вызове функции *CreateWindow*, когда в окне диалога создается дочернее окно управления.

В файле ABOUT3.C класс окна "EllipPush" регистрируется в *WinMain*:

```
wndclass.cbSize              = sizeof(wndclass);
wndclass.style               = CS_HREDRAW | CS_VREDRAW;
wndclass.lpfnWndProc         = EllipPushWndProc;
wndclass.cbClsExtra          = 0;
wndclass.cbWndExtra          = 0;
wndclass.hInstance           = hInstance;
wndclass.hIcon               = NULL;
wndclass.hCursor             = LoadCursor(NULL, IDC_ARROW);
wndclass.hbrBackground       =(HBRUSH)(COLOR_BTNFACE +1);
wndclass.lpszMenuName        = NULL;
wndclass.lpszClassName       = "EllipPush";
wndclass.hIconSm             = NULL;

RegisterClassEx(&wndclass);
```

В этом классе окна задается, что оконной процедурой является функция *EllipPushWndProc*, которая также находится в файле ABOUT3.C.

Оконная процедура *EllipPushWndProc* обрабатывает только три сообщения: WM_PAINT, WM_KEYUP и WM_LBUTTONUP. При обработке сообщения WM_PAINT она при помощи функции *GetClientRect* получает размер своего окна, а при помощи функции *GetWindowText* — текст, отображаемый на кнопке. Для рисования эллипса и текста используются функции Windows *Ellipse* и *DrawText*.

Сообщения WM_KEYUP и WM_LBUTTONUP обрабатываются очень просто:

```
case WM_KEYUP:
       if(wParam != VK_SPACE)
              break;        // идем дальше
case WM_LBUTTONUP:
       SendMessage(GetParent(hwnd), WM_COMMAND,  GetWindowLong(hwnd, GWL_ID),(LPARAM)hwnd);
       return 0;
```

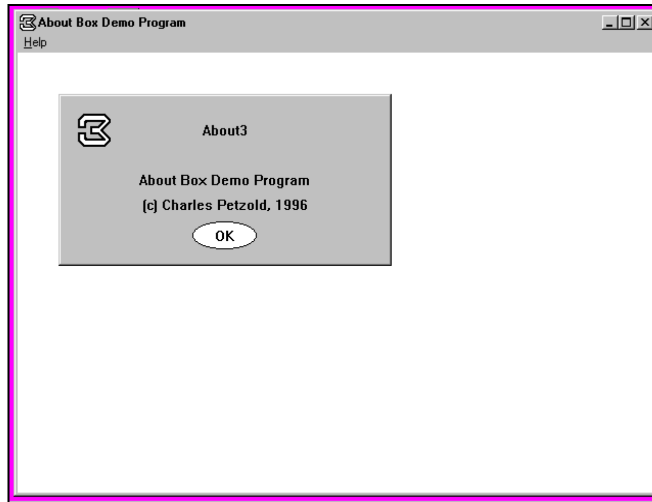


Рис. 11.6 Собственная кнопка, созданная программой ABOUT3

Оконная процедура с помощью функции *GetParent* получает описатель своего родительского окна (окна диалога) и посылает ему сообщение WM_COMMAND с параметром *wParam* равным идентификатору дочернего окна элемента управления, который получен с помощью функции *GetWindowLong*. Затем оконная процедура диалогового окна передает это сообщение процедуре диалога программы ABOUT3. В результате создана своя собственная кнопка, показанная на рис. 11.6. Для создания других пользовательских окон элементов управления в окне диалога можно использовать точно такой же метод.

Действительно ли это все, что нужно? К сожалению, нет. Процедура *EllipPushWndProc* — это лишь основа той логики, которая обычно применяется для поддержки дочернего окна элемента управления. Например, созданная кнопка никак внешне не реагирует на нажатие, как это происходит с нормальной кнопкой. Для инвертирования цвета внутренней области кнопки оконная процедура должна обрабатывать сообщения WM_KEYDOWN (от клавиши <Spacebar>) и WM_LBUTTONDOWN. Кроме этого, в ответ на сообщение WM_LBUTTONDOWN оконной процедуре следует захватывать мышь (capture) и освобождать (release) ее (и возвращать кнопке нормальный цвет), если мышь перемещается вне рабочей области окна кнопки, в то время как кнопка мыши остается нажатой. Дочернее окно должно посылать своему родительскому окну сообщение WM_COMMAND только в том случае, если кнопка мыши отпускается в тот момент, когда мышь еще захвачена.

Кроме этого, *EllipPushWndProc* не обрабатывает сообщения WM_ENABLE. Как уже упоминалось выше, с помощью функции *EnableWindow* процедура диалога может сделать окно недоступным. Текст в дочернем окне можно было бы выводить в сером, а не черном цвете, чтобы показать, что окно элемента управления недоступно и не может получать сообщений.

Если в оконной процедуре дочернего окна элемента управления нужно хранить данные, которые отличаются для каждого создаваемого окна, то задаются положительные значения поля *cbWndExtra* в структуре класса окна. Это резервирует память во внутренней структуре окна, доступ к которой можно получить используя функции *SetWindowWord*, *SetWindowLong*, *GetWindowWord* и *GetWindowLong*.

Окна сообщений

Давайте немного передохнем. Мы рассмотрели способы создания собственных элементов управления в окнах диалога. Теперь остановимся на альтернативе окнам диалога — окнах сообщений (message boxes). Мы использовали диалоговые окна раньше, в главе 7, но подробно они не были рассмотрены.

Окно сообщения является естественной и легко используемой альтернативой окну диалога, когда необходимо получить простой ответ от пользователя. Обычно вызов функции выглядит следующим образом:

```
iItem = MessageBox(hwndParent, szText, szCaption, iType);
```

Окно сообщения имеет заголовок (строку символов *szCaption*), одну или более строк текста (*szText*), одну или более кнопок и (необязательно) предопределенный значок. Одна из кнопок определяется по умолчанию. Возвращаемое значение *iItem* функции *MessageBox* связано с кнопкой, которая была нажата.

Как правило, параметром *hwndParent* является описатель того окна, которое создает окно сообщения. Если окно сообщения закрывается, то фокус ввода оказывается в этом окне. Если описатель окна недоступен, или если вам не нужно, чтобы фокус ввода получило одно из окон вашего приложения, то вместо этого описателя можно использовать NULL. Если окно сообщений используется внутри окна диалога, то в качестве этого параметра используйте описатель окна диалога (который мы назвали *hDlg*).

Параметр *szText* — это дальний указатель на оканчивающуюся нулем строку, которая отображается внутри окна сообщения. При необходимости Windows разбивает этот текст на несколько строк. Можно включить в текст символы табуляции, а используя символ возврата каретки или перевода строки, или оба вместе, можно разбить текст на строки по своему усмотрению. Строка *szCaption* обычно соответствует имени приложения.

Параметр *iType* представляет собой набор флагов, связанных с помощью поразрядного оператора OR языка С. Первая группа флагов задает кнопки, которые появляются в нижней части окна сообщений: MB_OK (по умолчанию), MB_OKCANCEL, MB_YESNO, MB_YESNOCANCEL, MB_RETRYCANCEL, MB_ABORTRETRYIGNORE и MB_HELP. Эти флаги позволяют использовать максимум четыре кнопки. Вторая группа флагов задает то, какая из этих четырех кнопок по умолчанию получит фокус ввода: MB_DEFBUTTON1 (по умолчанию), MB_DEFBUTTON2, MB_DEFBUTTON3 и MB_DEFBUTTON4.

Третья группа флагов задает значок, который появляется в окне сообщений: MB_ICONINFORMATION (что аналогично MB_ICONASTERISK), MB_ICONWARNING (аналогично MB_ICONEXCLAMATION), MB_ICONERROR (аналогично MB_ICONSTOP и MB_ICONHAND) и MB_ICONQUESTION. Значки по умолчанию не задаются. Если не будет установлен один из этих флагов, то в окне сообщений не будет значка. Информационный значок (MB_ICONINFORMATION) следует использовать для сообщения состояния, восклицательный знак (MB_ICONWARNING) — для напоминания, вопросительный знак (MB_ICONQUESTION) — для выяснения намерений пользователя и, наконец, значок ошибки (MB_ICONERROR) — для информирования пользователя о наличии серьезных проблем.

Четвертый набор флагов определяет, является ли окно сообщения модальным окном приложения (в этом случае пользователь может переключиться на другое приложение без закрытия окна сообщения), или это системное модальное окно, которое требует от пользователя, прежде чем что-то сделать, закрыть окно сообщения. Этими флагами являются MB_APPLMODAL (по умолчанию) и MB_SYSTEMMODAL. И наконец, можно использовать пятый флаг MB_NOFOCUS, что приводит к выводу окна сообщений без установки на него фокуса ввода.

В зависимости от нажатой кнопки, окно сообщений возвращает один из следующих идентификаторов: IDOK, IDCANCEL, IDYES, IDNO, IDRETRY, IDIGNORE, IDHELP или IDABORT.

Информация во всплывающих окнах

Одним из полезных применений окна сообщений при работе программы является обеспечение пользователя информацией о ходе ее выполнения. Было бы идеально использовать окно сообщения примерно так же часто, как функция *printf* реализуется в программах на С для MS-DOS, с форматированием строк и переменным числом аргументов. И действительно, можно создать функцию, которая позволяет это делать:

```
void OkMsgBox(char *szCaption, char *szFormat, ...)
{
	char szBuffer [256];
	char *pArguments;

	pArguments =(char *)&szFormat + sizeof(szFormat);
	vsprintf(szBuffer, szFormat, pArguments);
	MessageBox(NULL, szBuffer, szCaption, MB_OK);
}
```

Функция *vsprintf* похожа на функцию *sprintf*, за исключением того, что она использует указатель на набор аргументов (*pArguments*) вместо самих аргументов. Когда вызывается функция *OkMsgBox*, указатель устанавливается на аргументы в стеке. Первым параметром функции *OkMsgBox* является заголовок окна сообщения, вторым — строка формата, а третьим и последующими — выводимые на экран значения. Если нужно, чтобы, скажем, окно сообщений появлялось на экране каждый раз, когда оконная процедура получает сообщение WM_SIZE, то это можно было бы реализовать следующим образом:

```
case WM_SIZE:
	OkMsgBox("WM_SIZE Message",
		"wParam = %04X-%04X, lParam = %04X-%04X",
		HIWORD(wParam), LOWORD(wParam),
		HIWORD(lParam), LOWORD(lParam));
	[другие строки программы]
	return 0;
```

Таким образом внутри окна сообщения выводятся значения *wParam* и *lParam*.

Немодальные окна диалога

В начале этой главы говорилось, что окна диалога могут быть либо модальными (modal), либо немодальными (modeless). К этому моменту модальные окна диалога, встречающиеся наиболее часто из этих двух типов окон, уже

рассмотрены. Модальные окна диалога (исключая системные) позволяют пользователю переключаться между окнами диалога и другими программами. Однако, пользователь не может перейти в другое окно своей программы, пока не закроет модальное окно диалога. Немодальное окно диалога позволяет пользователю переключаться между окном диалога и окном, в котором оно было создано, а также между окном диалога и остальными программами. Таким образом, немодальное окно диалога больше напоминает обычные всплывающие окна, которые могут создаваться программой.

Немодальные окна диалога предпочтительнее в том случае, когда пользователь хотел бы на некоторое время оставить окно диалога на экране. Например, в программах обработки текстов немодальные окна диалога часто используются при поиске (Find) и замене (Replace). Если бы окно диалога Find было бы модальным, то пользователю пришлось бы выбрать в меню опцию Find, ввести искомую строку, закрыть окно, чтобы вернуться в документ, а затем при необходимости повторения поиска, снова воспроизвести весь этот процесс. Гораздо удобнее дать пользователю возможность переключаться между документом и окном диалога.

Как уже известно, модальные окна диалога создаются с помощью функции *DialogBox*. Эта функция возвращает значение только после закрытия окна диалога. Ее возвращаемым значением является второй параметр функции *EndDialog*, которая использовалась в процедуре окна диалога для его закрытия. Немодальные окна диалога создаются с помощью функции *CreateDialog*. Параметры этой функции те же, что и параметры функции *DialogBox*:

```
hDlgModeless = CreateDialog(hInstance, szTemplate, hwndParent, DialogProc);
```

Отличие состоит в том, что функция *CreateDialog* сразу возвращает описатель окна диалога. Как правило, этот описатель хранится в глобальной переменной.

Хотя использование имен функций *DialogBox* для модальных окон диалога и *CreateDialog* для немодальных, может показаться несколько сомнительным, вы можете запомнить, что немодальные окна диалога похожи на обычные окна. А функция *CreateDialog* напоминает функцию *CreateWindow*, с помощью которой создаются обычные окна.

Различия между модальными и немодальными окнами диалога

Работа с немодальными окнами диалога похожа на работу с модальными, но есть несколько важных отличий:

- Немодальные окна диалога обычно содержат строку заголовка и значок системного меню. Инструкция STYLE в шаблоне окна диалога для немодального окна диалога будет выглядеть примерно так:

  ```
  STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU | WS_VISIBLE
  ```

  Строка заголовка и системное меню дают пользователю возможность перемещать немодальное окно диалога по экрану с помощью как мыши, так и клавиатуры. Модальные окна диалога обычно не имеют строки заголовка и системного меню, поскольку пользователь все равно ничего не сможет сделать с окном, в котором появляется модальное диалоговое окно.

- Обратите внимание, что в инструкцию STYLE включен стиль WS_VISIBLE. Если этого не сделать, то после вызова функции *CreateDialog* необходимо вызвать функцию *ShowWindow*:

  ```
  hDlgModeless = CreateDialog(...);
  ShowWindow(hDlgModeless, SW_SHOW);
  ```

  Если не включить в инструкцию STYLE стиль WS_VISIBLE и не вызвать функцию *ShowWindow*, немодальное окно диалога не появится на экране. Забывая об этом, программисты, которые привыкли работать с модальными окнами диалога, часто на первых порах испытывают трудности при создании немодальных окон диалога.

- В отличие от сообщений для модальных окон диалога и окон сообщений, сообщения для немодальных окон диалога проходят через очередь сообщений программы. Поэтому цикл их обработки должен быть изменен таким образом, чтобы эти сообщения передавались в оконную процедуру окна диалога. Вот как это делается: когда для создания немодального окна диалога используется функция *CreateDialog*, необходимо запомнить в глобальной переменной (например, *hDlgModeless*) описатель окна диалога, который является возвращаемым значением этой функции. Необходимо изменить цикл обработки сообщений, чтобы он выглядел так:

  ```
  while(GetMessage(&msg, NULL, 0, 0))
  {
          if(hDlgModeless == 0 || !IsDialogMessage(hDlgModeless, &msg))
          {
                  TranslateMessage(&msg);
                  DispatchMessage(&msg);
          }
  ```

```
}
```

Если сообщение предназначено для немодального окна диалога, то функция *IsDialogMessage* отправляет его оконной процедуре окна диалога и возвращает TRUE (ненулевое значение); в противном случае она возвращает FALSE (0). Функции *TranslateMessage* и *DispatchMessage* будут вызываться только в том случае, если *hDlgModeless* равен 0 или если сообщение не для окна диалога. Если для окна приложения используются быстрые клавиши, то цикл обработки сообщений должен стать таким:

```
while(GetMessage(&msg, NULL, 0, 0))
{
        if(hDlgModeless == 0 || !IsDialogMessage(hDlgModeless, &msg))
        {
                if(!TranslateAccelerator(hwnd, hAccel, &msg))
                {
                        TranslateMessage(&msg);
                        DispatchMessage(&msg);
                }
        }
}
```

Поскольку глобальные переменные инициализируются нулем, *hDlgModeless* останется равной 0 до тех пор, пока окно диалога не будет создано; таким образом гарантируется, что функция *IsDialogMessage* не будет вызвана с недействительным описателем окна. При закрытии немодального окна диалога необходимо, как это будет показано ниже, принять те же меры предосторожности.

Переменная *hDlgModeless* может использоваться и в других частях программы для проверки наличия на экране немодального окна диалога. Например, другие окна программы могут посылать окну диалога сообщения, пока *hDlgModeless* не равна 0.

- Для закрытия немодального окна диалога, вместо функции *EndDialog*, используйте функцию *DestroyWindow*. После вызова функции *DestroyWindow* глобальная переменная *hDlgModeless* должна быть установлена в 0.

По привычке, пользователь закрывает немодальное окно диалога используя опцию системного меню Close. Хотя опция Close разрешена, оконная процедура окна диалога внутри Windows не обрабатывает сообщения WM_CLOSE. Необходимо сделать это в процедуре диалога:

```
case WM_CLOSE:
        DestroyWindow(hDlg);
        hDlgModeless = 0;
        break;
```

Обратите внимание на отличие между параметром *hDlg* функции *DestroyWindow*, передаваемым в процедуру диалога и параметром *hDlgModeless* — глобальной переменной, являющейся возвращаемым значением функции *CreateDialog*, которая проверяется внутри цикла обработки сообщений.

Можно также обеспечить пользователю возможность закрывать немодальное окно диалога с помощью кнопки. При этом используйте ту же логику, что и для сообщения WM_CLOSE. Любая информация, которую окно диалога должно вернуть создавшему его окну, может храниться в глобальных переменных.

Новая программа COLORS

В программе COLORS1, описанной в главе 8, для вывода на экран трех полос прокрутки и шести текстовых элементов было создано девять дочерних окон. В тот момент это была одна из самых сложных наших программ. Изменим программу COLORS1 так, чтобы в ней использовались немодальные окна диалога. Эти изменения делают программу, а в особенности ее функцию *WndProc*, до смешного простой. Переделанная программа COLORS2 представлена на рис. 11.7.

COLORS2.MAK

```
#-----------------------
# COLORS2.MAK make file
#-----------------------

colors2.exe : colors2.obj colors2.res
     $(LINKER) $(GUIFLAGS) -OUT:colors2.exe colors2.obj colors2.res $(GUILIBS)

colors2.obj : colors2.c
     $(CC) $(CFLAGS) colors2.c
```

```
colors2.res : colors2.rc
     $(RC) $(RCVARS) colors2.rc
```

COLORS2.C

```
/*------------------------------------------------
   COLORS2.C -- Version using Modeless Dialog Box
               (c) Charles Petzold, 1996
  ------------------------------------------------*/

#include <windows.h>

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

HWND hDlgModeless;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char  szAppName[] = "Colors2";
     HWND         hwnd;
     MSG          msg;
     WNDCLASSEX   wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground = CreateSolidBrush(0L);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Color Scroll",
                         WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

     hDlgModeless = CreateDialog(hInstance, "ColorScrDlg", hwnd, ColorScrDlg);

     while(GetMessage(&msg, NULL, 0, 0))
          {
          if(hDlgModeless == 0 || !IsDialogMessage(hDlgModeless, &msg))
               {
               TranslateMessage(&msg);
               DispatchMessage (&msg);
               }
          }
     return msg.wParam;
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     switch(iMsg)
```

```
          {
          case WM_DESTROY :
               DeleteObject(
                   (HGDIOBJ) SetClassLong(hwnd, GCL_HBRBACKGROUND,
                        (LONG) GetStockObject(WHITE_BRUSH)));

               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }

BOOL CALLBACK ColorScrDlg(HWND hDlg, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int iColor[3];
     HWND       hwndParent, hCtrl;
     int        iCtrlID, iIndex;

     switch(iMsg)
          {
          case WM_INITDIALOG :
               for(iCtrlID = 10; iCtrlID < 13; iCtrlID++)
                    {
                    hCtrl = GetDlgItem(hDlg, iCtrlID);
                    SetScrollRange(hCtrl, SB_CTL, 0, 255, FALSE);
                    SetScrollPos  (hCtrl, SB_CTL, 0, FALSE);
                    }
               return TRUE;

          case WM_VSCROLL :
               hCtrl   =(HWND) lParam;
               iCtrlID = GetWindowLong(hCtrl, GWL_ID);
               iIndex  = iCtrlID - 10;
               hwndParent = GetParent(hDlg);

               switch(LOWORD(wParam))
                    {
                    case SB_PAGEDOWN :
                         iColor[iIndex] += 15;        // fall through
                    case SB_LINEDOWN :
                         iColor[iIndex] = min(255, iColor[iIndex] + 1);
                         break;
                    case SB_PAGEUP :
                         iColor[iIndex] -= 15;        // fall through
                    case SB_LINEUP :
                         iColor[iIndex] = max(0, iColor[iIndex] - 1);
                         break;
                    case SB_TOP :
                         iColor[iIndex] = 0;
                         break;
                    case SB_BOTTOM :
                         iColor[iIndex] = 255;
                         break;
                    case SB_THUMBPOSITION :
                    case SB_THUMBTRACK :
                         iColor[iIndex] = HIWORD(wParam);
                         break;
                    default :
                         return FALSE;
                    }
               SetScrollPos (hCtrl, SB_CTL,      iColor[iIndex], TRUE);
               SetDlgItemInt(hDlg,  iCtrlID + 3, iColor[iIndex], FALSE);

               DeleteObject(
```

```
                  (HGDIOBJ) SetClassLong(hwndParent, GCL_HBRBACKGROUND,
                       (LONG) CreateSolidBrush(
                             RGB(iColor[0], iColor[1], iColor[2]))));

              InvalidateRect(hwndParent, NULL, TRUE);
              return TRUE;
         }
     return FALSE;
     }
```

COLORS2.RC

```
/*----------------------------
   COLORS2.RC resource script
  ----------------------------*/

#include <windows.h>

#define SBS_VERT_TAB(SBS_VERT | WS_TABSTOP)

ColorScrDlg DIALOG  8, 16, 124, 132
     STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU | WS_VISIBLE
     CAPTION "Color Scroll Scrollbars"
     {
     CONTROL "&Red",   -1, "static",    SS_CENTER,    10,   4, 24,   8
     CONTROL "",       10, "scrollbar", SBS_VERT_TAB, 10,  16, 24, 100
     CONTROL "0",      13, "static",    SS_CENTER,    10, 120, 24,   8
     CONTROL "&Green", -1, "static",    SS_CENTER,    50,   4, 24,   8
     CONTROL "",       11, "scrollbar", SBS_VERT_TAB, 50,  16, 24, 100
     CONTROL "0",      14, "static",    SS_CENTER,    50, 120, 24,   8
     CONTROL "&Blue",  -1, "static",    SS_CENTER,    90,   4, 24,   8
     CONTROL "",       12, "scrollbar", SBS_VERT_TAB, 90,  16, 24, 100
     CONTROL "0",      15, "static",    SS_CENTER,    90, 120, 24,   8
     }
```

Рис. 11.7 Программа COLORS2

Хотя в программе COLORS1 выводимые на экран полосы прокрутки зависели от размеров окна, в новой версии их размер в немодальном окне диалога остается постоянным, как это показано на рис. 11.8. В шаблоне окна диалога в файле COLORS2.RC для всех девяти дочерних окон окна диалога используются инструкции CONTROL. Немодальное окно диалога программы COLORS2 создается в функции *WinMain* сразу после вызова функции *UpdateWindow*, предназначенной для главного окна программы. Обратите внимание, что стиль главного окна включает в себя идентификатор WS_CLIPCHILDREN, что дает программе возможность перерисовать главное окно, не затирая окна диалога.

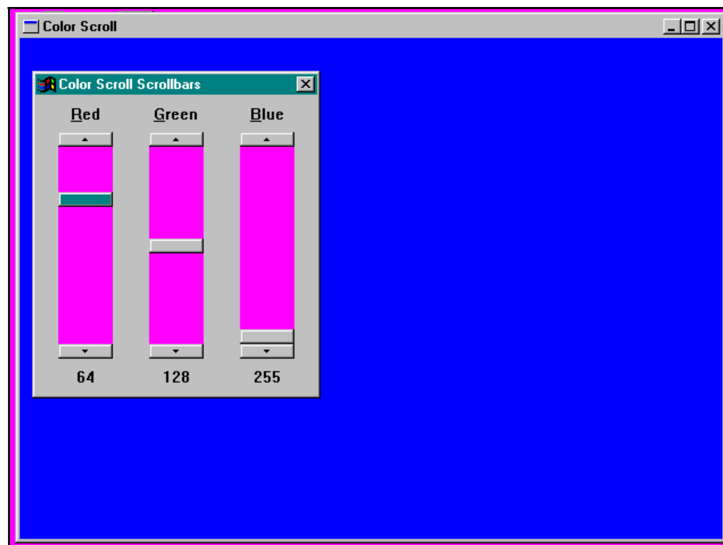


Рис. 11.8 Вид экрана программы COLORS2

Описатель окна диалога, который является возвращаемым значением функции *CreateDialog*, хранится в глобальной переменной *hDlgModeless* и проверяется в цикле обработки сообщений так, как описано выше. Однако, в данной программе нет необходимости хранить описатель в глобальной переменной и проверять его значение перед вызовом функции *IsDialogMessage*. При этом цикл обработки сообщений мог бы быть написан так:

```
while(GetMessage(&msg, NULL, 0, 0))
{
       if(!IsDialogMessage(hDlgModeless, &msg))
       {
              TranslateMessage(&msg);
              DispatchMessage(&msg);
       }
}
```

Поскольку окно диалога создается до того, как в программе вводится цикл обработки сообщений, и окно диалога не закрывается до тех пор пока не завершится программа, значение *hDlgModeless* всегда будет корректным. Ниже приведен код, который может быть вставлен в оконную процедуру диалога, если вам захочется добавить возможность закрытия окна диалога:

```
case WM_CLOSE:
       DestroyWindow(hDlg);
       hDlgModeless = 0;
       break;
```

В программе COLORS1 функция *SetWindowText* отображала в текстовом виде значения трех целых, преобразованных в текст с помощью функции *itoa*. Это выглядело так:

```
SetWindowText(hwndValue[i], itoa(color[i]), szBuffer, 10));
```

Величина *i* соответствовала идентификатору обрабатываемой в данный момент полосы прокрутки, а массив *hwndValue* содержал описатели трех (по числу цветов) статических дочерних окон управления.

В новой версии, чтобы задать выводимое на экран число для каждого текстового поля каждого окна управления используется функция *SetDlgItemInt*:

```
SetDlgItemInt(hDlg, iCtrlID + 3, iColor[iIndex], FALSE);
```

(Хотя функция *SetDlgItemInt* и соответствующая ей функция *GetDlgItemInt* чаще всего используются в окнах редактирования, они также могут применяться для задания текстового поля и в других окнах элементов управления, например статических.) Переменная *iCtrlID* определяет идентификатор полосы прокрутки, а добавление 3 к этому числу превращает его в идентификатор соответствующей числовой метки. Третий параметр задает цвет. Обычно четвертый параметр устанавливается в TRUE, чтобы показать, что числа большие 32767 должны отображаться как отрицательные. Однако, в нашей программы диапазон значений от 0 до 255, поэтому величина четвертого параметра значения не имеет.

В процессе превращения программы COLORS1 в COLORS2 мы передаем Windows все больше и больше задач. В первой версии функция *CreateWindow* вызывалась десять раз; в новой версии по одному разу вызываются функции *CreateWindow* и *CreateDialog*. Но если вам кажется, что количество вызовов функции *CreateWindow* сведено к минимуму, загрузите с диска следующую программу.

Программа HEXCALC: обычное окно или окно диалога?

Возможно, вершиной программистской лени является программа HEXCALC, приведенная на рис. 11.9. В этой программе функция *CreateWindow* вообще не вызывается, нигде не обрабатываются сообщения WM_PAINT, нигде не получают контекста устройства и нигде не обрабатываются сообщения мыши. Несмотря на это, программа работает, представляя собой менее 150 строк исходного текста, и являясь шестнадцатеричным калькулятором, реализующим 10 функций, и имеющим законченный интерфейс клавиатуры и мыши. Окно программы калькулятора показано на рис. 11.10.

HEXCALC.MAK

```
#-----------------------
# HEXCALC.MAK make file
#-----------------------

hexcalc.exe : hexcalc.obj hexcalc.res
     $(LINKER) $(GUIFLAGS) -OUT:hexcalc.exe hexcalc.obj hexcalc.res $(GUILIBS)

hexcalc.obj : hexcalc.c
     $(CC) $(CFLAGS) hexcalc.c
```

```
hexcalc.res : hexcalc.rc hexcalc.ico
     $(RC) $(RCVARS) hexcalc.rc
```

HEXCALC.C

```
/*----------------------------------------
   HEXCALC.C -- Hexadecimal Calculator
                (c) Charles Petzold, 1996
  ----------------------------------------*/

#include <windows.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char  szAppName[] = "HexCalc";
     HWND         hwnd;
     MSG          msg;
     WNDCLASSEX   wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = DLGWINDOWEXTRA;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(hInstance, szAppName);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH)(COLOR_WINDOW + 1);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(hInstance, szAppName);
     RegisterClassEx(&wndclass);

     hwnd = CreateDialog(hInstance, szAppName, 0, NULL);

     ShowWindow(hwnd, iCmdShow);

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

void ShowNumber(HWND hwnd, UINT iNumber)
     {
     char szBuffer[20];

     SetDlgItemText(hwnd, VK_ESCAPE, strupr(ltoa(iNumber, szBuffer, 16)));
     }

DWORD CalcIt(UINT iFirstNum, int iOperation, UINT iNum)
     {
     switch(iOperation)
          {
          case '=' : return iNum;
```

```
          case '+' : return iFirstNum +  iNum;
          case '-' : return iFirstNum -  iNum;
          case '*' : return iFirstNum *  iNum;
          case '&' : return iFirstNum &  iNum;
          case '|' : return iFirstNum |  iNum;
          case '^' : return iFirstNum ^  iNum;
          case '<' : return iFirstNum << iNum;
          case '>' : return iFirstNum >> iNum;
          case '/' : return iNum ? iFirstNum / iNum : UINT_MAX;
          case '%' : return iNum ? iFirstNum % iNum : UINT_MAX;
          default  : return 0;
          }
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static BOOL  bNewNumber = TRUE;
     static int   iOperation = '=';
     static UINT  iNumber, iFirstNum;
     HWND         hButton;

     switch(iMsg)
          {
          case WM_KEYDOWN :                   // left arrow --> backspace
               if(wParam != VK_LEFT)
                    break;
               wParam = VK_BACK;
                                              // fall through
          case WM_CHAR :
               if((wParam = toupper(wParam)) == VK_RETURN)
                    wParam = '=';

               hButton = GetDlgItem(hwnd, wParam);

               if(hButton != NULL)
                    {
                    SendMessage(hButton, BM_SETSTATE, 1, 0);
                    SendMessage(hButton, BM_SETSTATE, 0, 0);
                    }
               else
                    {
                    MessageBeep(0);
                    break;
                    }
                                              // fall through
          case WM_COMMAND :
               SetFocus(hwnd);

               if(LOWORD(wParam) == VK_BACK)         // backspace
                    ShowNumber(hwnd, iNumber /= 16);

               else if(LOWORD(wParam) == VK_ESCAPE)  // escape
                    ShowNumber(hwnd, iNumber = 0);

               else if(isxdigit(LOWORD(wParam)))    // hex digit
                    {
                    if(bNewNumber)
                         {
                         iFirstNum = iNumber;
                         iNumber = 0;
                         }
                    bNewNumber = FALSE;

                    if(iNumber <= UINT_MAX >> 4)
```

```
                         ShowNumber(hwnd, iNumber = 16 * iNumber + wParam -
                             (isdigit(wParam) ? '0' : 'A' - 10));
                    else
                         MessageBeep(0);
                    }
               else                                    // operation
                    {
                    if(!bNewNumber)
                         ShowNumber(hwnd, iNumber =
                              CalcIt(iFirstNum, iOperation, iNumber));
                    bNewNumber = TRUE;
                    iOperation = LOWORD(wParam);
                    }
               return 0;

          case WM_DESTROY :
               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

HEXCALC.RC

```
/*----------------------------
   HEXCALC.RC resource script
  ----------------------------*/

#include <windows.h>

HexCalc ICON hexcalc.ico

HexCalc DIALOG 32768, 0, 102, 122
     STYLE WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX
     CLASS "HexCalc"
     CAPTION "Hex Calculator"
     {
     PUSHBUTTON "D",       68,  8,  24, 14, 14
     PUSHBUTTON "A",       65,  8,  40, 14, 14
     PUSHBUTTON "7",       55,  8,  56, 14, 14
     PUSHBUTTON "4",       52,  8,  72, 14, 14
     PUSHBUTTON "1",       49,  8,  88, 14, 14
     PUSHBUTTON "0",       48,  8, 104, 14, 14
     PUSHBUTTON "0",       27, 26,   4, 50, 14
     PUSHBUTTON "E",       69, 26,  24, 14, 14
     PUSHBUTTON "B",       66, 26,  40, 14, 14
     PUSHBUTTON "8",       56, 26,  56, 14, 14
     PUSHBUTTON "5",       53, 26,  72, 14, 14
     PUSHBUTTON "2",       50, 26,  88, 14, 14
     PUSHBUTTON "Back",     8, 26, 104, 32, 14
     PUSHBUTTON "C",       67, 44,  40, 14, 14
     PUSHBUTTON "F",       70, 44,  24, 14, 14
     PUSHBUTTON "9",       57, 44,  56, 14, 14
     PUSHBUTTON "6",       54, 44,  72, 14, 14
     PUSHBUTTON "3",       51, 44,  88, 14, 14
     PUSHBUTTON "+",       43, 62,  24, 14, 14
     PUSHBUTTON "-",       45, 62,  40, 14, 14
     PUSHBUTTON "*",       42, 62,  56, 14, 14
     PUSHBUTTON "/",       47, 62,  72, 14, 14
     PUSHBUTTON "%",       37, 62,  88, 14, 14
     PUSHBUTTON "Equals",  61, 62, 104, 32, 14
     PUSHBUTTON "&&",      38, 80,  24, 14, 14
     PUSHBUTTON "|",      124, 80,  40, 14, 14
     PUSHBUTTON "^",       94, 80,  56, 14, 14
     PUSHBUTTON "<",       60, 80,  72, 14, 14
```

```
    PUSHBUTTON ">",       62, 80,  88, 14, 14
    }
```

HEXCALC.ICO

Рис. 11.9 Программа HEXCALC

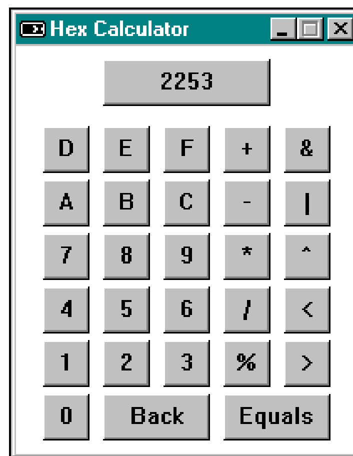


Рис. 11.10 Окно программы HEXCALC

Программа HEXCALC — это обычный калькулятор с нефиксированной записью, использующий обозначение языка С для записи операций. Он работает с беззнаковыми 32-разрядными целыми и может выполнять сложение, вычитание, умножение, деление и определять остаток от деления; осуществлять поразрядные операции AND, OR и исключающее OR; сдвигать числа влево или вправо на один разряд. Деление на 0 приводит к результату FFFFFFFF.

Для работы с калькулятором в программе HEXCALC можно использовать как мышь, так и клавиатуру. Начинать надо с ввода первого числа, щелкнув мышью над ним или набрав его на клавиатуре (до восьми шестнадцатеричных цифр), затем вводится знак операции и второе число. Вывести результат можно либо щелкнув мышью на кнопке Equals, либо нажав клавиши <Equals> или <Enter>. Для исправления ввода используется кнопка Back, либо клавиши <Backspace> или "Стрелка влево". Щелчок в окошке с результатом или нажатие клавиши <Esc> полностью стирает текущий результат.

Необычность программы HEXCALC состоит в том, что выводимое на экран окно кажется гибридом обычного перекрывающегося окна и немодального окна диалога. С одной стороны, все сообщения в программе HEXCALC обрабатываются в функции *WndProc*, которая, как кажется, является обычной оконной процедурой. Функция возвращает длинное целое число и обрабатывает сообщение WM_DESTROY и, как обычная оконная процедура, вызывает *DefWindowProc*. С другой стороны, окно создается в функции *WinMain* с помощью вызова функции *CreateDialog*, использующей шаблон окна диалога из файла HEXCALC.RC. Итак, что же представляет из себя программа HEXCALC, обычное перекрывающееся окно или немодальное окно диалога?

Ответ прост, он состоит в том, что окно диалога — это тоже окно. Обычно Windows использует свою собственную внутреннюю оконную процедуру для обработки сообщений всплывающего окна диалога. Затем Windows передает эти сообщения процедуре окна диалога внутри программы, создавшей окно диалога. В программе HEXCALC мы заставляем Windows для образования всплывающего окна применять шаблон окна диалога, но сообщения для этого окна мы обрабатываем сами.

Внимательное изучение файла HEXCALC.RC позволит обнаружить, как это делается. Начало шаблона окна диалога выглядит так:

```
HexCalc DIALOG 32768, 0, 102, 122
       STYLE WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX
       CLASS "HexCalc"
       CAPTION "Hex Calculator"
```

Обратите внимание на использование идентификаторов, таких как WS_OVERLAPPED и WS_MINIMIZEBOX, которые можно было бы использовать и для создания обычного окна с помощью функции *CreateWindow*. Решающее отличие между нашим окном диалога и всем тем, что мы создавали до сих пор, заключено в

инструкции CLASS. Когда в предыдущих шаблонах окон диалога эта инструкция не использовалась, Windows регистрировала класс окна диалога и использовала собственную оконную процедуру для обработки сообщений окна диалога. Включение в шаблон диалога инструкции CLASS сообщает Windows о необходимости отправлять сообщения куда-то в другое место, а точнее в оконную процедуру, заданную в классе окна "HexCalc".

Класс окна "HexCalc" регистрируется в функции *WinMain* программы HEXCALC точно также, как регистрируется класс обычного окна. Однако, обратите внимание на следующее очень важное отличие: поле *cbWndExtra* структуры WNDCLASS устанавливается в значение DLGWINDOWEXTRA. Это существенно для тех процедур диалога, которые регистрируются в приложении.

После регистрации класса окна, *WinMain* вызывает функцию *CreateDialog*:

```
hwnd = CreateDialog(hInstance, szAppName, 0, NULL);
```

Второй параметр (строка "HexCalc") является именем шаблона окна диалога. Третий параметр, который обычно является описателем родительского окна, устанавливается в 0, поскольку в нашем случае родительское окно отсутствует. Последний параметр, который обячно является адресом процедуры диалога, в нашем случае не требуется, поскольку Windows не обрабатывает сообщений и, следовательно, не может отправить их в процедуру диалога.

Вызов такой функции *CreateDialog* вместе с шаблоном окна диалога преобразуются операционной системой Windows именно в вызов функции *CreateWindow*, которая выглядит следующим образом:

```
hwnd = CreateWindow("HexCalc", "Hex Calculator",
              WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX,
              CW_USEDEFAULT, CW_USEDEFAULT,
              102 * 4 / cxChar, 122 * 8 / cyChar,
              NULL, NULL, hInstance, NULL);
```

Переменные *cxChar* и *cyChar* являются высотой и шириной символа системного шрифта.

Мы извлекли колоссальную выгоду, позволив Windows создать такой вызов функции *CreateWindow*: Windows не ограничится созданием одного всплывающего окна, она также будет вызывать функцию *CreateWindow* для создания всех 29 дочерних окон управления, которыми являются кнопки, определенные в шаблоне окна диалога. Все эти окна управления посылают оконной процедуре родительского окна, которая является ничем иным как функцией *WndProc*, сообщения WM_COMMAND. Это превосходный способ создания такого окна, в котором должно содержаться множество дочерних окон.

Творческое использование идентификаторов дочерних окон элементов управления

В программе HEXCALC нет заголовочного файла с идентификаторами всех дочерних окон управления, определенных в шаблоне окна диалога. Обойтись без этого файла можно, поскольку идентификационным номером каждой из кнопок установлен код ASCII того текста, который появляется на кнопке при ее выводе на экран. Это означает, что *WndProc* может обращаться с сообщениями WM_COMMAND и WM_CHAR почти одинаково. В каждом случае параметр *wParam* является кодом ASCII текста кнопки.

Конечно, небольшая модификация сообщений от клавиатуры все же необходима. *WndProc* обрабатывает сообщения WM_KEYDOWN таким образом, чтобы преобразовать клавишу "стрелка влево" в клавишу <Backspace>. При обработке сообщений WM_CHAR *WndProc* преобразует код символов к верхнему регистру, а клавишу <Enter> к ASCII-коду клавиши <Equals>.

Правильность сообщений WM_CHAR контролируется с помощью вызова функции *GetDlgItem*. Если возвращаемым значением функции является 0, то значит символ клавиатуры не является одним из идентификаторов, определенных в шаблоне окна диалога. Однако, если символ один из этих идентификаторов, то соответствующей кнопке посылается пара сообщений BM_SETSTATE, так, что она в момент нажатия "залипает":

```
hButton =(GetDlgItem(hwnd, wParam);
if(hButton != NULL)
{
       SendMessage(hButton, BM_SETSTATE, 1, 0);
       SendMessage(hButton, BM_SETSTATE, 0, 0);
}
```

Это с минимальными усилиями делает более привлекательным процесс нажатия клавиш в интерфейсе программы HEXCALC.

Когда *WndProc* обрабатывает сообщения WM_COMMAND, она всегда помещает фокус ввода в родительское окно:

```
case WM_COMMAND:
```

```
        SetFocus(hwnd);
```

В противном случае фокус ввода оказался бы на одной из кнопок, независимо от того, был ли на ней щелчок мыши.

Диалоговые окна общего пользования

Одной из первостепенных задач Windows было содействовать созданию стандартизированного интерфейса пользователя. Для многих общепринятых пунктов меню это было сделано достаточно быстро. Почти все производители программного обеспечения для открытия файла использовали последовательность Alt-File-Open. Однако, сами окна диалога для открытия файла часто были совершенно непохожи.

Начиная с версии Windows 3.1, решение этой задачи было найдено и продолжает обеспечиваться под Windows 95. Этим решением стала библиотека диалоговых окон общего пользования (common dialog box library). В этой библиотеке содержатся несколько функций, которые вызывают стандартные окна диалога для открытия и сохранения файлов, поиска и замены, выбора цветов и шрифтов (все это будет показано в этой главе) и для печати (о чем будет рассказано в главе 15).

Для использования этих функций в первую очередь необходимо инициализировать поля структуры и передать функции библиотеки диалоговых окон общего пользования указатель на эту структуру. Функции создают и отображают окно диалога. Когда пользователь закрывает окно диалога, вызванная функция возвращает управление программе, и пользователь получает информацию из структуры, указатель на которую был передан функции.

Функции и структуры, которые необходимы для использования этих окон диалога, определяются в заголовочном файле COMMDLG.H. Файл COMDLG32.LIB представляет собой библиотеку импорта диалоговых окон общего пользования, а файл COMDLG32.DLL — это динамически подключаемая библиотека, в которой содержатся шаблоны и процедуры окон диалога.

Модернизированная программа POPPAD

Когда в главе 10 к программе POPPAD добавили меню, несколько стандартных опций меню не были реализованы. Теперь можно добавить в программу POPPAD логику для обработки команд создания файлов, их считывания и сохранения отредактированных файлов на диске. Кроме этого, в программу POPPAD будет добавлена логика выбора шрифта, а также логика поиска и замены.

Файлы, в которых находится программа POPPAD3, представлены на рис. 11.11.

POPPAD3.MAK

```
#-----------------------
# POPPAD3.MAK make file
#-----------------------

poppad3.exe : poppad.obj   popfile.obj  popfind.obj \
              popfont.obj popprnt0.obj poppad.res
     $(LINKER) $(GUIFLAGS) -OUT:poppad3.exe poppad.obj popfile.obj \
               popfind.obj popfont.obj popprnt0.obj poppad.res $(GUILIBS)

poppad.obj : poppad.c poppad.h
     $(CC) $(CFLAGS) poppad.c
popfile.obj : popfile.c
     $(CC) $(CFLAGS) popfile.c

popfind.obj : popfind.c
     $(CC) $(CFLAGS) popfind.c

popfont.obj : popfont.c
     $(CC) $(CFLAGS) popfont.c

popprnt0.obj : popprnt0.c
     $(CC) $(CFLAGS) popprnt0.c

poppad.res : poppad.rc poppad.h poppad.ico
     $(RC) $(RCVARS) poppad.rc
```

POPPAD.C

```
/*---------------------------------------
```

```
   POPPAD.C -- Popup Editor
              (c) Charles Petzold, 1996
  ---------------------------------------*/

#include <windows.h>
#include <commdlg.h>
#include <stdlib.h>
#include "poppad.h"

#define EDITID   1
#define UNTITLED "(untitled)"

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

          // Functions in POPFILE.C

void PopFileInitialize(HWND);
BOOL PopFileOpenDlg   (HWND, PSTR, PSTR);
BOOL PopFileSaveDlg   (HWND, PSTR, PSTR);
BOOL PopFileRead      (HWND, PSTR);
BOOL PopFileWrite     (HWND, PSTR);

          // Functions in POPFIND.C

HWND PopFindFindDlg    (HWND);
HWND PopFindReplaceDlg (HWND);
BOOL PopFindFindText   (HWND, int *, LPFINDREPLACE);
BOOL PopFindReplaceText(HWND, int *, LPFINDREPLACE);
BOOL PopFindNextText   (HWND, int *);
BOOL PopFindValidFind  (void);

          // Functions in POPFONT.C
void PopFontInitialize  (HWND);
BOOL PopFontChooseFont  (HWND);
void PopFontSetFont     (HWND);
void PopFontDeinitialize(void);

          // Functions in POPPRNT.C

BOOL PopPrntPrintFile(HINSTANCE, HWND, HWND, PSTR);

          // Global variables

static char szAppName[] = "PopPad";
static HWND hDlgModeless;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     MSG         msg;
     HWND        hwnd;
     HACCEL      hAccel;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(hInstance, szAppName);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
```

```
     wndclass.lpszMenuName  = szAppName;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(hInstance, szAppName);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, NULL,
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, szCmdLine);

     ShowWindow(hwnd, iCmdShow);
     UpdateWindow(hwnd);

     hAccel = LoadAccelerators(hInstance, szAppName);

     while(GetMessage(&msg, NULL, 0, 0))
          {
          if(hDlgModeless == NULL || !IsDialogMessage(hDlgModeless, &msg))
               {
               if(!TranslateAccelerator(hwnd, hAccel, &msg))
                    {
                    TranslateMessage(&msg);
                    DispatchMessage(&msg);
                    }
               }
          }
     return msg.wParam;
     }

void DoCaption(HWND hwnd, char *szTitleName)
     {
     char szCaption[64 + _MAX_FNAME + _MAX_EXT];

     wsprintf(szCaption, "%s - %s", szAppName,
               szTitleName[0] ? szTitleName : UNTITLED);

     SetWindowText(hwnd, szCaption);
     }

void OkMessage(HWND hwnd, char *szMessage, char *szTitleName)
     {
     char szBuffer[64 + _MAX_FNAME + _MAX_EXT];

     wsprintf(szBuffer, szMessage, szTitleName[0] ? szTitleName : UNTITLED);

     MessageBox(hwnd, szBuffer, szAppName, MB_OK | MB_ICONEXCLAMATION);
     }

short AskAboutSave(HWND hwnd, char *szTitleName)
     {
     char szBuffer[64 + _MAX_FNAME + _MAX_EXT];
     int  iReturn;

     wsprintf(szBuffer, "Save current changes in %s?",
               szTitleName[0] ? szTitleName : UNTITLED);

     iReturn = MessageBox(hwnd, szBuffer, szAppName,
                          MB_YESNOCANCEL | MB_ICONQUESTION);

     if(iReturn == IDYES)
          if(!SendMessage(hwnd, WM_COMMAND, IDM_SAVE, 0L))
               iReturn = IDCANCEL;
```

```
     return iReturn;
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static BOOL      bNeedSave = FALSE;
     static char      szFileName[_MAX_PATH];
     static char      szTitleName[_MAX_FNAME + _MAX_EXT];
     static HINSTANCE hInst;
     static HWND      hwndEdit;
     static int       iOffset;
     static UINT      iMsgFindReplace;
     int              iSelBeg, iSelEnd, iEnable;
     LPFINDREPLACE    pfr;

     switch(iMsg)
          {
          case WM_CREATE :
               hInst =((LPCREATESTRUCT) lParam) -> hInstance;

                         // Create the edit control child window

               hwndEdit = CreateWindow("edit", NULL,
                         WS_CHILD | WS_VISIBLE | WS_HSCROLL | WS_VSCROLL |
                              WS_BORDER | ES_LEFT | ES_MULTILINE |
                              ES_NOHIDESEL | ES_AUTOHSCROLL | ES_AUTOVSCROLL,
                         0, 0, 0, 0,
                         hwnd,(HMENU) EDITID, hInst, NULL);

               SendMessage(hwndEdit, EM_LIMITTEXT, 32000, 0L);

                         // Initialize common dialog box stuff

               PopFileInitialize(hwnd);
               PopFontInitialize(hwndEdit);

               iMsgFindReplace = RegisterWindowMessage(FINDMSGSTRING);

                         // Process command line

               lstrcpy(szFileName,(PSTR)
                       (((LPCREATESTRUCT) lParam)->lpCreateParams));

               if(strlen(szFileName) > 0)
                    {
                    GetFileTitle(szFileName, szTitleName,
                                 sizeof(szTitleName));

                    if(!PopFileRead(hwndEdit, szFileName))
                         OkMessage(hwnd, "File %s cannot be read!",
                                          szTitleName);
                    }

               DoCaption(hwnd, szTitleName);
               return 0;

          case WM_SETFOCUS :
               SetFocus(hwndEdit);
               return 0;

          case WM_SIZE :
               MoveWindow(hwndEdit, 0, 0, LOWORD(lParam),
                                          HIWORD(lParam), TRUE);
```

```
          return 0;

     case WM_INITMENUPOPUP :
          switch(lParam)
               {
               case 1 :        // Edit menu

                         // Enable Undo if edit control can do it

                    EnableMenuItem((HMENU) wParam, IDM_UNDO,
                         SendMessage(hwndEdit, EM_CANUNDO, 0, 0L) ?
                              MF_ENABLED : MF_GRAYED);

                         // Enable Paste if text is in the clipboard

                    EnableMenuItem((HMENU) wParam, IDM_PASTE,
                         IsClipboardFormatAvailable(CF_TEXT) ?
                              MF_ENABLED : MF_GRAYED);

                         // Enable Cut, Copy, and Del if text is selected

                    SendMessage(hwndEdit, EM_GETSEL,(WPARAM) &iSelBeg,
                                                     (LPARAM) &iSelEnd);

                    iEnable = iSelBeg != iSelEnd ? MF_ENABLED : MF_GRAYED;

                    EnableMenuItem((HMENU) wParam, IDM_CUT,   iEnable);
                    EnableMenuItem((HMENU) wParam, IDM_COPY,  iEnable);
                    EnableMenuItem((HMENU) wParam, IDM_CLEAR, iEnable);
                    break;

               case 2 :        // Search menu

                         // Enable Find, Next, and Replace if modeless
                         //   dialogs are not already active

                    iEnable = hDlgModeless == NULL ?
                                   MF_ENABLED : MF_GRAYED;

                    EnableMenuItem((HMENU) wParam, IDM_FIND,    iEnable);
                    EnableMenuItem((HMENU) wParam, IDM_NEXT,    iEnable);
                    EnableMenuItem((HMENU) wParam, IDM_REPLACE, iEnable);
                    break;
               }
          return 0;

     case WM_COMMAND :
                         // Messages from edit control

          if(lParam && LOWORD(wParam) == EDITID)
               {
               switch(HIWORD(wParam))
                    {
                    case EN_UPDATE :
                         bNeedSave = TRUE;
                         return 0;

                    case EN_ERRSPACE :
                    case EN_MAXTEXT :
                         MessageBox(hwnd, "Edit control out of space.",
                                   szAppName, MB_OK | MB_ICONSTOP);
                         return 0;
                    }
               break;
```

```
          }

     switch(LOWORD(wParam))
          {
                    // Messages from File menu

          case IDM_NEW :
               if(bNeedSave && IDCANCEL ==
                         AskAboutSave(hwnd, szTitleName))
                    return 0;

               SetWindowText(hwndEdit, "\0");
               szFileName[0]  = '\0';
               szTitleName[0] = '\0';
               DoCaption(hwnd, szTitleName);
               bNeedSave = FALSE;
               return 0;

          case IDM_OPEN :
               if(bNeedSave && IDCANCEL ==
                         AskAboutSave(hwnd, szTitleName))
                    return 0;

               if(PopFileOpenDlg(hwnd, szFileName, szTitleName))
                    {
                    if(!PopFileRead(hwndEdit, szFileName))
                         {
                         OkMessage(hwnd, "Could not read file %s!",
                                          szTitleName);
                         szFileName[0]  = '\0';
                         szTitleName[0] = '\0';
                         }
                    }

               DoCaption(hwnd, szTitleName);
               bNeedSave = FALSE;
               return 0;

          case IDM_SAVE :
               if(szFileName[0])
                    {
                    if(PopFileWrite(hwndEdit, szFileName))
                         {
                         bNeedSave = FALSE;
                         return 1;
                         }
                    else
                         OkMessage(hwnd, "Could not write file %s",
                                          szTitleName);
                    return 0;
                    }
                                        // fall through
          case IDM_SAVEAS :
               if(PopFileSaveDlg(hwnd, szFileName, szTitleName))
                    {
                    DoCaption(hwnd, szTitleName);

                    if(PopFileWrite(hwndEdit, szFileName))
                         {
                         bNeedSave = FALSE;
                         return 1;
                         }
                    else
                         OkMessage(hwnd, "Could not write file %s",
```

```
                              szTitleName);
          }
     return 0;

case IDM_PRINT :
     if(!PopPrntPrintFile(hInst, hwnd, hwndEdit,
                          szTitleName))
          OkMessage(hwnd, "Could not print file %s",
                     szTitleName);
     return 0;

case IDM_EXIT :
     SendMessage(hwnd, WM_CLOSE, 0, 0);
     return 0;

          // Messages from Edit menu

case IDM_UNDO :
     SendMessage(hwndEdit, WM_UNDO, 0, 0);
     return 0;

case IDM_CUT :
     SendMessage(hwndEdit, WM_CUT, 0, 0);
     return 0;

case IDM_COPY :
     SendMessage(hwndEdit, WM_COPY, 0, 0);
     return 0;

case IDM_PASTE :
     SendMessage(hwndEdit, WM_PASTE, 0, 0);
     return 0;

case IDM_CLEAR :
     SendMessage(hwndEdit, WM_CLEAR, 0, 0);
     return 0;

case IDM_SELALL :
     SendMessage(hwndEdit, EM_SETSEL, 0, -1);
     return 0;

          // Messages from Search menu

case IDM_FIND :
     SendMessage(hwndEdit, EM_GETSEL, NULL,
                                 (LPARAM) &iOffset);

     hDlgModeless = PopFindFindDlg(hwnd);
     return 0;

case IDM_NEXT :
     SendMessage(hwndEdit, EM_GETSEL, NULL,
                                 (LPARAM) &iOffset);

     if(PopFindValidFind())
          PopFindNextText(hwndEdit, &iOffset);
     else
          hDlgModeless = PopFindFindDlg(hwnd);

     return 0;

case IDM_REPLACE :
     SendMessage(hwndEdit, EM_GETSEL, NULL,
                                 (LPARAM) &iOffset);
```

```
                    hDlgModeless = PopFindReplaceDlg(hwnd);
                    return 0;

               case IDM_FONT :
                    if(PopFontChooseFont(hwnd))
                         PopFontSetFont(hwndEdit);

                    return 0;

                         // Messages from Help menu

               case IDM_HELP :
                    OkMessage(hwnd, "Help not yet implemented!", "\0");
                    return 0;

               case IDM_ABOUT :
                    DialogBox(hInst, "AboutBox", hwnd, AboutDlgProc);
                    return 0;
               }
          break;

     case WM_CLOSE :
          if(!bNeedSave || IDCANCEL != AskAboutSave(hwnd, szTitleName))
               DestroyWindow(hwnd);

          return 0;

     case WM_QUERYENDSESSION :
          if(!bNeedSave || IDCANCEL != AskAboutSave(hwnd, szTitleName))
               return 1;

          return 0;

     case WM_DESTROY :
          PopFontDeinitialize();
          PostQuitMessage(0);
          return 0;

     default:
                    // Process "Find-Replace" iMsgs

          if(iMsg == iMsgFindReplace)
               {
               pfr =(LPFINDREPLACE) lParam;

               if(pfr->Flags & FR_DIALOGTERM)
                    hDlgModeless = NULL;

               if(pfr->Flags & FR_FINDNEXT)
                    if(!PopFindFindText(hwndEdit, &iOffset, pfr))
                         OkMessage(hwnd, "Text not found!", "\0");

               if(pfr->Flags & FR_REPLACE ||
                   pfr->Flags & FR_REPLACEALL)
                    if(!PopFindReplaceText(hwndEdit, &iOffset, pfr))
                         OkMessage(hwnd, "Text not found!", "\0");

               if(pfr->Flags & FR_REPLACEALL)
                    while(PopFindReplaceText(hwndEdit, &iOffset, pfr));

               return 0;
               }
          break;
     }
```

```
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }

BOOL CALLBACK AboutDlgProc(HWND hDlg, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     switch(iMsg)
          {
          case WM_INITDIALOG :
               return TRUE;

          case WM_COMMAND :
               switch(LOWORD(wParam))
                    {
                    case IDOK :
                         EndDialog(hDlg, 0);
                         return TRUE;
                    }
               break;
          }
     return FALSE;
     }
```

POPFILE.C

```
/*------------------------------------------
   POPFILE.C -- Popup Editor File Functions
  ------------------------------------------*/

#include <windows.h>
#include <commdlg.h>
#include <stdlib.h>
#include <stdio.h>

static OPENFILENAME ofn;

void PopFileInitialize(HWND hwnd)
     {
     static char szFilter[] = "Text Files(*.TXT)\0*.txt\0"  \
                              "ASCII Files(*.ASC)\0*.asc\0" \
                              "All Files(*.*)\0*.*\0\0";

     ofn.lStructSize       = sizeof(OPENFILENAME);
     ofn.hwndOwner         = hwnd;
     ofn.hInstance         = NULL;
     ofn.lpstrFilter       = szFilter;
     ofn.lpstrCustomFilter = NULL;
     ofn.nMaxCustFilter    = 0;
     ofn.nFilterIndex      = 0;
     ofn.lpstrFile         = NULL;          // Set in Open and Close functions
     ofn.nMaxFile          = _MAX_PATH;
     ofn.lpstrFileTitle    = NULL;          // Set in Open and Close functions
     ofn.nMaxFileTitle     = _MAX_FNAME + _MAX_EXT;
     ofn.lpstrInitialDir   = NULL;
     ofn.lpstrTitle        = NULL;
     ofn.Flags             = 0;             // Set in Open and Close functions
     ofn.nFileOffset       = 0;
     ofn.nFileExtension    = 0;
     ofn.lpstrDefExt       = "txt";
     ofn.lCustData         = 0L;
     ofn.lpfnHook          = NULL;
     ofn.lpTemplateName    = NULL;
     }

BOOL PopFileOpenDlg(HWND hwnd, PSTR pstrFileName, PSTR pstrTitleName)
     {
```

```
     ofn.hwndOwner         = hwnd;
     ofn.lpstrFile         = pstrFileName;
     ofn.lpstrFileTitle    = pstrTitleName;
     ofn.Flags             = OFN_HIDEREADONLY | OFN_CREATEPROMPT;

     return GetOpenFileName(&ofn);
     }

BOOL PopFileSaveDlg(HWND hwnd, PSTR pstrFileName, PSTR pstrTitleName)
     {
     ofn.hwndOwner         = hwnd;
     ofn.lpstrFile         = pstrFileName;
     ofn.lpstrFileTitle    = pstrTitleName;
     ofn.Flags             = OFN_OVERWRITEPROMPT;

     return GetSaveFileName(&ofn);
     }

static long PopFileLength(FILE *file)
     {
     int iCurrentPos, iFileLength;

     iCurrentPos = ftell(file);
     fseek(file, 0, SEEK_END);

     iFileLength = ftell(file);

     fseek(file, iCurrentPos, SEEK_SET);

     return iFileLength;
     }

BOOL PopFileRead(HWND hwndEdit, PSTR pstrFileName)
     {
     FILE  *file;
     int    iLength;
     PSTR   pstrBuffer;

     if(NULL ==(file = fopen(pstrFileName, "rb")))
          return FALSE;

     iLength = PopFileLength(file);

     if(NULL ==(pstrBuffer =(PSTR) malloc(iLength)))
          {
          fclose(file);
          return FALSE;
          }

     fread(pstrBuffer, 1, iLength, file);
     fclose(file);
     pstrBuffer[iLength] = '\0';

     SetWindowText(hwndEdit, pstrBuffer);
     free(pstrBuffer);

     return TRUE;
     }

BOOL PopFileWrite(HWND hwndEdit, PSTR pstrFileName)
     {
     FILE  *file;
     int    iLength;
     PSTR   pstrBuffer;
```

```
     if(NULL ==(file = fopen(pstrFileName, "wb")))
          return FALSE;

     iLength = GetWindowTextLength(hwndEdit);

     if(NULL ==(pstrBuffer =(PSTR) malloc(iLength + 1)))
          {
          fclose(file);
          return FALSE;
          }

     GetWindowText(hwndEdit, pstrBuffer, iLength + 1);

     if(iLength !=(int) fwrite(pstrBuffer, 1, iLength, file))
          {
          fclose(file);
          free(pstrBuffer);
          return FALSE;
          }

     fclose(file);
     free(pstrBuffer);

     return TRUE;
     }
```

POPFIND.C

```
/*--------------------------------------------------------
   POPFIND.C -- Popup Editor Search and Replace Functions
  --------------------------------------------------------*/

#include <windows.h>
#include <commdlg.h>
#include <string.h>
#define MAX_STRING_LEN   256

static char szFindText [MAX_STRING_LEN];
static char szReplText [MAX_STRING_LEN];

HWND PopFindFindDlg(HWND hwnd)
     {
     static FINDREPLACE fr;       // must be static for modeless dialog!!!

     fr.lStructSize      = sizeof(FINDREPLACE);
     fr.hwndOwner        = hwnd;
     fr.hInstance        = NULL;
     fr.Flags            = FR_HIDEUPDOWN | FR_HIDEMATCHCASE | FR_HIDEWHOLEWORD;
     fr.lpstrFindWhat    = szFindText;
     fr.lpstrReplaceWith = NULL;
     fr.wFindWhatLen     = sizeof(szFindText);
     fr.wReplaceWithLen  = 0;
     fr.lCustData        = 0;
     fr.lpfnHook         = NULL;
     fr.lpTemplateName   = NULL;

     return FindText(&fr);
     }
HWND PopFindReplaceDlg(HWND hwnd)
     {
     static FINDREPLACE fr;       // must be static for modeless dialog!!!

     fr.lStructSize      = sizeof(FINDREPLACE);
     fr.hwndOwner        = hwnd;
```

```
     fr.hInstance        = NULL;
     fr.Flags            = FR_HIDEUPDOWN | FR_HIDEMATCHCASE | FR_HIDEWHOLEWORD;
     fr.lpstrFindWhat    = szFindText;
     fr.lpstrReplaceWith = szReplText;
     fr.wFindWhatLen     = sizeof(szFindText);
     fr.wReplaceWithLen  = sizeof(szReplText);
     fr.lCustData        = 0;
     fr.lpfnHook         = NULL;
     fr.lpTemplateName   = NULL;

     return ReplaceText(&fr);
     }

BOOL PopFindFindText(HWND hwndEdit, int *piSearchOffset, LPFINDREPLACE pfr)
     {
     int   iLength, iPos;
     PSTR  pstrDoc, pstrPos;

               // Read in the edit document

     iLength = GetWindowTextLength(hwndEdit);

     if(NULL ==(pstrDoc =(PSTR) malloc(iLength + 1)))
          return FALSE;

     GetWindowText(hwndEdit, pstrDoc, iLength + 1);

               // Search the document for the find string

     pstrPos = strstr(pstrDoc + *piSearchOffset, pfr->lpstrFindWhat);
     free(pstrDoc);

               // Return an error code if the string cannot be found

     if(pstrPos == NULL)
          return FALSE;

               // Find the position in the document and the new start offset

     iPos = pstrPos - pstrDoc;
     *piSearchOffset = iPos + strlen(pfr->lpstrFindWhat);

               // Select the found text

     SendMessage(hwndEdit, EM_SETSEL, iPos, *piSearchOffset);
     SendMessage(hwndEdit, EM_SCROLLCARET, 0, 0);

     return TRUE;
     }

BOOL PopFindNextText(HWND hwndEdit, int *piSearchOffset)
     {
     FINDREPLACE fr;

     fr.lpstrFindWhat = szFindText;

     return PopFindFindText(hwndEdit, piSearchOffset, &fr);
     }

BOOL PopFindReplaceText(HWND hwndEdit, int *piSearchOffset, LPFINDREPLACE pfr)
     {
               // Find the text

     if(!PopFindFindText(hwndEdit, piSearchOffset, pfr))
```

```
          return FALSE;

               // Replace it

     SendMessage(hwndEdit, EM_REPLACESEL, 0,(LPARAM) pfr->lpstrReplaceWith);

     return TRUE;
     }

BOOL PopFindValidFind(void)
     {
     return *szFindText != '\0';
     }
```

POPFONT.C

```
/*------------------------------------------
   POPFONT.C -- Popup Editor Font Functions
  ------------------------------------------*/

#include <windows.h>
#include <commdlg.h>

static LOGFONT logfont;
static HFONT   hFont;

BOOL PopFontChooseFont(HWND hwnd)
     {
     CHOOSEFONT cf;

     cf.lStructSize      = sizeof(CHOOSEFONT);
     cf.hwndOwner        = hwnd;
     cf.hDC              = NULL;
     cf.lpLogFont        = &logfont;
     cf.iPointSize       = 0;
     cf.Flags            = CF_INITTOLOGFONTSTRUCT | CF_SCREENFONTS
                                                  | CF_EFFECTS;
     cf.rgbColors        = 0L;
     cf.lCustData        = 0L;
     cf.lpfnHook         = NULL;
     cf.lpTemplateName   = NULL;
     cf.hInstance        = NULL;
     cf.lpszStyle        = NULL;
     cf.nFontType        = 0;               // Returned from ChooseFont
     cf.nSizeMin         = 0;
     cf.nSizeMax         = 0;

     return ChooseFont(&cf);
     }

void PopFontInitialize(HWND hwndEdit)
     {
     GetObject(GetStockObject(SYSTEM_FONT), sizeof(LOGFONT),
                                        (PSTR) &logfont);
     hFont = CreateFontIndirect(&logfont);
     SendMessage(hwndEdit, WM_SETFONT,(WPARAM) hFont, 0);
     }

void PopFontSetFont(HWND hwndEdit)
     {
     HFONT hFontNew;
     RECT  rect;

     hFontNew = CreateFontIndirect(&logfont);
     SendMessage(hwndEdit, WM_SETFONT,(WPARAM) hFontNew, 0);
```

```
     DeleteObject(hFont);
     hFont = hFontNew;
     GetClientRect(hwndEdit, &rect);
     InvalidateRect(hwndEdit, &rect, TRUE);
     }

void PopFontDeinitialize(void)
     {
     DeleteObject(hFont);
     }
```

POPPRNT0.C

```
/*---------------------------------------------------------------
   POPPRNT0.C -- Popup Editor Printing Functions(dummy version)
  ---------------------------------------------------------------*/

#include <windows.h>
BOOL PopPrntPrintFile(HINSTANCE hInst, HWND hwnd, HWND hwndEdit,
                                     PSTR pstrTitleName)
     {
     return FALSE;
     }
```

POPPAD.RC

```
/*---------------------------
   POPPAD.RC resource script
  ---------------------------*/

#include <windows.h>
#include "poppad.h"

PopPad ICON "poppad.ico"

PopPad MENU
     {
     POPUP "&File"
          {
          MENUITEM "&New\tCtrl+N",        IDM_NEW
          MENUITEM "&Open...\tCtrl+O",    IDM_OPEN
          MENUITEM "&Save\tCtrl+S",       IDM_SAVE
          MENUITEM "Save &As...",         IDM_SAVEAS
          MENUITEM SEPARATOR
          MENUITEM "&Print...\tCtrl+P",   IDM_PRINT
          MENUITEM SEPARATOR
          MENUITEM "E&xit",               IDM_EXIT
          }
     POPUP "&Edit"
          {
          MENUITEM "&Undo\tCtrl+Z",       IDM_UNDO
          MENUITEM SEPARATOR
          MENUITEM "Cu&t\tCtrl+X",        IDM_CUT
          MENUITEM "&Copy\tCtrl+C",       IDM_COPY
          MENUITEM "&Paste\tCtrl+V",      IDM_PASTE
          MENUITEM "De&lete\tDel",        IDM_CLEAR
          MENUITEM SEPARATOR
          MENUITEM "&Select All",         IDM_SELALL
          }
     POPUP "&Search"
          {
          MENUITEM "&Find...\tCtrl+F",    IDM_FIND
          MENUITEM "Find &Next\tF3",      IDM_NEXT
          MENUITEM "R&eplace...\tCtrl+R", IDM_REPLACE
          }
     POPUP "F&ormat"
```

```
          {
          MENUITEM "&Font...",            IDM_FONT
          }
     POPUP "&Help"
          {
          MENUITEM "&Help",               IDM_HELP
          MENUITEM "&About PopPad...",    IDM_ABOUT
          }
     }

PopPad ACCELERATORS
     {
     "^N",         IDM_NEW
     "^O",         IDM_OPEN
     "^S",         IDM_SAVE
     "^P",         IDM_PRINT
     "^Z",         IDM_UNDO
     VK_BACK,      IDM_UNDO,  VIRTKEY, ALT
     "^X",         IDM_CUT
     VK_DELETE,    IDM_CUT,   VIRTKEY, SHIFT
     "^C",         IDM_COPY
     VK_INSERT,    IDM_COPY,  VIRTKEY, CONTROL
     "^V",         IDM_PASTE
     VK_INSERT,    IDM_PASTE, VIRTKEY, SHIFT
     VK_DELETE,    IDM_CLEAR, VIRTKEY
     "^F",         IDM_FIND
     VK_F3,        IDM_NEXT,  VIRTKEY
     "^R",         IDM_REPLACE
     VK_F1,        IDM_HELP,  VIRTKEY
     }

AboutBox DIALOG  20, 20, 160, 80
     STYLE WS_POPUP | WS_DLGFRAME
     {
     CTEXT "PopPad"                              -1,   0, 12, 160,  8
     ICON  "PopPad"                              -1,   8,  8,   0,  0
     CTEXT "Popup Editor for Microsoft Windows"  -1,   0, 36, 160,  8
     CTEXT "Copyright(c) Charles Petzold, 1996" -1,   0, 48, 160,  8
     DEFPUSHBUTTON "OK"                        IDOK,  64, 60,  32, 14, WS_GROUP
     }

PrintDlgBox DIALOG 20, 20, 100, 76
     STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU | WS_VISIBLE
     CAPTION "PopPad"
     {
     CTEXT "Sending",                  -1,  0, 10, 100,  8
     CTEXT "",                  IDD_FNAME,  0, 20, 100,  8
     CTEXT "to print spooler.",        -1,  0, 30, 100,  8
     DEFPUSHBUTTON  "Cancel",    IDCANCEL, 34, 50,  32, 14, WS_GROUP
     }
```

POPPAD.H

```
/*----------------------
   POPPAD.H header file
  ----------------------*/

#define IDM_NEW          10
#define IDM_OPEN         11
#define IDM_SAVE         12
#define IDM_SAVEAS       13
#define IDM_PRINT        14
#define IDM_EXIT         15

#define IDM_UNDO         20
```

```
#define IDM_CUT         21
#define IDM_COPY        22
#define IDM_PASTE       23
#define IDM_CLEAR       24
#define IDM_SELALL      25

#define IDM_FIND        30
#define IDM_NEXT        31
#define IDM_REPLACE     32

#define IDM_FONT        40

#define IDM_HELP        50
#define IDM_ABOUT       51

#define IDD_FNAME       10
```

POPPAD.ICO

Рис. 11.11 Программа POPPAD3

Чтобы в главе 15 повторно не воспроизводить исходный код программы, в меню была добавлена опция печати и поддержка некоторых других возможностей, что отражено в файле POPPAD.RC.

Все основные исходные программы записаны в файл POPPAD.C. В файле POPFILE.C находится код вызова окон диалога File Open и File Save, а также функции ввода/вывода. Файл POPFIND.C содержит логику поиска и замены, а файл POPFONT.C — логику выбора шрифта. В файле POPPRNT0.C почти ничего нет: в главе 15, при создании окончательной версии программы POPPAD он будет заменен на файл POPPRNT.C.

Рассмотрим сначала файл POPPAD.C. Обратите внимание, что параметр *szCmdLine* функции *WinMain* записывается последним при вызове функции *CreateWindow*. Эта строка могла бы содержать имя файла, если бы, программа POPPAD3 запускалась из командной строки с параметром. В *WndProc* при обработке сообщения WM_CREATE, заданное имя файла передается функции *PopFileRead*, которая находится в файле POPFILE.C.

В файле POPPAD.C поддерживаются две строки для хранения имени файла: в первой с именем *szFileName*, расположенной в *WndProc*, полностью задается диск, путь и имя файла. Во второй с именем *szTitleName* задается только само имя. В программе POPPAD3 она используется в функции *DoCaption* для вывода в заголовке окна имени файла; кроме этого вторая строка используется в функциях *OKMessage* и *AskAboutSave*, чтобы вывести на экран окно сообщения для пользователя.

В файле POPFILE.C находятся несколько функций для вывода на экран окон диалога File Open и File Save, а также фактической реализации ввода/вывода файлов. Окна диалога выводятся на экран с помощью функций *GetOpenFileName* и *GetSaveFileName*, которые находятся в динамически подключаемой библиотеке диалоговых окон общего пользования (COMDLG32.DLL). Обе эти функции используют структуру типа OPENFILENAME, определяемую в заголовочном файле COMMDLG.H. В файле POPFILE.C для этой структуры используется глобальная переменная *ofn*. Большинство полей структуры *ofn* инициализируются в функции *PopFileInitialize*, которая вызывается в файле POPPAD.C при обработке в *WndProc* сообщения WM_CREATE.

Структурную переменную *ofn* удобно сделать статической и глобальной, поскольку функции *GetOpenFileName* и *GetSaveFileName* возвращают в структуру некоторую информацию, которая необходима при последующих вызовах этих функций. Хотя в диалоговых окнах общего пользования имеется множество опций, включая возможность задания собственного шаблона окна диалога и проникновения в процедуру окна диалога, в файле POPPAD.C используются только базовые возможности диалоговых окон File Open и File Save. Задаваемыми полями структуры OPENFILENAME являются: *lStructSize* (размер структуры), *hwndOwner* (владелец окна диалога), *lpstrFilter* (о котором разговор впереди), *lpstrFile* и *nMaxFile* (указатель на буфер, в котором задано полное, с учетом пути, имя файла и размер этого буфера), *lpstrFileTitle* и *nMaxFileTitle* (буфер и его размер только для имени файла), *Flags* (для установки опций окна диалога) и *lpstrDefExt* (здесь задается строка текста с заданным по умолчанию расширением имени файла, если пользователь не задает собственное при наборе имени файла в окне диалога).

Когда пользователь выбирает опцию Open меню File, программа POPPAD3 вызывает функцию *PopFileOpenDlg* из файла POPFILE.C, при этом функции передаются описатель окна, указатель на буфер полного имени файла и указатель на буфер только имени файла. Функция *PopFileOpenDlg* помещает в структуру OPENFILENAME

соответственно поля *hwndOwner*, *lpstrFile* и *lpstrFileTitle*, устанавливает *Flags* в значение OFN_HIDEREADONLY | OFN_CREATEPROMPT, а затем вызывает функцию *GetOpenFileName*, которая выводит на экран хорошо нам знакомое окно диалога, показанное на рис. 11.12. В заданном по умолчанию окне диалога File Open имеется флажок, который дает пользователю возможность назначить, что файл должен быть открыт только для чтения; при установке флага OFN_HIDEREADONLY функция *GetOpenFileName* не выводит этот флажок на экран.

Когда пользователь закрывает показанное на рисунке окно диалога, функция *GetOpenFileName* завершает свою работу. Если заданного пользователем файла не существует, то флаг OFN_CREATEPROMPT заставляет функцию *GetOpenFileName* выводить на экран окно сообщения, в котором пользователю задается вопрос о необходимости создания нового файла.

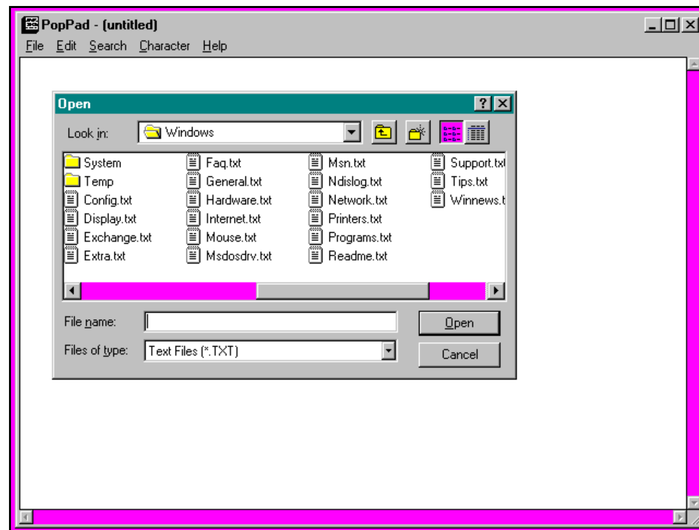


Рис. 11.12 Окно диалога File Open

В поле комбинированного списка, расположенном в левом нижнем углу окна диалога, перечислены типы файлов, которые будут выведены в окне. Это поле называют фильтром (filter). Пользователь может изменить фильтр, выбрав в комбинированном списке другой тип файлов. В функции *PopFileInitialize* из файла POPFILE.C в переменной *szFilter* фильтр определен для трех типов файлов: текстовых с расширением .TXT, файлов ASCII с расширением .ASC и файлов всех типов. Это значение устанавливается в поле *lpstrFilter* структуры OPENFILENAME.

Если пользователь меняет фильтр при активном окне диалога, то в поле *nFilterIndex* структуры OPENFILENAME отражается сделанный пользователем выбор. Поскольку структура хранится как статическая переменная, при следующем вызове окна диалога фильтр будет установлен в соответствие с выбранным типом файла.

Аналогично работает функция *PopFileSaveDlg* из файла POPFILE.C. Она устанавливает параметр *Flags* в OFN_OVERWRITEPROMPT и для вывода на экран окна диалога File Save вызывает функцию *GetSaveFileName*. Если выбранный пользователем файл уже существует, флаг OFN_OVERWRITEPROMPT приводит к выводу на экран окна сообщения, с запросом о необходимости перезаписать существующий файл. Другие функции файла POPFILE.C реализуют ввод/вывод файлов, используя при этом стандартные библиотечные функции языка С. (О вводе/выводе файлов в Windows 95 подробно будет рассказано в главе 13.)

Изменение шрифта

Как обеспечить пользователя возможностью легко выбирать себе шрифты показано в файле POPFONT.C.

При обработке сообщения WM_CREATE в программе POPPAD3 вызывается функция *PopFontInitialize* из файла POPFONT.C. Эта функция получает структуру LOGFONT, сформированную на основе системного шрифта, создает на ее базе шрифт и для установки нового шрифта посылает дочернему окну редактирования сообщение WM_SETFONT. (Хотя заданным по умолчанию шрифтом дочернего окна редактирования является системный шрифт, функция *PopFontInitialize* создает для него новый дополнительный шрифт, поскольку в конечном итоге все равно шрифт будет удален, что для стандартного системного шрифта было бы неразумным.)

Когда программа POPPAD3 при выборе опции шрифта получает сообщение WM_COMMAND, вызывается функция *PopFontChooseFont*. Эта функция инициализирует структуру CHOOSEFONT, а затем для вывода на экран диалогового окна выбора шрифта вызывает функцию *ChooseFont*. Если пользователь нажимает кнопку OK, возвращаемым значением функции *ChooseFont* будет TRUE. Тогда программа POPPAD3 вызывает функцию *PopFontSetFont* для установки в дочернем окне редактирования нового шрифта. Старый шрифт удаляется.

И наконец, при обработке сообщения WM_DESTROY, программа POPPAD3 вызывает функцию *PopFontDeinitialize* для удаления последнего шрифта, созданного функцией *PopFontSetFont*.

Поиск и замена

Библиотека диалоговых окон общего пользования включает в себя также два окна диалога для выполнения функций поиска и замены текста. В обеих этих функциях (*FindText* и *ReplaceText*) используется структура типа FINDREPLACE. В файле POPFIND.C, приведенном на рис. 11.11, для вызова этих функций имеются две других функции (*PopFindFindDlg* и *PopFindReplaceDlg*); кроме этого в нем имеются еще функции для поиска и замены текста в дочернем окне редактирования.

Имеется несколько замечаний, связанных с использованием функций поиска и замены. Во-первых, окна диалога, которые они вызывают, являются немодальными, что означает в случае активных окон диалога, необходимо изменять цикл обработки сообщений, чтобы вызвать функцию *IsDialogMessage*. Во-вторых, структура FINDREPLACE, передаваемая функциям *FindText* и *ReplaceText*, должна быть задана как статическая переменная; и поскольку окна диалога являются немодальными, то функции должны заканчивать свою работу уже после того, как окна диалога выведены на экран, а не после того, как они закрыты. И несмотря на это, необходимо продолжать обеспечивать возможность доступа к структуре из процедуры окна диалога.

В-третьих, до тех пор, пока окна диалога остаются на экране, функции *FindText* и *ReplaceText* взаимодействуют с окном-владельцем посредством специального сообщения. Номер этого сообщения может быть получен с помощью вызова функции *RegisterWindowMessage* с параметром FINDMSGSTRING. Это делается при обработке в *WndProc* сообщения WM_CREATE, и полученный номер сообщения сохраняется в статической переменной.

При обработке очередного сообщения в *WndProc* переменная сообщения сравнивается со значением, возвращаемым функцией *RegisterWindowMessage*. Параметр *lParam* сообщения — это указатель на структуру FINDREPLACE, поле *Flags* которое показывает, использовал ли пользователь окно диалога для поиска и замены текста или оно закрывается. Для непосредственной реализации поиска и замены в программе POPPAD3 вызываются функции *PopFindFindDlg* и *PopFindReplaceDlg*, находящиеся в файле POPFIND.C.

Программа для Windows, содержащая всего один вызов функции

К настоящему времени нами были созданы две программы, в которых имеется возможность просматривать выбираемые цвета: программа COLORS1 в главе 8 и программа COLORS2 в этой главе. Теперь настал черед программы COLORS3, в которой функция Windows вызывается только один раз. Исходный код программы COLORS3 представлен на рис. 11.13.

Единственной функцией Windows, которая вызывается в программе COLORS3, является функция *ChooseColor* — это еще одна функция библиотеки диалоговых окон общего пользования. Окно диалога, которое она выводит на экран, показано на рис. 11.14. Процесс подбора цветов похож на тот, который был в программах COLORS1 и COLORS2, но он несколько более интерактивный.

COLORS3.MAK

```
#-----------------------
# COLORS3.MAK make file
#-----------------------

colors3.exe : colors3.obj
     $(LINKER) $(GUIFLAGS) -OUT:colors3.exe colors3.obj $(GUILIBS)

colors3.obj : colors3.c
     $(CC) $(CFLAGS) colors3.c
```

COLORS3.C

```
/*----------------------------------------------
   COLORS3.C -- Version using Common Dialog Box
                (c) Charles Petzold, 1996
  ----------------------------------------------*/

#include <windows.h>
#include <commdlg.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
```

```
static CHOOSECOLOR cc;
static COLORREF    crCustColors[16];

cc.lStructSize    = sizeof(CHOOSECOLOR);
cc.hwndOwner      = NULL;
cc.hInstance      = NULL;
cc.rgbResult      = RGB(0x80, 0x80, 0x80);
cc.lpCustColors   = crCustColors;
cc.Flags          = CC_RGBINIT | CC_FULLOPEN;
cc.lCustData      = 0L;
cc.lpfnHook       = NULL;
cc.lpTemplateName = NULL;

return ChooseColor(&cc);
}
```

Рис. 11.13 Программа COLORS3

В функции *ChooseColor* используется структура типа CHOOSECOLOR, а для хранения выбранных пользователем в окне диалога цветов — массив из 16 элементов типа COLORREF (DWORD). Поле *rgbResult* может быть инициализировано значением того цвета, который в дальнейшем появится на экране, если в поле *Flags* установлен флаг CC_RGBINIT. При нормальном использовании функции, в поле *rgbResult* устанавливается тот цвет, который выбирает пользователь.

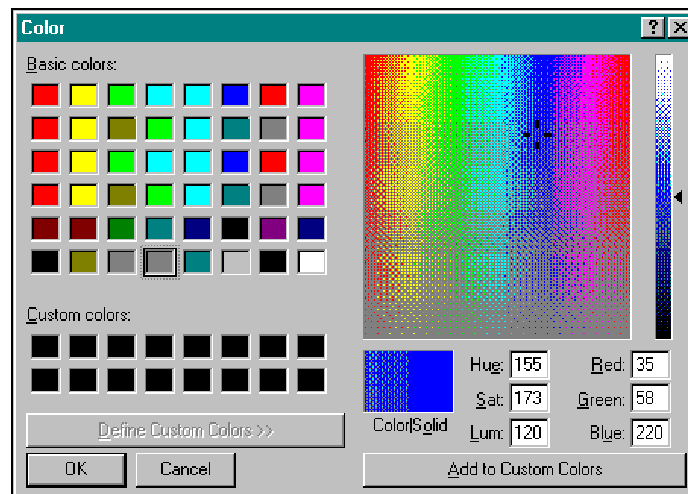


Рис. 11.14 Вид экрана программы COLORS3

Обратите внимание, что поле *hwndOwner* окна диалога Color устанавливается в NULL. Когда функция *ChooseColor* для вывода на экран окна диалога вызывает функцию *DialogBox*, третий параметр функции *DialogBox* также устанавливается в NULL. Такая установка совершенно нормальна. Она означает, что у окна диалога нет родительского окна. В панели задач Windows 95 появится заголовок диалогового окна, и будет казаться, что окно диалога функционирует во многом так, как обычное окно.

Этот прием можно использовать для создания в программе своих собственных окон диалога. Или создать такую программу для Windows, которая только создает диалоговое окно, а всю обработку осуществляет в диалоговой процедуре.

Глава 12 Современный пользовательский интерфейс

12

Каждая вновь выпущенная версия Windows предлагала улучшенный вариант пользовательского интерфейса, и Windows 95 — не исключение. Она предлагает целый набор усовершенствований, облегчающих работу пользователя. Различные формы программы Windows Explorer (включающие Network Neighborhood и Control Panel) упрощают просмотр жестких дисков, сетевых ресурсов, и окон, управляющих системными настройками. Новая оболочка включает старую файловую систему MS-DOS в область имен, содержащей сетевые файлы и серверы печати. Наиболее очевидная часть файловой системы — это рабочий стол (desktop) (указываемый в командной строке как \WINDOWS\DESKTOP), который представляет собой настроенное пользователем окно на доступные программы и файлы. Другой очевидной частью файловой системы является меню Start (расположенное в каталоге \WINDOWS\Start Menu), поддерживающее иерархические меню, как альтернативу рабочему столу в элементизации программ и файлов данных.

Для упрощения создания Windows — программ с интерфейсом пользователя, соответствующим элегантному интерфейсу оболочки системы, Microsoft разработала библиотеку элементов управления общего пользования (common control library). Из семнадцати элементов управления общего пользовани некоторые — такие, как панель инструментов (toolbar) и строка состояния (status bar) — уже много лет используются во многих приложениях, созданных разработчиками, способными самостоятельно создавать программные средства (или использовать набор средств, поддерживаемых библиотеками классов, такими как Microsoft Foundation Class (MFC) Library или OWL фирмы Borland). Другие элементы управления общего пользования, такие как иерархическое дерево просмотра (tree view) и конфигурируемое окно списка (list view), были введены впервые в Windows 95. На рис. 12.1 обобщена информация об элементах управления общего пользования, разделенных на четыре категории: элементы управления главного окна (frame controls), составные диалоговые элементы управления (compound dialog controls), элементы управления Windows Explorer (Windows Explorer controls) и другие элементы управления (miscellaneous controls).

Категория	**Элемент управления**	**Описание**
Элементы управления главного окна		Элементы управления, обычно используемые в главном окне.
	Toolbar (панель инструментов)	Состоит из кнопок быстрого доступа.
	Tooltip (окно подсказки)	Обеспечивает пользователя быстрой подсказкой, отображая текст во всплывающем окне.
	Status bar (строка состояния)	Информационная строка, обычно размещаемая в нижней части окна приложения.
Составные диалоговые элементы управления		Элементы управления для списков свойств и мастеров (wizards).
	Property page (страница свойств)	Немодальное диалоговое окно, используемое как одна страница в списке свойств или мастере.
	Property sheet (набор страниц свойств)	Набор из множества окон страниц свойств.
Элементы управления Windows Explorer		Элементы управления для построения приложений, похожих на Windows Explorer.
	Tree view (дерево просмотра)	Отображает иерархически элементизированный список (левая панель окна программы Windows Explorer).

Категория	Элемент управления	Описание
	List view (список просмотра)	Отображает список элементов, идентифицируемых битовым образом и текстовыми данными (правая панель окна программы Windows Explorer).
Другие элементы управления		
	Animation (анимационное изображение)	Проигрывает анимационную последовательность для индикации длительной операции.
	Drag list (список, поддерживающий операции типа drag/drop)	Окно списка, поддерживающее простые операции drag/drop по отношению к себе и другим окнам типа Drag list. (*Не* drag/drop OLE-контейнер).
	Header (заголовок списка просмотра)	Отображает горизонтальные заголовки для столбцов (используется совместно со списком просмотра).
	Hot-Key (горячая клавиша)	Отображает результат операции определения клавиш активизации (горячих клавиш).
	Image list (список изображений)	Элемент управления для хранения набора растровых изображений (битовых образов, курсоров, значков), не являющийся окном.
	Progress bar (индикатор процесса)	Отображает динамику длительной операции как процент от выполненной задачи.
	Rich edit (усовершенствованный редактор)	Редактор, поддерживающий множество шрифтов и базовые возможности OLE-контейнера.
	Tab (набор закладок для выбора)	Отображает список закладок для выбора. Tabs используются в окне набора страниц свойств для выбора страницы свойств. Панель задач (task bar) Windows 95 — есть элемент управления Tab, использующий кнопки вместо закладок.
	Trackbar (окно с движком для выбора значения из диапазона)	Тип полосы прокрутки для выбора значения в заданном диапазоне.
	Up-Down (полоса прокрутки, связанная с окном редактирования для увеличения или уменьшения на 1 целочисленного значения)	Тип полосы прокрутки, состоящий из двух стрелок (но собственно без полосы) для увеличения или уменьшения на 1 величины, находящейся в связанном поле редактирования.

Рис. 12.1 Элементы управления, поддерживаемые библиотекой COMCTL32.DLL

В этой главе мы рассмотрим основы работы со всеми этими элементами управления и остановимся на первых двух категориях — элементах управления главного окна и составных элементах управления — представляющих собой набор, который будет использоваться в каждом приложении для Windows. Две другие категории — элементы управления Windows Explorer и другие элементы управления — более детально рассматриваются в книге Nancy Cluts "*Programming the Windows 95 User Interface*" (Microsoft Press, 1995).

Основы элементов управления общего пользования

Каждый элемент управления общего пользования, за исключением списка изображений, реализован как класс окна. С этой точки зрения, элементы управления общего пользования похожи на предопределенные элементы управления диалоговых окон, появившиеся в первой версии системы Windows. Оба типа элементов управления строятся с помощью функции *CreateWindow*, настраиваются с использованием конкретных флагов стиля класса, управляются специфичными для данного класса сообщениями и приводятся к нужному состоянию с применением обычных API — вызовов, манипулирующих с окнами. Оба типа элементов управления также посылают уведомляющие сообщения родительскому окну, информируя обо всех происходящих событиях.

Разница между элементами управления общего пользования и предопределенными элементами управления состоит в том, какие сообщения они посылают для уведомления. Предопределенные элементы управления посылают уведомляющие сообщения WM_COMMAND, в то время как элементы управления общего пользования (за некоторыми исключениями) посылают сообщения WM_NOTIFY. Хотя при поверхностном взгляде механизмы передачи отличаются, идея, лежащая в основе этих уведомляющих сообщений, одна и та же: родительское окно может иметь возможность реагировать на все интересующие его события.

Наиболее важно при работе с элементами управления обоих типов помнить, что элемент управления — это окно, и все, что вы уже знаете о манипулировании окнами, применимо при манипулировании элементами управления. Также запомните, что все, что вы уже знаете о работе с предопределенными диалоговыми элементами управления относится и к работе с элементами управления общего пользования. Фактически, только один элемент управления — усовершенствованный редактор — является расширенным окном редактирования, поддерживающим такой же базовый набор стилей управления, сообщений и уведомлений как и оригинал (плюс некоторые новые возможности). Все что вы уже знаете о работе с обычным окном редактирования, поможет вам в дальнейшем при работе с усовершенствованным редактором.

Так же как и при работе с предопределенными элементами управления, преимущества от использования элементов управления общего пользования состоят в том, что, считая каждый из этих элементов "черным ящиком", вы получаете множество возможностей с минимумом затрат с вашей стороны. Ключевым моментом для работы с элементами управления общего пользования является понимание управления и конфигурирование набора конкретных средств таким образом, чтобы при минимальных затратах добиться нужного поведения и внешнего вида элемента управления. Узнав о возможностях, остается только добиться того, чтобы сделать элемент управления работающим на вас.

Инициализация библиотеки

Все элементы управления общего пользования, за исключением усовершенствованного редактора, находятся в файле COMCTL32.DLL, который впервые появился в Microsoft Windows NT 3.51 и Windows 95. (По имени этой DLL можно было бы предположить, что где-то существует 16-разрядная версия этой библиотеки. Microsoft сообщала, что не планирует выпуск 16-битной версии библиотеки элементов управления общего пользования. Microsoft сделала доступными некоторые элементы похожей библиотеки элементов управления общего пользования, поставляемой с Microsoft Windows for Workgroups 3.11, но эта библиотека никогда не поддерживалась официально, и не входит в существующие версии операционных систем Microsoft.)

Для использования какого-либо элемента управления общего пользования программа сначала вызывает функцию *InitCommonControls*, которая регистрирует классы окон элементов управления, используя функцию *RegisterClass*. Функция *InitCommonControls* не имеет параметров и не возвращает никакого значения:

```
InitCommonControls();
```

Описание этой функции вместе с другими описаниями, необходимыми для использования библиотеки элементов управления общего пользования, находится в файле COMMCTRL.H. Этот файл не является частью основной группы файлов, на которые имеет ссылки файл WINDOWS.H. Поэтому, любой исходный файл, ссылающийся на функции элементов управления общего пользования, типы данных, символические константы, должен обязательно содержать следующую строку:

```
#include <commcrtl.h>
```

Для того чтобы помочь компоновщику в поиске функций элементов управления общего пользования, некоторые ссылки должны быть сделаны на статическую библиотеку элементов управления общего пользования COMCTL32.LIB. Ваша среда разработки могла уже включить такую ссылку. Если ее нет, то компоновщик выведет сообщение такого вида:

```
error: unresolved external symbol __imp__InitCommonControls@0
```

Для решения этой проблемы добавьте файл COMCTL32.LIB в список компонуемых библиотек.

Ввиду размера и сложности усовершенствованного редактора, он располагается в его собственной динамически подключаемой библиотеке RICHED32.DLL. (Microsoft не собирается выпускать библиотеку RICHED16.DLL.) Усовершенствованный редактор регистрирует самого себя при загрузке этой библиотеки, которую вы вызываете посредством функции *LoadLibrary*:

```
LoadLibrary( "RICHED32.DLL");
```

Описания усовершенствованного редактора находятся в файле RICHEDIT.H, а связанные с OLE описания для него находятся в файле RICHOLE.H.

Создание элементов управления общего пользования

Наиболее общий путь создания окна элемента управления общего пользования состоит в вызове функции *CreateWindow* или *CreateWindowEx*. (Функция *CreateWindowEx* идентична функции *CreateWindow*, с тем исключением, что она использует дополнительные стили. Эти стили окна рассматриваются в следующем разделе.) Например, приведенный вызов строит панель инструментов:

```
HWND hwndToolBar = CreateWindow(
                            TOOLBARCLASSNAME,
                            NULL,
                            CCS_TOP | WS_CHILD | WS_VISIBLE | WS_BORDER | WS_CLIPSIBLINGS,
                            0, 0, 0, 0,
                            hwndParent,
                            (HMENU) 1,
                            hInst,
                            0
);
```

Рассмотрение некоторых параметров мы пока отложим. Имя класса окна не задается в кавычках, поскольку это символическая константа, определение которой зависит от набора символов, выбранного при построении программы. Для набора символов ANSI символическая константа TOOLBARCLASSNAME заменяется строкой "ToolbarWindow32"; для набора символов UNICODE символ "L" ставится перед этим именем (L"ToolbarWindow32") для создания UNICODE-строки. Все классы элементов управления общего пользования определяются этим способом.

Чаще всего элементы управления общего пользования создаются как дочерние окна, что определяется заданием флага WS_CHILD и установкой описателя родительского окна *hwndParent*. Как показано в примере, дочерние окна часто создаются с начальным местоположением (*x, y*) и размерами (*cx, cy*) равными нулю, а затем изменяют свой размер при изменении размеров родительского окна (т. е. когда родительское окно получает сообщение WM_SIZE).

Альтернативой вызову функции *CreateWindow* является вызов специализированной функции создания элемента управления, которая обычно выполняет некоторую стандартную инициализацию. Примером специализированной функции создания элемента управления является функция *CreateToolbarEx*, строящая панель инструментов, и добавляющая в нее кнопки. В других случаях, таких как набор страниц свойств и страница свойств, имя класса недоступно, поэтому требуется вызов специализированной функции: *PropertySheet* строит набор страниц свойств, а *CreatePropertySheetPage* строит индивидуальные страницы свойств. Список изображений строится вызовом функции *ImageList_Create* — специализированной функции, поскольку список изображений не является окном. На рис. 12.2 приведены все имена классов элементов управления общего пользования и функции их создания.

Категория/Элемент управления	**Класс элемента управления**	**Функция создания**
	Элементы управления главного окна	
Панель инструментов	TOOLBARCLASSNAME	*CreateToolbarEx*
Окно подсказки	TOOLTIPS_CLASS	Нет
Строка состояния	STATUSCLASSNAME	*CreateStatusWindow*
Анимационное изображение	ANIMATE_CLASS	Нет
Индикатор процесса	PROGRESS_CLASS	Нет
	Составные диалоговые элементы управления	
Страница свойств	Нет	*CreatePropertySheetPage*
Набор страниц свойств	Нет	*PropertySheet*
	Элементы управления Windows Explorer	
Дерево просмотра	WC_TREEVIEW	Нет
Список просмотра	WC_LISTVIEW	Нет
Список изображений	Нет	*ImageList_Create*
	Другие элементы управления	
Список, поддерживающий операции типа drag/drop	"listbox" (ANSI) или L"listbox" (UNICODE)	*MakeDragList*
Заголовок списка просмотра	WC_HEADER	Нет
Горячая клавиша	HOTKEY_CLASS	Нет
Усовершенствованный редактор	"RichEdit" (ANSI) или L"RichEdit" (UNICODE)	Нет
Набор закладок для выбора	WC_TABCONTROL	Нет
Окно с движком для выбора значения из диапазона	TRACKBAR_CLASS	Нет
Полоса прокрутки, связанная с окном редактирования для изменения значения	UPDOWN_CLASS	*CreateUpDownControl*

Рис. 12.2 Имена классов и функции создания элементов управления общего пользования

Стили элементов управления общего пользования

Необходимо приложить немало усилий при создании любого типа окна — вашего собственного окна, предопределенного элемента управления или элемента управления общего пользования. Эта работа состоит в выборе правильных флагов стиля окна. (Вспомните, флаги стиля объединяются побитовой операцией OR языка С и передаются вместе как один из двух параметров в функцию *CreateWindowEx* : или как первый параметр

dwExStyle, или как четвертый — *dwStyle*.) Эта работа достаточна трудна, поскольку флаги стиля влияют на широкий диапазон возможностей, включающих визуальное представление окна (или отсутствие отображения, если флаг WS_VISIBLE пропущен по невнимательности), поведение окна и конкретные типы взаимодействия между окнами.

При создании элементов управления общего пользования существует четыре набора флагов стиля: флаги основного стиля окна (WS_), флаги основного стиля элемента управления общего пользования (CCS_), флаги стиля, специфичные для конкретного элемента управления и флаги расширенного стиля (WS_EX_). Флаги первого из перечисленных типов передаются в функцию *CreateWindowEx* в качестве четвертого параметра; флаги последнего типа передаются в функцию *CreateWindowEx* в качестве первого параметра.

Основные стили окна

Основные стили окна имеют имена, начинающиеся с префикса WS_ и могут влиять на окна любого класса. Из примерно двадцати этих стилей окна семь применяются к элементам управления общего пользования: WS_CHILD, WS_VISIBLE, WS_DISABLED, WS_BORDER, WS_TABSTOP, WS_CLIPCHILDREN и WS_CLIPSIBLINGS.

Любое окно элемента управления общего пользования будет использовать бит стиля WS_CHILD, что делает элемент управления дочерним по отношению к какому-то родительскому окну, на поверхности которого будет расположен элемент управления. Когда элемент управления посылает уведомляющие сообщения, он посылает их родительскому окну. Дочерние окна автоматически удаляются при удалении их родительского окна.

Флаг стиля WS_VISIBLE позволяет окну быть отображенным (тем не менее наличие этого флага не гарантирует того, что окно не сможет быть перекрыто другим окном). Частой программной ошибкой является отсутствие флага стиля WS_VISIBLE, что заставляет программиста повсюду (в книгах, журналах, службах поддержки online) искать причину того, что окно не отображается ("потерялось"). Изменяйте, если необходимо, видимость окна после его создания с помощью функций *ShowWindow* или *SetWindowPos*.

Флаг WS_DISABLED делает окно запрещенным, т. е. такое окно не получает сообщений от мыши и клавиатуры. Наиболее общее использование этого флага состоит в запрещении элемента управления в диалоговом окне; будучи запрещенными, большинство элементов управления изменяют свой вид, чтобы дать пользователю понять, что они в данный момент недоступны. Например, кнопка OK в диалоговом окне File Open запрещена до тех пор, пока в поле редактирования имени файла — пусто. Кнопка становится разрешенной (доступной), когда какой-либо текст введен в окно задания имени файла. Окно создается один раз с использованием функции *CreateWindow*, а вызовы функций *EnableWindow* позволяют делать это окно запрещенным или разрешенным.

Флаг WS_BORDER вызывает появление рамки вокруг окна элемента управления.

Если элемент управления находится в диалоговом окне, то использование флага WS_TABSTOP включает его в список переходов по клавише <Tab>.

Флаги WS_CLIPCHILDREN и WS_CLIPSIBLINGS защищают поверхность дочерних окон от внешнего разрушения. Термин "отсечение" (clipping) относится к прорисовке границ между окнами. В то время, как отсечение всегда разрешено для перекрывающихся и всплывающих окон, оно *запрещено* для дочерних окон. Флаг WS_CLIPCHILDREN разрешает отсечение, когда рисуется родительское окно дочерних окон, тем самым предотвращая рисование в родительском окне на дочерних окнах. Флаг WS_CLIPSIBLINGS разрешает отсечение между дочерними окнами, имеющими общее родительское окно, для предотвращения соперничества между ними — борьбы между двумя дочерними окнами, имеющими общего родителя, за пиксели в перекрывающихся участках. При работе с дочерними окнами эти два стиля помогают решить проблемы непонятного визуального представления этих окон.

Флаги расширенного стиля окна

Флаги расширенного стиля окна имеют имена, начинающиеся с префикса WS_EX_, и передаются в функцию *CreateWindowEx* в качестве первого параметра. Из 27 расширенных стилей 3 относятся к созданию дочерних окон: WS_EX_CLIENTEDGE, WS_EX_STATICEDGE и WS_EX_NOPARENTNOTIFY.

Флаги стиля WS_EX_CLIENTEDGE и WS_EX_STATICEDGE поддерживают трехмерность изображения. Надлежащее их использование позволяет приложению хорошо выглядеть по отношению к другим приложениям Windows 95. Обратите внимание, что эти флаги стиля поддерживаются только в Windows 95 и версиях Windows NT, имеющих интерфейс системы Windows 95. (Эти флаги не имеют никакого эффекта в более ранних версиях операционной системы.)

Флаг стиля WS_EX_CLIENTEDGE строит "углубленную" область основной рабочей зоны приложения. Например, окно текста программы текстового процессора. Так как большинство элементов управления, таких как панель инструментов и строка состояния, создаются *вне* этой области, избегайте этих флагов стиля для родительских окон панели инструментов и строки статуса. Кроме того, избегайте использования флага стиля

WS_EX_OVERLAPPEDWINDOW, который включает как часть своего определения флаг стиля WS_EX_CLIENTEDGE.

Флаг стиля WS_EX_STATICEDGE создает углубленное представление только для окон, в которые осуществляется вывод. Например, уведомляющее окно на панели задач Windows 95 — маленькое окно у правого края панели задач, содержащее значки строки состояния и часы — использует этот стиль. Этот флаг используется с такими элементами управления как индикатор процесса и окно анимационного изображения. Кроме того, элементы управления, предназначенные только для вывода, входящие в состав строки состояния или панели инструментов, будут, вероятно, использовать этот флаг стиля, так как его уникальное представление легко понимается опытным пользователем Windows 95.

Флаг стиля WS_EX_NOPARENTNOTIFY отменяет посылку уведомляющих сообщений WM_PARENTNOTIFY дочерним окном родительскому. Без установки этого бита дочернее окно посылает уведомляющие сообщения своему родительскому окну, когда дочернее окно создается, уничтожается или получает сообщение о нажатии клавиш мыши. Элементы управления диалогового окна всегда строятся с использованием этого флага для снижения трафика сообщений.

Флаги основного стиля элемента управления общего пользования

Библиотека элементов управления общего пользования поддерживает набор значений стиля с префиксом CCS_ для использования с панелями инструментов, окнами состояния и заголовками списка просмотра. Это флаги CCS_ADJUSTABLE, CCS_BOTTOM, CCS_NODIVIDER, CCS_NOMOVEY, CCS_NOPARENTALIGN, CCS_NORESIZE и CCS_TOP. Поскольку смысл этих флагов стиля зависит от конкретного элемента управления, при дальнейшем рассмотрении мы будем обращать ваше внимание на детали.

Флаги стиля, специфичные для конкретного элемента управления

Элементы управления общего пользования Windows 95 так же как и пред-определенные элементы управления диалогового окна имеют специфичные флаги стиля, такие как BS_PUSHBUTTON, ES_MULTILINE, LBS_SORT. Так же как и предопределенные элементы управления, каждый стиль элемента управления общего пользования имеет уникальный префикс. Все эти префиксы приведены в таблице на рис. 12.3. Более простые элементы управления общего пользования не имеют специфичных флагов стиля.

Детальное рассмотрение индивидуальных флагов стилей будет приведено позднее в этой главе при подробном описании конкретных элементов управления общего пользования.

Категория/Элемент управления	Префикс флага стиля	Пример
Элементы управления главного окна		
Панель инструментов	TBSTYLE_	TBSTYLE_ALTDRAG
Окно подсказки	Нет	
Строка состояния	SBARS_	SBARS_SIZEGRIP
Анимационное изображение	ACS_	ACS_AUTOPLAY
Индикатор процесса	Нет	
Составные диалоговые элементы управления		
Страница свойств	Нет	
Набор страниц свойств	Нет	
Элементы управления Windows Explorer		
Дерево просмотра	TVS_	TVS_HASBUTTONS
Список просмотра	LVS_	LVS_ALIGNLEFT
Список изображений	Нет	
Другие элементы управления		
Список, поддерживающий операции типа drag/drop	Нет	
Заголовок списка просмотра	HDS_	HDS_BUTTONS
Горячая клавиша	Нет	
Усовершенствованный редактор	ES_	ES_DISABLENOSCROLL
Набор закладок для выбора	TCS_	TCS_BUTTONS
Окно с движком для выбора значения из диапазона	TBS_	TBS_AUTOTICKS
Полоса прокрутки, связанная с окном редактирования для изменения значения	UDS_	UDS_ALIGNLEFT

Рис. 12.3 Префиксы флагов стилей элементов управления общего пользования

Посылка сообщений элементам управления общего пользования

После создания окна элемента управления общего пользования для управления его действиями ему посылаются сообщения. Как можно предположить, для этого требуется вызов функции *SendMessage* с ее традиционными четырьмя параметрами: описатель окна, идентификатор сообщения, значение *wParam*, значение *lParam*. Так же как существуют специфические флаги стилей элементов управления общего пользования, так и существуют специфические сообщения.

Альтернативой вызовам функции *SendMessage* является использование набора макросов языка С, определенных в файле COMMCTRL.H, которые получают специфичный для сообщения набор параметров, осуществляют необходимые преобразования (например, упаковка двух величин типа *shorts* в одну величину типа *lParam*), а затем вызывают функцию *SendMessage*. Возвращаемое значение с целью минимизации числа сообщений компилятора также преобразуется к нужному типу, поскольку значение типа LRESULT, возвращаемое функцией *SendMessage*, не совпадает с ожидаемым типом возвращаемого значения.

В качестве примера того, как удобны эти макросы, посылающие сообщения элементам управления, рассмотрим сообщение TVM_INSERTITEM для добавления элемента в дерево просмотра. Это сообщение добавляет простой элемент в дерево просмотра. Ниже приведено выражение, использующее вызов функции *SendMessage*:

```
hItem =(HTREEITEM)SendMessage( hwndTV, TVM_INSERTITEM, 0,(LPARAM)(LPTV_INSERTSTRUCT)&tvis );
```

Ниже показано, как послать то же самое сообщение, используя макрос *TreeView_InsertItem*:

```
hItem = TreeView_InsertItem(hwndTV, &tvis);
```

Макрос проще для чтения (имя макроса содержит имя класса окна и имя сообщения), он требует только половину параметров, а дает тот же результат, поскольку он расширяется в вызов функции *SendMessage* с соответствующим преобразованием параметров. Перейдя однажды на работу с макросами, обратно вернуться будет уже трудно.

(Если вам интересно использование набора аналогичных макросов для предопределенных элементов управления, то вы можете найти их определения в файле WINDOWSX.H. Макросы не документированы ни в одном файле помощи. Они достаточно хорошо сами себя объясняют и просты в использовании. Так же как и применение макросов для элементов управления общего пользования, так и использование макросов из файла WINDOWSX.H может сделать программу проще для написания и чтения.)

Несмотря на то что такие макросы очень полезны, файлы Win32, к сожалению, включают определения лишь для половины элементов управления общего пользования. Файл COMMCTRL.H содержит лишь макроопределения для следующих классов: анимационное изображение, заголовок списка просмотра, список просмотра, набор страниц свойств (в файле PRSHT.H), дерево просмотра и набор закладок для выбора.

Поскольку эти макросы так удобны, на прилагаемом к книге компакт-диске вы найдете набор макросов для других элементов управления общего пользования. Там находятся макросы для следующих классов: горячая клавиша, индикатор процесса, усовершенствованный редактор, строка состояния, панель инструментов, окно подсказки, окно с движком для выбора значения из диапазона и полоса прокрутки, связанная с окном редактирования для изменения значения. Эти макросы находятся в файле \PETZOLD\CHAP12\COMCTHLP.H.

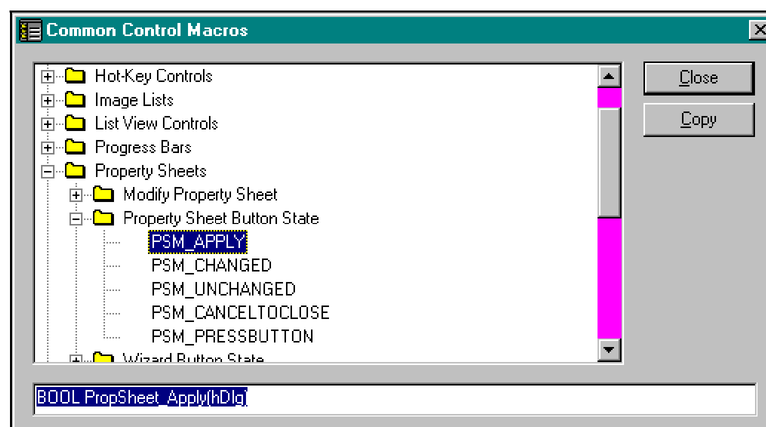


Рис. 12.4 Программа CTLMACRO обеспечивает быстрый доступ к макросам сообщений элементов управления общего пользования

На прилагаемом компакт-диске находится также программа CTLMACRO, которая каталогизирует все макросы элементов управления общего пользования. Как показано на рис. 12.4, программа CTLMACRO элементизирует все макросы элементов управления общего пользования иерархически, и они доступны посредством простого дерева просмотра. Когда вы найдете сообщение, которое вам необходимо, нажмите кнопку Copy для копирования

макроса в папку обмена Clipboard. Оттуда его легко вставить в программу, используя редактор, в котором разрабатывается программа.

Уведомляющие сообщения от элементов управления общего пользования

Как и предопределенные элементы управления, элементы управления общего пользования посылают своему родительскому окну уведомляющие сообщения. Уведомляющие сообщения информируют родительское окно о том, что что-то произошло с окном: пользователь нажал кнопку мыши на элементе управления, напечатал текст, переместил фокус ввода на элемент управления или переместил его с элемента управления.

В отличие от предопределенных элементов управления, которые посылают уведомления как сообщения WM_COMMAND, элементы управления общего пользования посылают уведомления как сообщения WM_NOTIFY. Таким образом, если добавить элемент управления общего пользования к существующему коду, то смешивания обработки уведомляющих сообщений от предопределенных элементов управления и элементов управления общего пользования в программе не произойдет. Сообщения WM_NOTIFY также предотвращают путаницу с уведомляющими сообщениями от меню, которые тоже выражаются в виде сообщений WM_COMMAND.

Однако, не все уведомления элементов управления общего пользования приходят как сообщения WM_NOTIFY. В частности, панель инструментов, использующая сообщения WM_NOTIFY для большинства уведомлений, посылает сообщения WM_COMMAND, когда нажимается кнопка. Поскольку панель инструментов используется для поддержки выбора из меню, тот же код обработки сообщений WM_COMMAND, что для поддержки меню и быстрых клавиш, будет обрабатывать сообщения от панели инструментов. Еще одно исключение — полоса прокрутки, связанная с окном редактирования для изменения значения, которая также посылает сообщения WM_VSCROLL или WM_HSCROLL при нажатии на стрелки.

Хотя каждый элемент управления общего пользования имеет свой собственный набор кодов уведомления, существует общий набор уведомлений. Этот набор приведен в следующей таблице:

Код уведомления	**Описание**
NM_CLICK	Пользователь сделал щелчок левой кнопкой мыши
NM_DBLCLK	Пользователь сделал двойной щелчок левой кнопкой мыши
NM_KILLFOCUS	Элемент управления потерял фокус ввода
NM_OUTOFMEMORY	Ошибка нехватки 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 общего пользования обязательно посылают каждое из этих уведомляющих сообщений. Например, набор закладок для выбора не посылает уведомлений, связанных с изменением фокуса ввода (NM_KILLFOCUS и NM_SETFOCUS). При работе с каждым элементом управления общего пользования следует разобраться с тем, какие уведомляющие сообщения он посылает.

Один из методов разобраться в этом лабиринте уведомляющих сообщений, посылаемых элементами управления общего пользования, состоит в обработке каждого сообщения WM_NOTIFY и выводе соответствующего кода уведомления в окно отладчика. Эта задача упрощается за счет того, что коды уведомлений для каждого элемента управления общего пользования определены в уникальном диапазоне. Поэтому, простая таблица поиска может быть использована для интерпретации кода уведомления. Не имеет значения, с каким элементом управления общего пользования идет работа. Можно обрабатывать сообщение WM_NOTIFY так, как показано ниже, для вывода информации в окно отладчика с помощью функции *OutputDebugString*. Для просмотра результатов работы этой функции в Win32 API вам следует запустить программу в отладчике:

```
case WM_NOTIFY:
	{
		int idCtrl =(int) wParam;
		LPNMHDR pnmh =(LPNMHDR) lParam;

		#ifdef _DEBUG
		// вывод уведомляющих сообщений в окно отладчика
		LPSTR pText;
		if( QueryNotifyText(pnmh->code, &pText) )
		{
			OutputDebugString(pText);
			OutputDebugString("\r\n");
		}
		#endif
```

```
        [другие строки программы]

        return 0;
    }
```

Каждый пример программы в этой главе обрабатывает сообщение WM_NOTIFY так, как показано выше. Описания, которые необходимы для вызова функции *QueryNotifyText*, приведены ниже:

```
typedef struct tagCONTROLNOTIFICATIONS
{
    UINT  nCode;
    LPSTR pName;
} CONTROLNOTIFICATIONS;

BOOL QueryNotifyText(UINT nNotifyCode, LPSTR *pName);
```

Функция и определения данных для *QueryNotifyText* приведены ниже:

```
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include <prsht.h>
#include "notify.h"

CONTROLNOTIFICATIONS cnLookupTable[] =
{
     NM_OUTOFMEMORY,            "NM_OUTOFMEMORY",
     NM_CLICK,                  "NM_CLICK",
     NM_DBLCLK,                 "NM_DBLCLK",
     NM_RETURN,                 "NM_RETURN",
     NM_RCLICK,                 "NM_RCLICK",
     NM_RDBLCLK,                "NM_RDBLCLK",
     NM_SETFOCUS,               "NM_SETFOCUS",
     NM_KILLFOCUS,              "NM_KILLFOCUS",
     LVN_ITEMCHANGING,          "LVN_ITEMCHANGING",
     LVN_ITEMCHANGED,           "LVN_ITEMCHANGED",
     LVN_INSERTITEM,            "LVN_INSERTITEM",
     LVN_DELETEITEM,            "LVN_DELETEITEM",
     LVN_DELETEALLITEMS,        "LVN_DELETEALLITEMS",
     LVN_BEGINLABELEDITA,       "LVN_BEGINLABELEDITA",
     LVN_BEGINLABELEDITW,       "LVN_BEGINLABELEDITW",
     LVN_ENDLABELEDITA,         "LVN_ENDLABELEDITA",
     LVN_ENDLABELEDITW,         "LVN_ENDLABELEDITW",
     LVN_COLUMNCLICK,           "LVN_COLUMNCLICK",
     LVN_BEGINDRAG,             "LVN_BEGINDRAG",
     LVN_BEGINRDRAG,            "LVN_BEGINRDRAG",
     LVN_GETDISPINFOA,          "LVN_GETDISPINFOA",
     LVN_GETDISPINFOW,          "LVN_GETDISPINFOW",
     LVN_SETDISPINFOA,          "LVN_SETDISPINFOA",
     LVN_SETDISPINFOW,          "LVN_SETDISPINFOW",
     LVN_KEYDOWN,               "LVN_KEYDOWN",
     HDN_ITEMCHANGINGA,         "HDN_ITEMCHANGINGA",
     HDN_ITEMCHANGINGW,         "HDN_ITEMCHANGINGW",
     HDN_ITEMCHANGEDA,          "HDN_ITEMCHANGEDA",
     HDN_ITEMCHANGEDW,          "HDN_ITEMCHANGEDW",
     HDN_ITEMCLICKA,            "HDN_ITEMCLICKA",
     HDN_ITEMCLICKW,            "HDN_ITEMCLICKW",
     HDN_ITEMDBLCLICKA,         "HDN_ITEMDBLCLICKA",
     HDN_ITEMDBLCLICKW,         "HDN_ITEMDBLCLICKW",
     HDN_DIVIDERDBLCLICKA,      "HDN_DIVIDERDBLCLICKA",
     HDN_DIVIDERDBLCLICKW,      "HDN_DIVIDERDBLCLICKW",
     HDN_BEGINTRACKA,           "HDN_BEGINTRACKA",
     HDN_BEGINTRACKW,           "HDN_BEGINTRACKW",
     HDN_ENDTRACKA,             "HDN_ENDTRACKA",
     HDN_ENDTRACKW,             "HDN_ENDTRACKW",
     HDN_TRACKA,                "HDN_TRACKA",
     HDN_TRACKW,                "HDN_TRACKW",
```

```
     TVN_SELCHANGINGA,          "TVN_SELCHANGINGA",
     TVN_SELCHANGINGW,          "TVN_SELCHANGINGW",
     TVN_SELCHANGEDA,           "TVN_SELCHANGEDA",
     TVN_SELCHANGEDW,           "TVN_SELCHANGEDW",
     TVN_GETDISPINFOA,          "TVN_GETDISPINFOA",
     TVN_GETDISPINFOW,          "TVN_GETDISPINFOW",
     TVN_SETDISPINFOA,          "TVN_SETDISPINFOA",
     TVN_SETDISPINFOW,          "TVN_SETDISPINFOW",
     TVN_ITEMEXPANDINGA,        "TVN_ITEMEXPANDINGA",
     TVN_ITEMEXPANDINGW,        "TVN_ITEMEXPANDINGW",
     TVN_ITEMEXPANDEDA,         "TVN_ITEMEXPANDEDA",
     TVN_ITEMEXPANDEDW,         "TVN_ITEMEXPANDEDW",
     TVN_BEGINDRAGA,            "TVN_BEGINDRAGA",
     TVN_BEGINDRAGW,            "TVN_BEGINDRAGW",
     TVN_BEGINRDRAGA,           "TVN_BEGINRDRAGA",
     TVN_BEGINRDRAGW,           "TVN_BEGINRDRAGW",
     TVN_DELETEITEMA,           "TVN_DELETEITEMA",
     TVN_DELETEITEMW,           "TVN_DELETEITEMW",
     TVN_BEGINLABELEDITA,       "TVN_BEGINLABELEDITA",
     TVN_BEGINLABELEDITW,       "TVN_BEGINLABELEDITW",
     TVN_ENDLABELEDITA,         "TVN_ENDLABELEDITA",
     TVN_ENDLABELEDITW,         "TVN_ENDLABELEDITW",
     TVN_KEYDOWN,               "TVN_KEYDOWN",
     TTN_NEEDTEXTA,             "TTN_NEEDTEXTA",
     TTN_NEEDTEXTW,             "TTN_NEEDTEXTW",
     TTN_SHOW,                  "TTN_SHOW",
     TTN_POP,                   "TTN_POP",
     TCN_KEYDOWN,               "TCN_KEYDOWN",
     TCN_SELCHANGE,             "TCN_SELCHANGE",
     TCN_SELCHANGING,           "TCN_SELCHANGING",
     TBN_GETBUTTONINFOA,        "TBN_GETBUTTONINFOA",
     TBN_GETBUTTONINFOW,        "TBN_GETBUTTONINFOW",
     TBN_BEGINDRAG,             "TBN_BEGINDRAG",
     TBN_ENDDRAG,               "TBN_ENDDRAG",
     TBN_BEGINADJUST,           "TBN_BEGINADJUST",
     TBN_ENDADJUST,             "TBN_ENDADJUST",
     TBN_RESET,                 "TBN_RESET",
     TBN_QUERYINSERT,           "TBN_QUERYINSERT",
     TBN_QUERYDELETE,           "TBN_QUERYDELETE",
     TBN_TOOLBARCHANGE,         "TBN_TOOLBARCHANGE",
     TBN_CUSTHELP,              "TBN_CUSTHELP",
     UDN_DELTAPOS,              "UDN_DELTAPOS",
     PSN_SETACTIVE,             "PSN_SETACTIVE",
     PSN_KILLACTIVE,            "PSN_KILLACTIVE",
     PSN_APPLY,                 "PSN_APPLY",
     PSN_RESET,                 "PSN_RESET",
     PSN_HELP,                  "PSN_HELP",
     PSN_WIZBACK,               "PSN_WIZBACK",
     PSN_WIZNEXT,               "PSN_WIZNEXT",
     PSN_WIZFINISH,             "PSN_WIZFINISH",
     PSN_QUERYCANCEL,           "PSN_QUERYCANCEL"
};

int NOTIFY_COUNT = sizeof(cnLookupTable) / sizeof(CONTROLNOTIFICATIONS);

//-------------------------------------------------------------------
// QueryNotifyText: Convert notification codes into text
                  (Преобразование уведомляющего кода в текст)
//-------------------------------------------------------------------
BOOL QueryNotifyText(UINT nNotifyCode, LPSTR *pName)
     {
     BOOL bFound = FALSE;
     int  iNotify;
```

```
for(iNotify = 0; iNotify < NOTIFY_COUNT; iNotify++)
     {
     if(cnLookupTable[iNotify].nCode == nNotifyCode)
          {
          *pName = cnLookupTable[iNotify].pName;
          return TRUE;
          }
     }

// Unknown notification code(неизвестный уведомляющий код)
*pName = "** Unknown **";
return FALSE;
}
```

Элементы управления главного окна

Три элемента управления общего пользования часто используются в главных окнах: панели инструментов, окна подсказки и строки состояния. На рис. 12.5 показаны примеры каждого из этих элементов управления, реализованных в программе GADGETS.

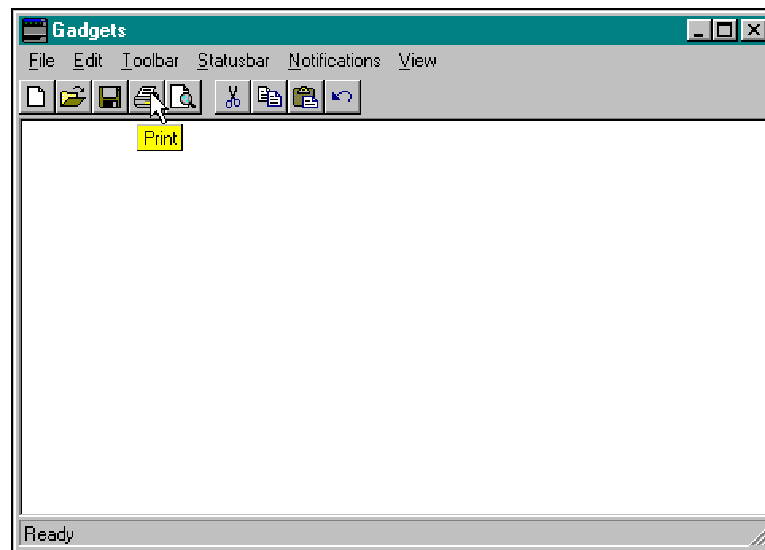


Рис. 12.5 Программа GADGETS, иллюстрирующая использование элементов управления общего пользования: панели инструментов, окна подсказки, строки состояния

Панели инструментов

Панель инструментов — это дочернее окно, обычно расположенное под меню программы, и состоящее из кнопок, соответствующих наиболее употребительным пунктам меню и опциям программы. Кнопки панели инструментов сами по себе не являются окнами, они являются графическими объектами, нарисованными с использованием битовых образов на поверхности окна панели инструментов.

Метками кнопок панели инструментов могут быть либо битовые образы, либо битовые образы с текстовыми метками. (Настоящая реализация не поддерживает кнопки только с текстовыми метками.) Панель инструментов устанавливает размеры всех кнопок одинаковыми, а размеры кнопок с текстовыми метками устанавливают такой величины, чтобы разместить наиболее длинную метку. Поэтому, следует выбирать короткие строки для текстовых меток, чтобы избежать слишком больших кнопок.

Кроме кнопок панель инструментов может содержать другие дочерние окна элементов управления, такие как окно комбинированного списка (combo box). Создаются встроенные элементы управления с помощью вызова функции *CreateWindow* при задании окна панели инструментов как родительского. Как указано в книге Nancy Cuts "*Programming the Windows 95 User Interface*", реальным препятствием для помещения элементов управления в панель инструментов является резервирование достаточно большого пиксельного пространства. В панели инструментов могут также содержаться разделители между кнопками и встроенными элементами управления. Так же как и случае меню, когда разделители определяются в описании меню специальным флагом SEPARATOR, так и в случае панели инструментов разделители кнопок строятся с использованием кнопок панели инструментов стиля

TBSTYLE_SEP. Дюжина или примерно столько разделителей требуется для резервирования места для окна комбинированного списка в панели инструментов.

Создание панели инструментов

Панель инструментов создается либо путем вызова функции *CreateWindow* и задания имени класса TOOLBARCLASSNAME, либо путем вызова функции *CreateToolbarEx*, которая создает панель инструментов и инициализирует набор кнопок. Ниже приведен прототип функции *CreateToolbarEx*:

```
HWND CreateToolbarEx(
       HWND hwnd, DWORD ws, UINT wID,
       int nBitmaps, HINSTANCE hBMInst,
       UINT wBMID, LPCTBBUTTON lpButtons,
       int iNumButtons, int dxButton,
       int dyButton, int dxBitmap,
       int dyBitmap, UINT uStructSize
);
```

Первые три параметра используются в вызове функции *CreateWindow*, который осуществляет функция *CreateToolbarEx*: *hwnd* — это описатель родительского окна, *ws* — флаги стиля окна, *wID* — идентификатор дочернего окна панели инструментов.

Следующие три параметра используются для загрузки ресурса битового образа, который содержит рисунки всех кнопок (множество изображений упакованы в строку в один битовый образ): *nBitmaps* — число изображений в битовом образе, *hBMInst* и *wBMID* идентифицируют ресурс битового образа для загрузки.

Параметр *lpButtons* — это указатель на массив элементов типа TBBUTTON, *iNumButtons* — задает число элементов в массиве. Каждый элемент TBBUTTON определяет битовый образ, идентификатор команды, тип кнопки, ее начальное состояние.

Размер каждой кнопки основывается на базе размеров изображения битового образа (*dxBitmap, dyBitmap*). Минимальная ширина кнопки равна *dxBitmap*+7 пикселей, минимальная высота кнопки — *dyBitmap*+7 пикселей. Существует возможность задать значения *dxBitmap* и *dyBitmap* для установки размеров кнопок больше минимальных. Если необходимости в этом нет, то значения *dxBitmap* и *dyBitmap* устанавливаются равными нулю.

Вариант панели инструментов основывается на размере структуры TBBUTTON, который задается последним параметром *uStructSize*, и должен равняться *sizeof* (TBBUTTON).

При создании панель инструментов устанавливает свой размер и местоположение в "правильные" значения: высота устанавливается в соответствии с высотой кнопок, ширина устанавливается в соответствии с шириной рабочей области родительского окна. Панель инструментов размещается в верхней части родительского окна. Заставить панель инструментов изменить эти установки можно только путем модификации стилей окна. Как это делать — описано в следующем разделе.

Стили окна панели инструментов

Основные свойства панели инструментов управляются с помощью установки флагов стиля окна. Разрешено совместное использование флагов основного стиля элементов управления общего пользования (CCS_) и флагов стиля, специфичных для панели инструментов (TBSTYLE_) (см. таблицу):

Категория	Флаг стиля	Описание
Представление	CCS_NODIVIDER	Запрещает рисование разделительной линии над панелью инструментов
	TBSTYLE_WRAPABLE	Поддерживает панели инструментов, состоящую из нескольких строк
Автоматическое размещение по оси *y*	CCS_TOP	Помещает панель инструментов в верхнюю часть родительского окна (по умолчанию), выравнивая ширину по родительскому окну и высоту по размерам кнопки.
	CCS_BOTTOM	Помещает панель инструментов в нижнюю часть родительского окна, выравнивая ширину по рабочей зоне родительского окна и высоту по размерам кнопки.
	CCS_NOMOVEY	Устанавливает начальное положение по оси *x* (у левой границы родительского окна), но не устанавливает начальное положение по оси *y*, выравнивая ширину по рабочей зоне родительского окна и высоту по размерам кнопки.

Категория	Флаг стиля	Описание
Запрещение автоматического перемещения и автоматического изменения размера	CCS_NOPARENTALIGN	Панель инструментов устанавливает свою высоту, но не положение и ширину. Для нормальной работы посылается сообщение для изменения размера после создания.
	CCS_NORESIZE	Запрещает все автоматические перемещения и изменения размеров.Это запрещает следующие флаги стиля: CCS_TOP, CCS_BOTTOM, CCS_NOMOVEY и CCS_NOPARENTALIGN. Вы должны явно задать размеры и положение панели инструментов.
Изменение конфигурации панели инструментов	CCS_ADJUSTABLE	Поддерживает использование левой кнопки мыши при нажатой клавише <Shift> для перемещения и двойной щелчок для вывода диалогового окна изменения конфигурации. (Более детально об изменении конфигурации панели инструментов читайте соответствующий раздел этой главы.)
	TBSTYLE_ALTDRAG	Изменяет панель инструментов, имеющую стиль CCS_ADJUSTABLE так, что для перемещения кнопок вместо левой кнопки мыши и клавиши <Shift> используется клавиша <Alt> и левая кнопка мыши.
Поддержка окон подсказки	TBSTYLE_TOOLTIPS	Строит элемент управления окно подсказки.

При создании панели инструментов устанавливается по умолчанию только стиль CCS_TOP, что приводит к расположению панели инструментов в верхней части родительского окна. После этого, родительское окно может сделать запрос панели инструментов для изменения ее размеров и местоположения посредством посылки сообщения TB_AUTOSIZE, что обычно и делает родительское окно при обработке сообщения WM_SIZE. За исключением посылки сообщения с запросом об изменении размера панель инструментов с флагом CCS_TOP или CCS_BOTTOM достаточно самостоятельна и не требует другого обслуживания. Присутствие флагов стиля, которые запрещают свойства автоматического перемещения и автоматического изменения размеров панели инструментов, требует немного больших затрат для обслуживания панели инструментов при изменении размеров родительского окна.

Два флага стиля изменяют представление панели инструментов. Первый, CCS_NODIVIDER, удаляет разделительную линию, призванную отделить кнопки панели инструментов стиля CCS_TOP от меню приложения. Для панелей инструментов, отображаемых в других местах (таких как вторая панель инструментов, выводимая под первой), вероятно, потребуется спрятать разделительную линию. Это справедливо и для окон, имеющих панель инструментов, но не имеющих меню — эта разделительная линия будет плохо смотреться. Другой флаг стиля, относящийся к изменению представления панели инструментов, TBSTYLE_WRAPABLE позволяет переносить на другую строку кнопки панели инструментов. Без этого флага, кнопки, которые слишком длинные чтобы поместиться в одну линию, делаются невидимыми и становятся недоступными для пользователя.

Три флага стиля управляют перемещением относительно оси *y*: CCS_TOP (по умолчанию), CCS_BOTTOM и CCS_NOMOVEY. Эти флаги определяют, как будет расположена панель инструментов в ее родительском окне как при создании, так и при получении сообщения TB_AUTOSIZE. Флаг CCS_NOMOVEY делает возможным только приведение ширины (для полного использования рабочей области родительского окна), высоты и месторасположения относительно оси *x*, и оставляет на ваше усмотрение перемещение панели инструментов вдоль оси *y*, как при указании значений по оси *y* в вызове функции *CreateWindow*, так и при вызове функций типа *MoveWindow* после того, как панель инструментов будет создана. Панель инструментов с этим флагом была бы удобна в качестве второй панели инструментов, расположенной непосредственно под панелью инструментов с флагом CCS_TOP.

Два других флага стиля ограничивают возможности автоматического перемещения и автоматического изменения размеров панели инструментов. При использовании флага CCS_NOPARENTALIGN высота панели инструментов устанавливается в соответствии с высотой кнопок, но необходимо установить местоположение и ширину. Флаг CCS_NORESIZE полностью запрещает любое перемещение и изменение размеров панели инструментов. Этот флаг был бы полезным при использовании панели инструментов как элемента управления в диалоговом окне, когда она должна занимать указанную область без самостоятельного изменения размеров. Этот флаг был бы также полезен при установке двух и более панелей инструментов друг за другом в одной строке.

Флаг стиля CCS_ADJUSTABLE строит панель инструментов, которую пользователь может изменять "на лету". Кнопки могут быть перемещены в рамках панели инструментов или перетащены за панель инструментов с помощью клавиши <Shift> и левой кнопки мыши. Двойной щелчок левой кнопкой мыши приводит к вызову диалогового окна изменения

конфигурации панели инструментов (Customize Toolbar dialog box) для добавления, удаления или перемещения кнопок. (Это диалоговое окно появляется и при получении панелью инструментов сообщения TB_CUSTOMIZE.) Изменение конфигурации требует от родительского окна реагирования на некоторые уведомляющие сообщения — TBN_QUERYINSERT и TBN_QUERYDELETE, посылаемые среди других панелью инструментов, которые запрашивают разрешение на вставку или удаление кнопок. Для панели инструментов с изменяемой конфигурацией флаг стиля TBSTYLE_ALTDRAG изменяет пользовательский интерфейс перемещения кнопок с сочетания клавиши <Shift> и левой кнопки мыши на сочетание клавиши <Alt> и левой кнопки мыши для тех случаев, когда первое сочетание используется в других целях.

Флаг стиля TBSTYLE_TOOLTIPS требует от панели инструментов создания элемента управления подсказка, который выводит маленькое окно с текстом для каждой кнопки. Родительское окно панели инструментов получает уведомление TTN_NEEDTEXT (в форме сообщения WM_NOTIFY), когда окну подсказки необходим текст для конкретной кнопки.

Задание изображений на поверхности кнопок

Оконная процедура панели инструментов преобразует простой битовый образ в множество изображений, необходимых для вывода на поверхность кнопок. При этом все изображения будут иметь одинаковый размер. Битовые образы с более чем одним изображением должны помещать изображения в одну строку так, чтобы второе изображение находилось справа от первого, третье справа от второго и т. д. Функция *CreateToolbarEx* принимает только один идентификатор битового образа. Вы можете добавлять битовые образы с одним или более изображением кнопок путем посылки сообщения TB_ADDBITMAP панели инструментов.

Библиотека элементов управления общего пользования имеет два набора битовых образов, готовых для использования. Первый набор битовых образов содержит изображения кнопок, соответствующие командам меню File и Edit; второй набор содержит изображения кнопок для различных типов просмотра. Каждый набор содержит два одинаковых ряда битовых образов двух размеров: большого (24x24) и маленького (16x16) пикселей. Для доступа к этим битовым образам необходимо задать специальное значение HINST_COMMCTRL в качестве описателя экземпляра ресурса битового образа (*hBMInst*) — параметра функции *CreateToolbarEx* (или в соответствующей структуре при посылке сообщения TB_ADDBITMAP). Значение параметра идентификатора битового образа *wBMID* выбирается из следующей таблицы:

Идентификатор ресурса битового образа	Описание
IDB_STD_SMALL_COLOR	16x16 изображения кнопок меню File и Edit
IDB_STD_LARGE_COLOR	24x24 изображения кнопок меню File и Edit
IDB_VIEW_SMALL_COLOR	16x16 изображения кнопок меню View
IDB_VIEW_LARGE_COLOR	24x24 изображения кнопок меню View

Два "стандартных" битовых образа (IDB_STD_SMALL_COLOR и IDB_STD_LARGE_COLOR) содержат по 15 изображений. Связывание конкретного изображения с кнопкой на панели инструментов требует задания индекса изображения (относительно 0) в элементе *iBitmap* соответствующей кнопке структуры типа TBBUTTON. При использовании стандартных битовых образов выбирайте индексы из набора символических констант: STD_CUT, STD_COPY, STD_PASTE, STD_UNDO, STD_REDOW, STD_DELETE, STD_FILENEW, STD_FILEOPEN, STD_FILESAVE, STD_PRINTPRE, STD_PROPERTIES, STD_HELP, STD_FIND, STD_REPLACE, STD_PRINT.

Два битовых образа "вида" (view) (IDB_VIEW_SMALL_COLOR и IDB_VIEW_LARGE_COLOR) содержат по 12 изображений. Выбирайте индексы изображений кнопок, используя следующие символические константы: VIEW_LARGEICONS, VIEW_SMALLICONS, VIEW_LIST, VIEW_DETAILS, VIEW_SORTNAME, VIEW_SORTSIZE, VIEW_SORTDATE, VIEW_SORTTYPE, VIEW_PARENTFOLDER, VIEW_NETCONNECT, VIEW_NETDISCONNECT, VIEW_NEWFOLDER.

При использовании одного из этих четырех битовых образов нет необходимости устанавливать размеры (*dxButton, dyButton*) или размеры битового образа (*dxBitmap, dyBitmap*) поскольку функция *CreateToolbarEx* распознает эти битовые образы и устанавливает размеры сама.

Заполнение массива TBBUTTON

Создание панели инструментов требует определения конкретных параметров кнопок. Вам необходимо заполнить данными элементы массива TBBUTTON, которые определяют конкретные кнопки. После того, как панель инструментов создана, можно определить дополнительные кнопки путем посылки сообщения TB_ADDBUTTONS панели инструментов или вставить кнопки между существующими кнопками путем посылки сообщения TB_INSERTBUTTON.

TBBUTTON определяется в файле COMMCTRL.H следующим образом:

```
typedef struct _TBBUTTON
{
```

```
        int iBitmap;
        int idCommand;
        BYTE fsState;
        BYTE fsStyle;
        BYTE bReserved[2];
        DWORD dwData;
        int iString;
} TBBUTTON;
```

Обратите внимание, что это определение отличается от аналогичного, приведенного в файлах подсказки Win32, и не содержащего поля *bReserved*. При инициализации элементов структуры TBBUTTON следует определить два поля, связанные со всем массивом.

Поле *iBitmap* структуры TBBUTTON — это индекс (относительно нуля) изображения кнопки. При использовании битовых образов из библиотеки элементов управления общего пользования необходимо использовать константы STD_ и VIEW_ как значения в этом поле. В данной панели инструментов определяются собственные индексы изображений, в зависимости от порядка их включения в панель инструментов.

Поле *idCommand* структуры TBBUTTON — идентификатор команды, соответствующей кнопке. При нажатии кнопки она посылает сообщение WM_COMMAND с параметром *wParam* равным *idCommand*.

Поля *fsState* и *fsStyle* определяют начальное состояние и стиль кнопки. Последний не изменяется в течение всей жизни кнопки. Состояние кнопки устанавливается при ее создании и может быть изменено действиями пользователя или путем посылки сообщений панели инструментов. Ниже приведены пять стилей кнопок:

Стиль кнопки	**Описание**
TBSTYLE_BUTTON	Кнопка ведет себя как стандартная кнопка (pushbutton). Кнопка может быть нажата, но не может оставаться в нажатом состоянии.
TBSTYLE_SEP	Разделитель для создания пространства между кнопками или для резервирования места для дочерних элементов управления (таких как комбинированный список).
TBSTYLE_CHECK	Кнопка ведет себя как флажок (check box). Каждый щелчок мыши изменяет состояние кнопки (нажата/отжата).
TBSTYLE_GROUP	Кнопка является членом группы кнопок типа переключателей (radio buttons). Кнопка остается нажатой до тех пор, пока не будет нажата другая кнопка из этой группы.
TBSTYLE_CHECKGROUP	Объединяет свойства стилей TBSTYLE_CHECK и TBSTYLE_GROUP.

Пять стилей комбинируются для получения приблизительного эквивалента трех базовых типов кнопок диалоговых окон: кнопок, флажков, переключателей.

Ниже приведены шесть состояний кнопок:

Состояния кнопок	**Описание**
TBSTATE_CHECKED	Кнопка стиля флажок находится в нажатом состоянии
TBSTATE_PRESSED	Кнопка любого стиля находится в нажатом состоянии
TBSTATE_ENABLED	Кнопка доступна (может реагировать на действия мышью)
TBSTATE_HIDDEN	Скрытая кнопка. (Такая кнопка не отображается и ее место занимают другие кнопки.)
TBSTATE_INDETERMINATE	Кнопка находится в неопределенном состоянии, отображается серым цветом, может быть нажата.
TBSTATE_WRAP	Кнопки панели инструментов стиля TBSTYLE_WRAPABLE, находящиеся после кнопки, имеющей это состояние, отображаются в новой строке.

Каждое из этих состояний доступно для чтения и установки с помощью пары сообщений, специфичных для панели инструментов. Например, можно запросить, является ли конкретная кнопка доступной, путем посылки сообщения TB_ISBUTTONENABLED; посылая сообщение TB_ENABLEBUTTON можно сделать кнопку доступной или недоступной.

Поле *dwData* структуры TBBUTTON — это необязательное 4-х байтовое поле, предназначенное для использования. Например, вы можете сохранить в этом поле указатель на данные, специфические для этой кнопки. Возможно только однократная установка значения этого поля при создании кнопки. Вы можете получить значение из поля *dwData* путем посылки сообщения TB_GETBUTTON.

Поле *iString* структуры TBBUTTON — это индекс (относительно нуля) текстовой метки кнопки. Возможно добавление строк в список текстовых строк панели инструментов путем посылки сообщения TB_ADDSTRING.

Пример создания панели инструментов

Ниже приведен пример создания панели инструментов с маленькими (16x16) изображениями кнопок, взятыми из стандартного битового образа, поддерживаемого библиотекой элементов управления общего пользования:

```
HWND hwndToolBar = CreateToolbarEx(
       hwndParent,
       WS_CHILD | WS_VISIBLE | WS_CLIPSIBLINGS | CCS_TOP | TBSTYLE_TOOLTIPS,
       1, 0,
       HINST_COMMCTRL,
       IDB_STD_SMALL_COLOR,
       tbb,
       5, 0, 0, 0, 0,
       sizeof(TBBUTTON)
);
```

Тринадцать параметров функции *CreateToolbarEx* позволяют ей построить дочернее окно панели инструментов, загрузить простой битовый образ, содержащий изображения кнопок, создать пять кнопок, запросить поддержку окон подсказки и установить предполагаемую версию окна панели инструментов.

Ниже приведен массив TBBUTTON, необходимый для определения атрибутов конкретных кнопок панели инструментов:

```
TBBUTTON tbb[] =
{
       STD_FILENEW,  1, TBSTATE_ENABLED, TBSTYLE_BUTTON,0, 0, 0, 0,
       STD_FILEOPEN, 2, TBSTATE_ENABLED, TBSTYLE_BUTTON,0, 0, 0, 0,
       STD_FILESAVE, 3, TBSTATE_ENABLED, TBSTYLE_BUTTON,0, 0, 0, 0,
       STD_PRINT,    4, TBSTATE_ENABLED, TBSTYLE_BUTTON,0, 0, 0, 0,
       STD_PRINTPRE, 5, TBSTATE_ENABLED, TBSTYLE_BUTTON,0, 0, 0, 0
};
```

Альтернативой вызову функции *CreateToolbarEx* для создания панели инструментов может быть вызов функции *CreateWindow* и последующая посылка сообщений для инициализации различных аспектов панели инструментов. Ниже приведен код, который выполняет действия, аналогичные приведенному выше вызову функции *CreateToolbarEx*.

```
HWND hwndToolBar = CreateWindow(
       TOOLBARCLASSNAME,
       NULL,
       WS_CHILD | WS_VISIBLE | WS_CLIPSIBLINGS | CCS_TOP | TBSTYLE_TOOLTIPS,
       0, 0, 0, 0,
       hwndParent,
       (HMENU) 1,
       hInst,
       0
);

// устанавливаем версию как размер структуры TBBUTTON

ToolBar_ButtonStructSize( hwndToolBar );

ToolBar_AddBitmap( hwndToolBar, 1, &tbbitmap );

// создаем кнопки

ToolBar_AddButtons( hwndToolBar, 5, tbb );
```

Требуется дополнительное определение данных типа структуры TBADDBITMAP, которые задают описатель экземпляра и идентификатор ресурса битового образа панели инструментов:

```
TBADDBITMAP tbbitmap =
{
       HINST_COMMCTRL, IDB_STD_SMALL_COLOR
};
```

После создания панели инструментов, она посылает своему родительскому окну сообщения WM_COMMAND при нажатии на кнопку. Кроме обработки этих сообщений требуется программа для изменения размеров панели инструментов при изменении размеров родительского окна.

Перемещение и изменение размеров панели инструментов

Свойства панели инструментов автоматического изменения размеров и автоматического размещения начинают работать, когда панели инструментов посылается сообщение TB_AUTOSIZE. Обычно, это сообщение посылается тогда, когда родительское окно панели инструментов изменяет размер, т. е. получает сообщение WM_SIZE:

```
SendMessage( hwndToolBar, TB_AUTOSIZE, 0, 0L );
```

Вместо этого возможно использование макроса, определенного в файле COMCTHLP.H на прилагающемся компакт-диске:

```
ToolBar_AutoSize( hwndToolBar );
```

Реакция панели инструментов на это сообщение определяется флагами ее стиля. Например, панель инструментов стиля CCS_TOP или CCS_BOTTOM устанавливает свое местоположение и размеры. С другой стороны, панель инструментов стиля CCS_NORESIZE игнорирует это сообщение, и требует явной установки ее местоположения и размеров.

Поддержка элемента управления подсказка

Элемент управления подсказка — это маленькое окно, содержащее текст подсказки. Обычно, подсказка активизируется, когда курсор мыши оказывается в специфической области кнопки панели инструментов. Однако существует возможность активизировать ее в любое время. Элемент управления подсказка хранит список "горячих" (hot) областей, которые в контексте окон подсказки называются инструментами (tool), и которые могут быть или прямоугольными областями в окне, или окнами целиком. Окна подсказки могут появляться у инструментов, расположенных в окне любого типа, хотя мы рассмотрим их в связи с панелью инструментов.

Панели инструментов, построенные с флагом TBSTYLE_TOOLTIPS, имеют элемент управления подсказка. Каждая кнопка, добавляемая в панель инструментов, регистрирует инструмент подсказки. Для модификации подсказки панели инструментов, например для добавления комбинированного списка в список инструментов, необходимо получить описатель окна подсказки путем посылки сообщения TB_GETTOOLTIPS панели инструментов. Затем вы модифицируете окно подсказки путем посылки сообщений (имеющих префикс TTM_) непосредственно элементу управления — подсказке.

Когда подсказка становится активной, она делает запрос текста для отображения путем посылки сообщения WM_NOTIFY с кодом уведомления TTN_NEEDTEXT. Панель инструментов передает это сообщение своему родительскому окну, которое реагирует заполнением структуры TOOLTIPTEXT. Ниже приведен пример того, как необходимо обрабатывать этот запрос в случае, если панель инструментов состоит из трех кнопок и комбинированного списка:

```
case WM_NOTIFY:
{
        LPNMHDR pnmh =(LPNMHDR) lParam;
        LPSTR  pReply;

        // получить текст подсказки

        if(pnmh->code == TTN_NEEDTEXT)
        {
                LPTOOLTIPTEXT lpttt =(LPTOOLTIPTEXT) lParam;
                switch( lpttt->hdr.idFrom )
                {
                case 0:
                        pReply = "First Button";
                        break;
                case 1:
                        pReply = "Second Button";
                        break;
                case 2:
                        pReply = "Third Button";
                        break;
                default:
                        if((lpttt->uFlags & TTF_IDISHWND) &&
                           (lpttt->hdr.idFrom ==(UINT) hwndCombo))
                                pReply = "Combo box";
                        else
                                pReply = "Unknown";
                }
                lstrcpy(lpttt->szText, pReply);
        }
```

```
}
```

Одно странное свойство этого фрагмента кода состоит в том, что параметр *lParam* сообщения WM_NOTIFY преобразуется к двум различным типам указателя: к указателю на NMHDR и к указателю на TOOLTIPTEXT. Это объясняется тем фактом, что все сообщения WM_NOTIFY передают структуру типа NMHDR, содержащую детали, относящиеся к каждому типу элемента управления общего пользования. Структура типа TOOLTIPTEXT содержит структуру типа NMHDR как первый член данных.

В файле WINUSER.H структура NMHDR определена так:

```
typedef struct tagNMHDR
{
       HWND   hwndFrom;
       UINT   idFrom;
       UINT   code;
} NMHDR;

typedef NMHDR FAR *LPNMHDR;
```

Третий член этой структуры *code* содержит код уведомления, т. е. причины, по которой было послано сообщение WM_NOTIFY. Он может принимать значение одного из основных кодов уведомления (таких как NM_CLICK или NM_SETFOCUS) или специфичного кода уведомления для конкретного класса элемента управления (такого как TTN_NEEDTEXT в данном примере). Поле *hwndFrom* идентифицирует окно, пославшее сообщение, *idFrom* — или идентификатор окна, или идентификатор конкретного элемента (такого как кнопка панели инструментов) от имени которого было послано сообщение.

Когда подсказка посылает уведомление TTN_NEEDTEXT, она передает указатель на структуру данных, уникальную для этого кода уведомления: TOOLTIPTEXT. Эта структура определена в файле COMMCTRL.H следующим образом:

```
typedef struct tagTOOLTIPTEXTA
{
       NMHDR         hdr;
       LPSTR         lpszText;
       char          szText[80];
       HINSTANCE     hInst;
       UINT          uFlags;
} TOOLTIPTEXTA, FAR *LPTOOLTIPTEXTA;
```

Первый член структуры, *hdr*, содержит данные типа NMHDR, которые должны сопровождать любое сообщение WM_NOTIFY.

При получении сообщения TTN_NEEDTEXT первая задача — проверить, какая кнопка (или окно) была выбрана. Это осуществляется путем проверки поля *idFrom* структуры NMHDR, которое может быть либо индексом кнопки, если мышь над кнопкой, либо описателем окна, если мышь над дочерним окном панели инструментов. Поле *uFlags* содержит флаг TTF_IDISHWND для индикации того, что это описатель окна, а не индекс кнопки.

При получении уведомления TTN_NEEDTEXT существует три пути для передачи текста в подсказку: путем передачи указателя на строку в поле *lpszText*, путем копирования строки в буфер *szText* или путем задания идентификатора строки-ресурса. Так же как и при работе с другими ресурсами, строка-ресурс задается с помощью описателя экземпляра и уникального идентификатора. Описатель экземпляра копируется в поле *hinst*, а идентификатор — в поле *lpszText*, используя макрос MAKEINTRESOURCE:

```
lpttt->hinst = hInstance;
lpttt->lpszText = MAKEINTRESOURCE(100);
```

Добавление дочерних окон в панель инструментов

Панели инструментов поддерживают только кнопки, поэтому для помещения чего-либо отличного от кнопки в панель инструментов следует создать дочернее окно. Одним из наиболее частых типов окон, добавляемых в панель инструментов, являются комбинированные списки. Поэтому, комбинированные списки будут находиться в фокусе нашего рассмотрения (несмотря на то, что любой тип окна может быть расположен на панели инструментов). Поскольку основы создания дочерних окон рассмотрены в главе 8, здесь мы остановимся на некоторых подробностях: резервирование места в панели инструментов для дочернего окна, изменение размеров панели инструментов, поддержка окон подсказки.

Резервирование места для дочернего окна на панели инструментов

В книге "*Programming the Windows 95 User Interface*" Nancy Cluts предлагает включать разделители (кнопки стиля TBSTYLE_SEP) для сохранения пустого пространства при создании дочернего окна на панели инструментов. Вам

предстоит поэкспериментировать, чтобы определить правильное число разделителей, а пока программа GADGETS, приведенная далее в этой главе, использует 20 разделителей для резервирования места для комбинированного списка.

Присутствие разделителей упрощает расчет координат для окна, которые необходимо задать при вызове функции *CreateWindow*. Посылается сообщение TB_GETITEMRECT для получения координат в пикселях любого элемента панели инструментов — кнопки или разделителя, и возвращается четыре координаты прямоугольника указанного элемента. Ниже показано, каким образом программа GADGETS вычисляет размеры пространства, доступного для комбинированного списка, используя макрос сообщения из файла COMMCTRL.H (находится на прилагающемся компакт-диске):

```
RECT   r;
int    x, y, cx, cy;

// Вычисляем координаты для комбинированного списка

ToolBar_GetItemRect( hwndTB, 0, &r );
x = r.left;
y = r.top;
ToolBar_GetItemRect( hwndTB, 18, &r );
cx = r.right — x + 1;
```

Левый верхний угол элемента управления устанавливается в точку с координатами (*x, y*), в соответствии с размерами кнопок панели инструментов. Поскольку программа GADGETS использовала 20 разделителей на своей панели инструментов, запрос положения 19-го (или 18 при отсчете с нуля) разделителя, оставляет один разделитель между правым краем комбинированного списка и первой кнопкой.

Другим значением, необходимым для помещения дочернего окна на панель инструментов, является высота, которая для большинства элементов управления будет равна высоте кнопки (*r.bottom — r.top + 1*), полученной из сообщения TB_GETITEMRECT. Но комбинированный список ведет себя несколько иначе, чем другие типы окон, поскольку его часть редактирования (или статическая) изменяет размер самостоятельно с тем, чтобы прийти в соответствие шрифту, и высота, передаваемая в функцию *CreateWindow*, есть возможная высота для ниспадающего окна. Установка значения, достаточного для нескольких строк, хороша тогда, когда вы используете для расчета данные о шрифте, получаемые от функции *GetTextMetrics*. Программа GADGETS этого не делает и устанавливает это значение жестко:

```
cy = 100;
```

Создание элементов управления на панели инструментов

Существует несколько правил, используемых при создании дочерних окон на панели инструментов. Прежде всего, убедитесь, не забыли ли вы включить флаги стиля окна WS_VISIBLE и WS_CHILD. Без флага WS_VISIBLE вызов функции может оказаться успешным, однако окно на экране не появится. Флаг WS_CHILD используется для создания дочернего окна, которое должно иметь родителя, на поверхности которого будет расположено дочернее окно. Это вызывает появление проблемы, какое окно должно быть родительским для дочернего.

Очевидным выбором родительского окна для окон на панели инструментов является само окно панели инструментов, но это может привести к странным проявлениям. В частности, предопределенные элементы управления посылают уведомляющие сообщения (в виде сообщений WM_COMMAND) своему родителю, что может иметь неожиданный результат. Например, программа GADGETS постоянно сбивается в Windows 95, когда ее окно комбинированного списка панели инструментов создается с окном панели инструментов в роли родительского окна, а идентификатор окна комбинированного списка равен 101. (Заметим, что эта комбинация не вызывает никаких проблем в Windows NT 3.51. Это связано с отличающимися в этих операционных системах версиями библиотеки COMCTL32.DLL.) Изменение управляющего идентификатора — кажущееся решение этой проблемы, поскольку такой путь оставляет шанс (хоть и незначительный) на то, что в будущих версиях библиотеки элементов управления общего пользования будет использоваться идентификатор, который используем мы, и тогда указанная проблема возникнет вновь. И все же, панель инструментов должна быть родительским окном для того окна, которое хочет располагаться на ее поверхности.

Решение этого противоречия состоит в задании другого окна в качестве родительского в вызове функции *CreateWindow* — предпочтительнее одного из тех, которые имеют вашу собственную оконную процедуру. После того, как дочернее окно будет создано, оно помещается на панель инструментов путем вызова функции *SetParent*. Элементы управления диалогового окна лояльны к этому и посылают свои уведомляющие сообщения WM_COMMAND их первоначальному родительскому окну, что позволяет вам обрабатывать уведомляющие сообщения по своему усмотрению. Таким образом, уведомляющие сообщения посылаются одному окну, а дочернее окно физически располагается на другом окне — панели инструментов. Программа GADGETS делает эти два шага следующим образом:

```
hwndCombo = CreateWindow(
```

```
        "combobox",
        NULL,
        WS_CHILD | WS_VISIBLE | CBS_DROPDOWN,
        x, y, cx, cy,
        hwndParent,
        (HMENU) 100,
        hInst,
);

// Установление панели инструментов как родительского окна для комбинированного списка //
SetParent( hwndCombo, hwndTB );
```

Изменение размеров панели инструментов с дочерними окнами

Другая сложность, возникающая при использовании дочерних окон на панели инструментов состоит в том, что необходимо быть уверенным, что размеры самой панели инструментов достаточны для размещения дочерних окон. Между прочим, эта проблема актуальна для оболочек типа старой Windows, такой как Windows NT 3.51, а не для новейшей оболочки Windows 95. Но поскольку некоторые из пользователей вашей программы могут работать на указанной версии Windows NT, следует тестировать вашу программу в обеих средах.

Простейшее решение, для всех элементов управления кроме комбинированных списков, состоит в изменении размеров окна самим окном. Работа с комбинированным списком более трудоемка, поскольку он изменяет свои размеры в зависимости от текущего шрифта. Поэтому, вы можете косвенно воздействовать на размеры комбинированного списка, устанавливая его шрифт. Это делается путем создания шрифта необходимого размера и передачи его в комбинированный список путем посылки сообщения WM_SETFONT. Пока элемент управления короче, чем кнопки панели инструментов, обработка сообщения TB_AUTOSIZE будет выполнять всю хлопотливую работу по приведению размеров панели инструментов в соответствие с изменяющимися размерами ее родительского окна. (См. предшествующее обсуждение размещения и изменения размеров панели инструментов.)

Если вы не можете заставить дочернее окно быть меньше, необходимо сделать панель инструментов больше. Программа GADGETS демонстрирует этот подход и является основой для дальнейшего обсуждения.

Для получения полного контроля над размерами и местоположением панели инструментов создавайте панель инструментов с использованием флага стиля CCS_NORESIZE. Вопреки тому, что можно было бы подумать на основе имени, этот флаг не дает панели инструментов возможности самостоятельно изменять свои размеры и расположение. Если установлен этот флаг, то сообщение TB_AUTOSIZE не приводит ни к каким действиям. Поэтому, перемещение и установка местоположения панели инструментов целиком в вашей власти. Когда родительское окно получает сообщение WM_SIZE, вам следует рассчитать необходимые координаты местоположения (*x, y*) и размеры окна, а затем произвести необходимые изменения путем вызова функции типа *MoveWindow*.

Для приложений с простой панелью инструментов в верхней части родительского окна местоположение определяется просто — (*0, 0*). Ширина также определяется просто, поскольку сообщение WM_SIZE родительскому окну содержит в младшем слове параметра *lParam* ширину окна. Для изменения размеров панели инструментов в ответ на сообщение WM_SIZE необходим примерно такой код:

```
case WM_SIZE:
{
        int cx = LOWORD(lParam);
        MoveWindow(hwndToolBar, 0, 0, cx, cyToolBar, TRUE);
        [другие строки программы]
        return 0;
}
```

Остается пока непонятным, как определить высоту панели инструментов (*cyToolBar*). В программе GADGETS проблема решается в два этапа: сначала делается запрос определения размера комбинированного списка, затем запрос определения размера элемента панели инструментов, к которому добавляется дополнительная величина, равная 5 (величина поля, строго определенная в панели инструментов). Высота панели инструментов и есть максимум из этих двух значений:

```
// вычислить высоту панели инструментов
GetWindowRect(hwndCombo, &r);
cyToolBar = r.bottom — r.top + 1;
cyToolBar += y;
cyToolBar +=(2 * GetSystemMetrics(SM_CYBORDER));
ToolBar_GetItemRect(hwndTB, 0, &r);
cyToolBar = max(cyToolBar, r.bottom + 5);
```

Поддержка окон подсказки для дочерних окон панели инструментов

Кроме обработки уведомления TTN_NEEDTEXT создание работоспособного окна подсказки для кнопок панели инструментов — дело достаточно простое. Для дочерних окон в панели инструментов требуется еще одно дополнительное усилие: добавление дочернего окна в список инструментов, создаваемый элементом управления подсказка.

Добавление дочернего окна в список инструментов, создаваемый элементом управления подсказка, требует посылки сообщения TTM_ADDTOOL окну подсказки. Но для начала необходимо получить описатель окна подсказки. Это делается путем посылки панели инструментов сообщения TB_GETTOOLTIP:

```
hwndTT = ToolBar_GetToolTips(hwndToolBar);
```

Сообщение TTM_ADDTOOL требует указатель на структуру типа TOOLINFO, которая описывает инструмент элемента управления подсказка. Структура TOOLINFO определена в файле COMMCTRL.H как показано ниже:

```
typedef struct tagTOOLINFOA
{
       UINT   cbSize;
       UINT   uFlags;
       HWND   hwnd;
       UINT   uId;
       RECT   rect;
       HINSTANCE hinst;
       LPSTR  lpszText;
} TOOLINFOA, NEAR *PTOOLINFOA, FAR *LPTOOLINFOA;
```

Поле *cbSize* структуры TOOLINFO соответствует информации о версии, специфичной для элемента управления подсказка, и должен быть равен размеру структуры TOOLINFO. Пренебрежение этим правилом может привести к тому, что подсказка проигнорирует это сообщение.

Поле *uFlags* позволяет иметь некоторую гибкость при задании инструмента. Первый флаг, TTF_IDISHWND, требуется тогда, когда в качестве инструмента передается описатель дочернего окна. Он показывает, что в поле *uId* находится описатель окна. Другой флаг, TTF_CENTERTIP, заставляет окно подсказки выводиться в центре по горизонтали под инструментом, а не как по умолчанию — ниже и справа относительно курсора мыши. Третий флаг, TTF_SUBCLASS, требует, чтобы элемент управления подсказка создал новую оконную процедуру для отслеживания потока сообщений. Без этого флага вам пришлось бы создавать другой механизм для передачи подробностей сообщений мыши элементу управления подсказка.

Третье поле, *hwnd*, идентифицирует окно, содержащее инструмент. Конечно, в данном случае это окно панели инструментов.

Поле *uId* содержит произвольный идентификатор, который создатель инструмента (например, панель инструментов) присваивает каждому инструменту. Например, панель инструментов устанавливает каждой своей кнопке индекс относительно нуля и передает этот индекс в окно подсказки, когда добавляет каждую кнопку в список подсказки. Когда этот идентификатор содержит описатель окна, как это бывает, если целое дочернее окно становится инструментом, то флаг TTF_IDISHWND в поле *uFlags* уведомляет об этом подсказку.

Местоположение инструмента в его родительском окне задается структурой *rect*, что позволяет элементу управления подсказка осуществлять контроль положения указателя мыши для каждой конкретной кнопки на панели инструментов. В случае, если окно целиком является инструментом, т. е. если указан флаг TTF_IDISHWND, то это поле игнорируется.

Текст, который выводится в качестве подсказки на дисплей, определяется комбинацией значений полей *hinst* и *lpszText*, задающих текстовую строку как ресурс. В качестве альтернативы, возможно указать, что необходим запрос в уведомляющем сообщении TTN_NEEDTEXT. Для этого надо указать в поле *lpszText* значение LPSTR_TEXTCALLBACK.

Ниже показано, как программа GADGETS добавляет поддержку окна подсказки для комбинированного списка, находящегося в панели инструментов:

```
TOOLINFO ti;
ZeroMemory(&ti, sizeof(TOOLINFO));
ti.cbSize = sizeof(TOOLINFO);
ti.uFlags = TTF_IDISHWND | TTF_CENTERTIP | TTF_SUBCLASS;
ti.hwnd = hwndToolBar;
ti.uId =(UINT)(HWND) hwndComboBox;
ti.lpszText = LPSTR_TEXTCALLBACK;
bSuccess = ToolTip_AddTool(hwndTT, &ti);
```

Ценность комбинированных списков в том, что они состоят из комбинации двух или более элементов управления: контейнер комбинированного списка, окно редактирования или статическое окно и список. Для того чтобы обеспечить полную поддержку окон подсказки, видимые компоненты должны быть добавлены в список инструментов подсказки. Программа GADGETS добавляет поддержку окна подсказки для окна редактирования следующим образом:

```
// Добавление окна подсказки для окна редактирования комбинированного списка
hwndEdit = GetWindow(hwndComboBox, GW_CHILD);
ti.cbSize = sizeof(TOOLINFO);
ti.uFlags = TTF_IDISHWND | TTF_CENTERTIP | TTF_SUBCLASS;
ti.hwnd = hwndToolBar;
ti.uId =(UINT)(HWND) hwndEdit;
ti.lpszText = LPSTR_TEXTCALLBACK;
bSuccess = ToolTip_AddTool(hwndTT, &ti);
```

Если элемент управления подсказка знает о дочерних окнах, то подсказка появляется на экране тогда, когда курсор мыши останавливается над дочерним окном. В определении этого момента есть некая магия, что возможно благодаря введению новой оконной процедуры (window subclassing), на что указывает флаг TTF_SUBCLASS. Новая оконная процедура отслеживает сообщения мыши, и когда замечает подходящее, посылает сообщение TTM_RELAYEVENT в элемент управления подсказка.

Если по каким-либо причинам вы решите, что использование техники введения новой оконной процедуры нежелательно, например, потому что оно уже было сделано, то существует несколько альтернативных путей. Вы можете модифицировать оконную процедуру дочернего элемента управления и просто отправлять сообщение TTM_RELAYEVENT при получении сообщения от мыши. Также возможно добавить ловушку (hook) типа WH_GETMESSAGE для поиска сообщений мыши для конкретного окна. Вероятно, наиболее простой путь — это добавление небольшого фрагмента кода в цикл обработки сообщений вашей программы. Поскольку это основной путь, по которому проходят все сообщения мыши (а также клавиатуры, таймера и др.), то этот вариант хорош в тех случаях, когда применение флага TTF_SUBCLASS нецелесообразно. Ниже приведен фрагмент цикла обработки сообщений программы GADGETS, если бы в ней не использовался этот флаг:

```
while(GetMessage(&msg, NULL, 0, 0))
{
	if(pMsg->hwnd == hwndCombo || pMsg->hwnd == hwndEdit)
	{
		if(pMsg->message >= WM_MOUSEFIRST && pMsg->message <= WM_MOUSELAST)
		{
			ToolTip_RelayEvent(hwndTT, pMsg);
		}
	}
	TranslateMessage(&msg);
	DispatchMessage(&msg);
}
```

Изменение конфигурации панели инструментов

Конфигурация панелей инструментов может быть изменена пользователем во время работы программы. Для этого существует диалоговое окно изменения конфигурации панели инструментов (Customize Toolbar dialog box), показанное на рис. 12.6. В этом окне диалога находятся два списка: первый — список доступных кнопок, второй — список кнопок, находящихся в данный момент на панели инструментов. Пользователь имеет возможность щелкнуть на кнопках Add (добавить) и Remove (удалить) для перемещения элементов между двумя этими списками. Кроме того, оба окна списков поддерживают операции типа drag and drop, поэтому пользователь может перетаскивать элементы из одного списка в другой. Активизируется окно изменения конфигурации панели инструментов путем посылки ей сообщения TB_CUSTOMIZE.

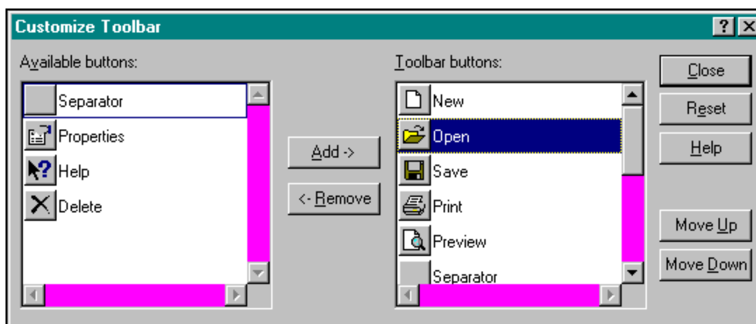


Рис. 12.6 Диалоговое окно изменения конфигурации панели инструментов

Два флага стиля обеспечивают доступ к возможностям панели инструментов с помощью мыши. Панель инструментов, созданная с использованием флага CCS_ADJUSTABLE, выводит на экран диалоговое окно изменения конфигурации в ответ на двойной щелчок левой клавиши мыши на теле панели инструментов. (Этот флаг не требуется, если необходимо, чтобы диалоговое окно изменения конфигурации панели инструментов появлялось под управлением программы; просто необходимо послать сообщение TB_CUSTOMIZE.) Кроме того, пользователь может перемещать кнопки по панели инструментов с помощью клавиатуры и мыши: нажимая и удерживая клавишу <Shift> и левую кнопку мыши. Пользователь может удалить кнопку из панели инструментов путем нажатия и удержания левой кнопки мыши и перетаскивания кнопки за пределы панели инструментов. Использование флага TBSTYLE_ALTDRAG позволяет сделать так, что для операций перетаскивания кнопок будет необходимо использование клавиши <ALT> (как в программе Microsoft Word for Windows) вместо клавиши <Shift>.

Ключ к изменению конфигурации панели инструментов лежит в правильной обработке соответствующих уведомляющих сообщений. Даже перед тем как появиться на экране, диалоговое окно изменения конфигурации панели инструментов посылает три инициализирующих запроса: TBN_QUERYINSERT, TBN_QUERYDELETE и TBN_GETBUTTONINFO. Все уведомляющие сообщения, связанные с диалоговым окном изменения конфигурации панели инструментов, приведены в следующей таблице:

Уведомляющее сообщение	Описание
TBN_BEGINADJUST и TBN_ENDADJUST	Существование диалогового окна изменения конфигурации ограничено этими двумя уведомлениями.
TBN_QUERYINSERT	Запрашивает разрешение на вставку в заданной позиции. Когда диалоговое окно запускается, возвращайте TRUE. В противном случае диалоговое окно выведено не будет.
TBN_QUERYDELETE	Запрашивает разрешение на удаление кнопки. Возвращайте TRUE, если "да", и FALSE — если "нет".
TBN_GETBUTTONINFO	Наборы этих сообщений запрашивают информацию обо всех кнопках, которые могут быть выведены на панель инструментов. Кнопки, не находящиеся в данный момент на панели инструментов заносятся в список доступных кнопок.
TBN_TOOLBARCHANGE	Кнопки на панели инструментов были перемещены, удалены или вставлены.
TBN_RESET	Пользователь нажал кнопку Reset в диалоговом окне изменения конфигурации.
TBN_CUSTHELP	Пользователь нажал кнопку Help в диалоговом окне изменения конфигурации.

Уведомление TBN_QUERYINSERT запрашивает разрешение на вставку любой новой кнопки в панель инструментов. Если ответ отрицателен (возвращаемое значение равно нулю или FALSE), то диалоговое окно на мгновение высвечивается на экране и исчезает. Это может удивить пользователя. Для того, чтобы быть уверенным в том, что диалоговое окно изменения конфигурации панели инструментов будет выведено на экран, необходимо в ответ на первый запрос вернуть "да" (не ноль или TRUE):

```
case WM_NOTIFY:
{
	LPNMHDR pnmh =(LPNMHDR) lParam;

	// Разрешаем изменение конфигурации панели инструментов
	if(pnmh->code == TBN_QUERYINSERT)
	{
		return 1;
	}
}
```

Набор уведомлений TBN_QUERYDELETE — одно на каждую кнопку панели инструментов — посылаются с запросом о том, может ли быть удалена конкретная кнопка. Ответ вызывает открытие доступа или закрытие доступа к кнопке Remove в диалоговом окне: TRUE разрешает удаление кнопки, FALSE — запрещает.

Последовательности уведомлений TBN_GETBUTTONINFO обеспечивают родительскому окну панели управления возможность идентифицировать все кнопки, которые могут располагаться в панели инструментов. Для каждого запроса заполняется структура типа TBBUTTON для каждой возможной кнопки, включая текст строки, который выводится рядом с изображением кнопки в диалоговом окне изменения конфигурации панели инструментов. Возврат значения TRUE означает, что вы заполнили данные внутри структуры и хотите вновь получить это уведомление. Возврат значения FALSE означает, что кнопок больше нет. В результате этого запроса заполняется список доступных кнопок (Available buttons) диалогового окна изменения конфигурации панели инструментов. Однако, в этот список заносятся только те кнопки, которые в данный момент времени не находятся в панели инструментов. Ниже приведен фрагмент программы для ответа на это сообщение для двух кнопок:

```
LPTBNOTIFY ptbn =(LPTBNOTIFY) lParam;
switch(ptbn->iItem)
{
case 0:
       lstrcpy(ptbn->pszText, "Help");
       ptbn->tbButton.iBitmap = STD_HELP;
       ptbn->tbButton.idCommand = 11;
       ptbn->tbButton.fsState = TBSTATE_ENABLED;
       ptbn->tbButton.fsStyle = TBSTYLE_BUTTON;
       ptbn->tbButton.dwData = 0;
       ptbn->tbButton.iString = 10;
       return 1;
case 1:
       lstrcpy(ptbn->pszText, "Delete");
       ptbn->tbButton.iBitmap = STD_DELETE;
       ptbn->tbButton.idCommand = 12;
       ptbn->tbButton.fsState = TBSTATE_ENABLED;
       ptbn->tbButton.fsStyle = TBSTYLE_BUTTON;
       ptbn->tbButton.dwData = 0;
       ptbn->tbButton.iString = 11;
       return 1;
default:
       return 0;
}
```

Если у вас уже есть массив описаний TBBUTTON, то самый быстрый путь ответа на это сообщение состоит в использовании функции *memcpy* для копирования TBBUTTON для каждого получаемого сообщения. Ниже показано, как программа GADGETS делает это:

```
lstrcpy(ptbn->pszText, GetString(ptbn->iItem));
memcpy(&ptbn->tbButton, &tbb[iButton], sizeof(TBBUTTON));
```

Строка состояния

Строки состояния (status bar) — окна только для вывода информации, часто располагающиеся в нижней части главного окна программы. Наиболее частое использование строки состояния состоит в том, что она используется для отображения подробного описания пунктов меню при их просмотре пользователем, также как официант комментирует тот или иной пункт меню в ресторане. Когда меню не просматривается, программы часто отображают в строке состояния сопутствующую информацию. Также часто отображается состояние клавиш сдвига — <Caps Lock>, <Num Lock>, <Scroll Lock>. В программах текстовых процессоров часто отображается также текущая страница, строка, столбец.

Строка состояния может работать в двух режимах, поэтому она может использоваться для описания элементов меню и вывода другой программной информации. В режиме описания пунктов меню (простом режиме, "simple mode") строка состояния расширяется для отображения простой строки текста. В режиме отображения состояния программы в строке состояния отображается одно или более окон, каждое из которых называется "частью" строки статуса. Конкретная часть строки состояния может быть создана как ниша с рамкой, которая выглядит приподнятой над поверхностью окна строки состояния, или без рамки. Кроме того, существует возможность добавить в строку состояния дочерние окна, такие как часы или индикатор прогресса. При переключении строки состояния из простого режима в режим отображения состояния программы и обратно строка состояния сохраняет один набор скрытых окон в то время, как отображается другой набор видимых окон.

Создание строки состояния

Простейший путь создания строки состояния состоит в вызове функции *CreateStatusWindow*:

```
hwndStatusBar = CreateStatusWindow(
       WS_CHILD | WS_VISIBLE | WS_CLIPSIBLINGS | CCS_BOTTOM,
       "Ready", hwndParent, 2
);
```

Эта функция вызывает функцию *CreateWindow*, которая создает дочернее окно с родительским окном *hwndParent*, с текстом окна "Ready" (этот текст отображается в первом окне строки состояния), идентификатором, равным 2. Флаги стиля окна заданы в первом параметре.

В приведенной ниже таблице дано описание всех флагов стилей, которые могут использоваться для создания удобной строки состояния. Вероятно, вам чаще всего придется задействовать флаг SBARS_SIZEGRIP, поскольку он задает вывод диагональной штриховки в правом углу строки состояния. Остальные флаги стиля модифицируют начальное состояние и местоположение строки состояния так, чтобы она могла находиться в другом месте, а не в нижней части рабочей области родительского окна, что определяется флагом CCS_BOTTOM.

Категория	Флаг стиля	Описание
Представление	SBARS_SIZEGRIP	Отображает диагональную штриховку в правом углу строки состояния. Эта область служит для изменения размеров родительского окна.
Начальное положение	CCS_TOP	Помещает строку состояния в верхнюю часть родительского окна.
	CCS_BOTTOM	Помещает строку состояния в нижнюю часть родительского окна (по умолчанию).
	CCS_NOMOVEY	Запрещает перемещение относительно оси *y*.
Запрет автоматического изменения размеров и местоположения	CCS_NOPARENTALIGN	Строка состояния устанавливает свою высоту (*cy*). Но не устанавливает свое местоположение (*x,y*) и ширину (*cx*). Для соответствующей обработки необходимо после создания посылать сообщение об изменении размеров.
	CCS_NORESIZE	Запрещает все автоматические перемещения и изменения размеров. Это запрещает следующие флаги стиля: CCS_TOP, CCS_BOTTOM, CCS_NOMOVEY и CCS_NOPARENTALIGN. Вы должны явно задать размеры и положение строки состояния.

Перемещение и изменение размеров строки состояния

Когда родительское окно строки состояния изменяет размер (при получении сообщения WM_SIZE), строка состояния должна занять новое положение и приобрести другие размеры, чтобы остаться в нижней части рабочей области родительского окна. Панель инструментов изменяет размеры в ответ на сообщение TB_AUTOSIZE (см. ранее в этой главе). У строки статуса нет аналогичного сообщения. Вместо него используется примерно следующие:

```
case WM_SIZE:
       {
              int cxParent = LOWORD(lParam);
              int cyParent = HIWORD(lParam);
              int x, y, cx, cy;
              RECT rWindow;

              // Оставить высоту окна строки состояний без изменений
              GetWindowRect(hwndStatusBar, &rWindow);
              cy = rWindow.bottom — rWindow.top;

              x = 0;
              y = cyParent — cy;
              cx = cxParent;
              MoveWindow(hwndStatusBar, x, y, cx, cy, TRUE);
       }
```

Этот код сохраняет высоту строки состояния без изменений и модифицирует ее ширину и местоположение таким образом, чтобы занять нужное место в рабочей области родительского окна.

Поддержка просмотра меню

Пользователи ожидают от Windows — программ отображения вспомогательной информации в строке состояния о том, какую функцию выполняет тот или иной пункт меню. Даже случайный пользователь Windows быстро учится тому, что под пунктом меню из одного слова скрывается значительно более широкое действие. И хотя опытные пользователи легко понимают смысл стандартных команд меню, иногда им приходится тратить много сил, чтобы понять смысл специфичных пунктов меню конкретной программы.

Меню окна посылает сообщение WM_MENUSELECT, когда пользователь просматривает пункты меню, и сообщение WM_COMMAND, когда пользователь выбирает пункт меню (см. главу 10). Для поддержки отображения вспомогательной информации о пунктах меню следует обрабатывать сообщение WM_MENUSELECT.

Для упрощения процесса обработки этого сообщения и отображения текста в строке состояния библиотекой элементов управления общего пользования поддерживается функция *MenuHelp*. Эта функция предполагает наличие таблицы строк, содержащей тексты вспомогательной информации для отображения, и структуры данных, связывающей пункты меню с идентификаторами текстовых строк из таблицы. Функция *MenuHelp* определена следующим образом:

```
void MenuHelp(
       UINT          uMsg,                // WM_MENUSELECT
       WPARAM        wParam,              // параметр wParam
       LPARAM        lParam,              // параметр lParam
       HMENU         hMainMenu,           // описатель главного меню
       HINSTANCE     hInst,               // описатель экземпляра
       HWND          hwndStatus,          // описатель окна строки состояния
       UINT FAR      *lpwIDs
);                                        // таблица строк
```

Первый параметр, *uMsg*, должен быть равен WM_MENUSELECT, хотя наличие этого параметра (и его описание в документации по Win32) говорит о том, что обработка сообщения WM_COMMAND была заложена еще при разработке функции. В качестве второго и третьего параметров передаются параметры *wParam* и *lParam* оконной процедуры. Эти три параметра, взятые вместе, описывают то, какую часть системы меню просматривает пользователь, и является ли он пунктом, всплывающего меню или системного.

Шестой параметр, *hwndStatus*, это описатель окна строки состояния. Функция *MenuHelp* посылает специфичное для строки состояния сообщение SB_SIMPLE для установки строки состояния в режим одного окна (простой) и для отображения соответствующего текста. Позднее, когда пользователь прекращает просмотр, функция *MenuHelp* посылает другое сообщение SB_SIMPLE для возврата строки состояния в режим многих окон (непростой).

Функция *MenuHelp* использует другие три параметра — *hMainMenu*, *hInst* и *lpwIDs* — для определения того, какую строку необходимо отобразить при просмотре элемента меню. Параметр *hInst* идентифицирует модуль, который может быть описателем экземпляра DLL или описателем экземпляра выполняемой программы, и является владельцем таблицы, содержащей строки меню (которые, как можно предположить, загружаются с помощью функции *LoadString*).

Возможность сделать так, чтобы функция *MenuHelp* работала правильно, состоит в том, чтобы передать правильное значение в четвертом и седьмом параметрах: *hMainMenu* и *lpwIDs*. Правильная установка этих значений является небольшим трюкачеством, поскольку существует три элемента для рассмотрения: командное меню, всплывающее меню и системное меню. Другая сложность состоит в том, что документация по Win32 говорит о том, что *lpwIDs* — массив, в котором осуществляет поиск функция *MenuHelp*. Несмотря на то, что имеет смысл использовать массив для хранения базовых значений строковых ресурсов, вы должны будете сами делать грамматический разбор массива, поскольку функция *MenuHelp* не делает этого.

Просмотр элементов меню

Функция *MenuHelp* вычисляет идентификатор ID строкового ресурса для элемента меню путем добавления значения идентификатора команды (полученного из младшего слова *wParam*) к значению, на которое ссылается *lpwIDs*. Например, следующий вызов функции *MenuHelp* отображает строковый ресурс с номером 125 в строке состояния:

```
UINT uiStringBase = 100;
WPARAM wParam = 25;
MenuHelp(WM_MENUSELECT, wParam, lParam, NULL, hInst, hwndStatus, &uiStringBase);
```

Как показано в примере, описатель меню *hMainMenu* может быть равен NULL для отображения элементов меню команд, поскольку функция *MenuHelp* не использует это значение для расчета.

Простейший путь привести в соответствие элементы меню команд и строковые ресурсы состоит в том, чтобы присвоить им одинаковые значения. Это позволит установить базу строки в ноль и ее игнорировать.

Просмотр элементов всплывающего меню

В случае всплывающего меню функция *MenuHelp* вычисляет строку-ресурс для отображения в строке состояния путем добавления индекса (с нулевой базой) всплывающего меню к значению, на которое ссылается *lpwIDs*. Для того чтобы это работало правильно, необходимо, чтобы четвертый параметр функции *MenuHelp* — *hMainMenu* — имел значение описателя родительского по отношению к всплывающему меню окна. Структура, приведенная ниже, обеспечивает удобный путь для установки соответствия между описателями меню и базой строковых ресурсов:

```
typedef struct tagPOPUPSTRING
{
       HMENU  hMenu;
       UINT   uiString;
} POPUPSTRING;
```

В программе GADGETS, в которой три пункта меню содержат всплывающие меню, эта структура данных определяется так:

```
POPUPSTRING   popstr[5];
```

и инициализируется при создании строки статуса следующим образом:

```
HMENU     hMainMenu = GetMenu(hwndParent);
popstr[0].hMenu     = hMainMenu;
popstr[0].uiString  = IDS_MAIN_MENU;
popstr[1].hMenu     = GetSubMenu(hMainMenu, 2);
popstr[1].uiString  = IDS_TOOLBAR_MENU;
popstr[2].hMenu     = GetSubMenu(hMainMenu, 3);
popstr[2].uiString  = IDS_STATUSBAR_MENU;
popstr[3].hMenu     = NULL;
popstr[3].uiString  = 0;
```

При получении сообщения WM_MENUSELECT параметр *lParam* содержит описатель меню родительского окна. Работа функции *MenuHelp* по подбору правильного ресурса строки требует от вас поиска в массиве и передачи адреса, как последнего параметра функции *MenuHelp*. Ниже показано, как это реализуется в программе GADGETS:

```
if((fuFlags & MF_POPUP) &&(!(fuFlags & MF_SYSMENU)))
{
       for(iMenu = 1; iMenu < MAX_MENUS; iMenu++)
       {
              if((HMENU) lParam == popstr[iMenu].hMenu)
              {
                     hMainMenu =(HMENU)lParam;
                     break;
              }
       }
}
```

Для того чтобы это работало корректно, *hMainMenu* должен быть установлен в значение описателя родительского меню всплывающего меню. Пока мы рассматривали обработку своих всплывающих меню, мы совершенно забыли о системном меню.

Просмотр системного меню

Функция *MenuHelp* обеспечивает индикацию в строке состояния вспомогательной информации для системного меню и элементов системного меню. Все, что необходимо для этого — параметры сообщения WM_MENUSELECT *wParam* и *lParam* в том же виде, что и для других типов элементов меню. Кроме того, значение *hMainMenu* не должно быть равно реальному описателю системного меню; NULL — вполне подходит.

Пример

Объединим теперь все эти фрагменты для элементов меню, всплывающих меню и системного меню. Ниже приведен код, иллюстрирующий то, каким образом программа GADGETS обрабатывает сообщение WM_MENUSELECT для того, чтобы отобразить вспомогательную информацию в строке состояния:

```
LRESULT Statusbar_MenuSelect(HWND hwnd, WPARAM wParam, LPARAM lParam)
{
     UINT fuFlags =(UINT) HIWORD(wParam);
     HMENU hMainMenu = NULL;
     int iMenu = 0;

     // Обработка несистемных всплывающих меню
     if((fuFlags & MF_POPUP) &&(!(fuFlags & MF_SYSMENU)))
     {
          for(iMenu = 1; iMenu < MAX_MENUS; iMenu++)
          {
               if((HMENU) lParam == popstr[iMenu].hMenu)
               {
                    hMainMenu =(HMENU)lParam;
                    break;
               }
          }
     }

     // Отображение вспомогательной информации в строке состояния
     MenuHelp(WM_MENUSELECT, wParam, lParam, hMainMenu, hInst,
               hwndStatusBar, &((UINT) popstr[iMenu].hMenu));

     return 0;
}
```

Программа GADGETS

Программа GADGETS объединяет вместе три рассмотренных элемента управления: панель инструментов, подсказку, строку состояния. Как показано на рис. 12.7, программа GADGETS имеет панель инструментов, содержащую комбинированный список, и строку состояния с возможностью изменения размеров окна. Она также имеет рабочее окно, которое содержит список для отображения всех кодов уведомлений, получаемых для каждого из этих элементов управления. Для того чтобы дать вам возможность почувствовать различные стили панели инструментов и строки состояния, программа GADGETS дает вам возможность разрешить или запретить эти флаги стилей, чтобы продемонстрировать незамедлительно эффект каждого из этих флагов стиля. На рис. 12.8 приведен исходный текст программы GADGETS.

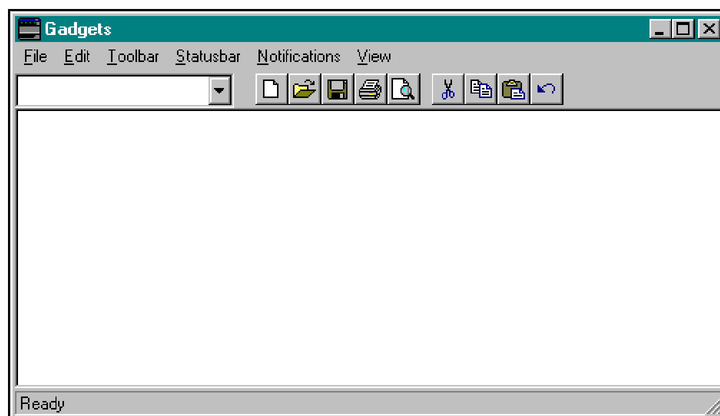


Рис. 12.7 Вывод на экран программы GADGETS

GADGETS.MAK

```
#-----------------------
# GADGETS.MAK make file
#-----------------------

gadgets.exe : gadgets.obj notifdef.obj statbar.obj \
              toolbar.obj tooltip.obj gadgets.res
     $(LINKER) $(GUIFLAGS) -OUT:gadgets.exe gadgets.obj \
         notifdef.obj statbar.obj toolbar.obj tooltip.obj \
         gadgets.res $(GUILIBS)

gadgets.obj : gadgets.c comcthlp.h gadgets.h
     $(CC) $(CFLAGS) gadgets.c

notifdef.obj : notifdef.c notifdef.h
     $(CC) $(CFLAGS) notifdef.c

statbar.obj : statbar.c comcthlp.h gadgets.h
     $(CC) $(CFLAGS) statbar.c

toolbar.obj : toolbar.c comcthlp.h gadgets.h notifdef.h
     $(CC) $(CFLAGS) toolbar.c

tooltip.obj : tooltip.c comcthlp.h gadgets.h notifdef.h
     $(CC) $(CFLAGS) tooltip.c

gadgets.res : gadgets.rc gadgets.ico
     $(RC) $(RCVARS) gadgets.rc
```

GADGETS.H

```
// Resource definitions.
#define IDM_FILE_NEW                    100    // -- Menu Commands --
#define IDM_FILE_OPEN                   101
#define IDM_FILE_SAVE                   102
#define IDM_FILE_SAVEAS                 103
#define IDM_FILE_PRINT                  104
```

```
#define IDM_FILE_PREVIEW            105
#define IDM_FILE_EXIT               106
#define IDM_EDIT_UNDO               200
#define IDM_EDIT_CUT                201
#define IDM_EDIT_COPY               202
#define IDM_EDIT_PASTE              203
#define IDM_EDIT_PROP               204
#define IDM_TB_HELP                 250
#define IDM_TB_DELETE               251
#define IDM_IGNORESIZE              300
#define IDM_STRINGS                 301
#define IDM_LARGEICONS              302
#define IDM_SMALLICONS              303
#define IDM_NODIVIDER               400
#define IDM_WRAPABLE                401
#define IDM_TOP                     402
#define IDM_BOTTOM                  403
#define IDM_NOMOVEY                 404
#define IDM_NOPARENTALIGN           405
#define IDM_NORESIZE                406
#define IDM_ADJUSTABLE              407
#define IDM_ALTDRAG                 408
#define IDM_TOOLTIPS                409
#define IDM_TB_CHECK                500
#define IDM_TB_ENABLE               501
#define IDM_TB_HIDE                 502
#define IDM_TB_INDETERMINATE        503
#define IDM_TB_PRESS                504
#define IDM_TB_BUTTONCOUNT          505
#define IDM_TB_GETROWS              506
#define IDM_TB_CUSTOMIZE            507
#define IDM_STAT_IGNORESIZE         600
#define IDM_STAT_SIZEGRIP           700
#define IDM_STAT_TOP                701
#define IDM_STAT_BOTTOM             702
#define IDM_STAT_NOMOVEY            703
#define IDM_STAT_NOPARENTALIGN      704
#define IDM_STAT_NORESIZE           705
#define IDM_ST_GETBORDERS           800
#define IDM_ST_GETPARTS             801
#define IDM_ST_SETTEXT              802
#define IDM_ST_SIMPLE               803
#define IDM_NOTIFICATIONS_CLEAR     900
#define IDM_VIEW_TOOLBAR            1000
#define IDM_VIEW_STATUS             1001
#define IDM_VIEW_NOTIFICATIONS      1002
#define IDM_COMBOBOX                4000
#define IDI_APP                     101    // -- Icons --
#define IDS_MAIN_MENU               71     // -- Strings --
#define IDS_MAIN_MENU1              72
#define IDS_MAIN_MENU2              73
#define IDS_MAIN_MENU3              74
#define IDS_MAIN_MENU4              75
#define IDS_MAIN_MENU5              76
#define IDS_TOOLBAR_MENU            80
#define IDS_TOOLBAR_MENU1           81
#define IDS_TOOLBAR_MENU2           82
#define IDS_TOOLBAR_MENU3           83
#define IDS_STATUSBAR_MENU          90
#define IDS_STATUSBAR_MENU1         91
#define IDS_STATUSBAR_MENU2         92
#define IDS_STATUSBAR_MENU3         93

#define IDC_TB_COMBOBOX             2000  // -- Toolbar combo box
```

```
// Toolbar functions.
HWND InitToolBar(HWND hwndParent);
HWND RebuildToolBar(HWND hwndParent, WORD wFlag);
void ToolBarMessage(HWND hwndTB, WORD wMsg);
LRESULT ToolBarNotify(HWND hwnd, WPARAM wParam, LPARAM lParam);

// Tooltip functions.
BOOL InitToolTip(HWND hwndToolBar, HWND hwndComboBox);
BOOL RelayToolTipMessage(LPMSG pMsg);
void CopyToolTipText(LPTOOLTIPTEXT lpttt);

// Status bar functions.
HWND InitStatusBar(HWND hwndParent);
HWND RebuildStatusBar(HWND hwndParent, WORD wFlag);
void StatusBarMessage(HWND hwndSB, WORD wMsg);
LRESULT Statusbar_MenuSelect(HWND, WPARAM, LPARAM);

// Notification window functions.
HWND ViewNotificationsToggle(HWND hwnd);
void DisplayNotificationDetails(WPARAM wParam, LPARAM lParam);
void ClearNotificationList();
```

NOTIFDEF.H

```
typedef struct tagCONTROLNOTIFICATIONS
{
UINT  nCode;
LPSTR pName;
} CONTROLNOTIFICATIONS;

BOOL QueryNotifyText(UINT nNotifyCode, LPSTR *pName);
void DisplayText(LPSTR pText);
```

GADGETS.C

```
/*-----------------------------------------
   GADGETS.C -- Gadgets for a frame window
                (c) Paul Yao, 1996
  -----------------------------------------*/
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include "comcthlp.h"
#include "gadgets.h"

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

char szAppName[]  = "Gadgets";
BOOL bIgnoreSize = FALSE;
HINSTANCE hInst;
HWND hwndClient = NULL;
HWND hwndToolBar = NULL;
HWND hwndStatusBar = NULL;
HWND hwndNotify = NULL;

extern DWORD dwToolBarStyles;
extern BOOL bStrings;
extern BOOL bLargeIcons;
extern BOOL bComboBox;
extern DWORD dwStatusBarStyles;
extern int cyToolBar;

//-------------------------------------------------------------------
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
```

```
                 PSTR lpszCmdLine, int cmdShow)
     {
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wc;

     hInst = hInstance;
     wc.cbSize        = sizeof(wc);
     wc.lpszClassName = szAppName;
     wc.hInstance     = hInstance;
     wc.lpfnWndProc   = WndProc;
     wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wc.hIcon         = LoadIcon(hInst, MAKEINTRESOURCE(IDI_APP));
     wc.lpszMenuName  = "MAIN";
     wc.hbrBackground =(HBRUSH)(COLOR_WINDOW + 1);
     wc.style         = 0;
     wc.cbClsExtra    = 0;
     wc.cbWndExtra    = 0;
     wc.hIconSm       = LoadIcon(hInst, MAKEINTRESOURCE(IDI_APP));

     RegisterClassEx(&wc);

     wc.lpszClassName = "ClientWndProc";
     wc.hInstance     = hInstance;
     wc.lpfnWndProc   = ClientWndProc;
     wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wc.lpszMenuName  = NULL;
     wc.hbrBackground =(HBRUSH)(COLOR_WINDOW + 1);
     wc.style         = 0;
     wc.cbClsExtra    = 0;
     wc.cbWndExtra    = 0;
     wc.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wc);

     hwnd = CreateWindowEx(0L,
                           szAppName, szAppName,
                           WS_OVERLAPPEDWINDOW,
                           CW_USEDEFAULT, CW_USEDEFAULT,
                           CW_USEDEFAULT, CW_USEDEFAULT,
                           NULL, NULL, hInstance, NULL);
     ShowWindow(hwnd, cmdShow);
     UpdateWindow(hwnd);

     InitCommonControls();

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

//-------------------------------------------------------------------
void MenuCheckMark(HMENU hmenu, int id, BOOL bCheck)
     {
     int iState;
     iState =(bCheck) ? MF_CHECKED : MF_UNCHECKED;
     CheckMenuItem(hmenu, id, iState);
     }

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

```
LRESULT CALLBACK
WndProc(HWND hwnd, UINT mMsg, WPARAM wParam, LPARAM lParam)
     {
     switch(mMsg)
          {
          case WM_CREATE :
               {
               // Create toolbar(source resides in toolbar.c)
               hwndToolBar = InitToolBar(hwnd);

               // Create status bar(source resides in statbar.c)
               hwndStatusBar = InitStatusBar(hwnd);

               // Create client window(contains notify list)
               hwndClient = CreateWindowEx(WS_EX_CLIENTEDGE,
                                 "ClientWndProc", NULL,
                                 WS_CHILD | WS_VISIBLE, 0, 0, 0, 0,
                                 hwnd,(HMENU) 4, hInst, NULL);

               return 0;
               }

          case WM_COMMAND :
               {
               // Toolbar button commands
               if(LOWORD(wParam) < 300)
                    {
                    char ach[80];
                    wsprintf(ach, "Got Command(%d)", wParam);
                    MessageBox(hwnd, ach, szAppName, MB_OK);
                    break;
                    }

               // Menu item commands
               switch(LOWORD(wParam))
                    {
                    // Toolbar settings
                    case IDM_STRINGS :
                    case IDM_LARGEICONS :
                    case IDM_SMALLICONS :
                    case IDM_NODIVIDER :
                    case IDM_WRAPABLE :
                    case IDM_TOP :
                    case IDM_BOTTOM :
                    case IDM_NOMOVEY :
                    case IDM_NOPARENTALIGN :
                    case IDM_NORESIZE :
                    case IDM_ADJUSTABLE :
                    case IDM_ALTDRAG :
                    case IDM_TOOLTIPS :
                    case IDM_COMBOBOX :
                         DestroyWindow(hwndToolBar);
                         hwndToolBar = RebuildToolBar(hwnd,
                                                      LOWORD(wParam));
                         break;

                    // Toolbar messages
                    case IDM_TB_CHECK :
                    case IDM_TB_ENABLE :
                    case IDM_TB_HIDE :
                    case IDM_TB_INDETERMINATE :
                    case IDM_TB_PRESS :
                    case IDM_TB_BUTTONCOUNT :
                    case IDM_TB_GETROWS :
```

```
case IDM_TB_CUSTOMIZE :
     ToolBarMessage(hwndToolBar, LOWORD(wParam));
     break;

// Status bar settings
case IDM_STAT_SIZEGRIP :
case IDM_STAT_TOP :
case IDM_STAT_BOTTOM :
case IDM_STAT_NOMOVEY :
case IDM_STAT_NOPARENTALIGN :
case IDM_STAT_NORESIZE :
     DestroyWindow(hwndStatusBar);
     hwndStatusBar = RebuildStatusBar(hwnd,
                                  LOWORD(wParam));
     break;

// Status bar messages
case IDM_ST_GETBORDERS :
case IDM_ST_GETPARTS :
case IDM_ST_SETTEXT :
case IDM_ST_SIMPLE :
     StatusBarMessage(hwndStatusBar, LOWORD(wParam));
     break;

// Toggle display of toolbar
case IDM_VIEW_TOOLBAR :
     {
     RECT r;

     if(hwndToolBar && IsWindowVisible(hwndToolBar))
          {
          ShowWindow(hwndToolBar, SW_HIDE);
          }
     else
          {
          ShowWindow(hwndToolBar, SW_SHOW);
          }

     // Resize other windows
     GetClientRect(hwnd, &r);
     PostMessage(hwnd, WM_SIZE, 0,
                 MAKELPARAM(r.right, r.bottom));

     break;
     }

// Toggle display of status bar
case IDM_VIEW_STATUS :
     {
     RECT r;

     if(hwndStatusBar && IsWindowVisible(hwndStatusBar))
          {
          ShowWindow(hwndStatusBar, SW_HIDE);
          }
     else
          {
          ShowWindow(hwndStatusBar, SW_SHOW);
          }

     // Resize other windows
     GetClientRect(hwnd, &r);
     PostMessage(hwnd, WM_SIZE, 0,
                 MAKELPARAM(r.right, r.bottom));
```

```
                break;
                }

            // Toggle display of notifications window
            case IDM_VIEW_NOTIFICATIONS :
                hwndNotify = ViewNotificationsToggle(hwndClient);
                break;

            // Toggle ignore WM_SIZE to show auto-size/auto-move
            case IDM_IGNORESIZE :
            case IDM_STAT_IGNORESIZE :
                {
                RECT r;

                bIgnoreSize = !bIgnoreSize;
                if(bIgnoreSize)
                    {
                    ShowWindow(hwndClient, SW_HIDE);
                    }
                else
                    {
                    ShowWindow(hwndClient, SW_SHOW);
                    GetClientRect(hwnd, &r);
                    PostMessage(hwnd, WM_SIZE, 0,
                                MAKELPARAM(r.right, r.bottom));
                    }
                break;
                }

            // Clear contents of notification window
            case IDM_NOTIFICATIONS_CLEAR :
                ClearNotificationList();
                break;
            }

        return 0;
        }

    case WM_INITMENU :
        {
        BOOL bCheck;
        HMENU hmenu =(HMENU) wParam;

        MenuCheckMark(hmenu, IDM_IGNORESIZE, bIgnoreSize);
        MenuCheckMark(hmenu, IDM_STAT_IGNORESIZE, bIgnoreSize);

        // Toolbar menu items
        MenuCheckMark(hmenu, IDM_STRINGS, bStrings);
        MenuCheckMark(hmenu, IDM_LARGEICONS, bLargeIcons);
        MenuCheckMark(hmenu, IDM_SMALLICONS, !bLargeIcons);
        MenuCheckMark(hmenu, IDM_COMBOBOX, bComboBox);

        bCheck =(dwToolBarStyles & CCS_NODIVIDER);
        MenuCheckMark(hmenu, IDM_NODIVIDER, bCheck);

        bCheck =(dwToolBarStyles & TBSTYLE_WRAPABLE);
        MenuCheckMark(hmenu, IDM_WRAPABLE, bCheck);

        bCheck =((dwToolBarStyles & 3) == CCS_TOP);
        MenuCheckMark(hmenu, IDM_TOP, bCheck);

        bCheck =((dwToolBarStyles & 3) == CCS_BOTTOM);
        MenuCheckMark(hmenu, IDM_BOTTOM, bCheck);
```

```
          bCheck =((dwToolBarStyles & 3) == CCS_NOMOVEY);
          MenuCheckMark(hmenu, IDM_NOMOVEY, bCheck);

          bCheck =(dwToolBarStyles & CCS_NOPARENTALIGN);
          MenuCheckMark(hmenu, IDM_NOPARENTALIGN, bCheck);

          bCheck =(dwToolBarStyles & CCS_NORESIZE);
          MenuCheckMark(hmenu, IDM_NORESIZE, bCheck);

          bCheck =(dwToolBarStyles & CCS_ADJUSTABLE);
          MenuCheckMark(hmenu, IDM_ADJUSTABLE, bCheck);

          bCheck =(dwToolBarStyles & TBSTYLE_ALTDRAG);
          MenuCheckMark(hmenu, IDM_ALTDRAG, bCheck);

          bCheck =(dwToolBarStyles & TBSTYLE_TOOLTIPS);
          MenuCheckMark(hmenu, IDM_TOOLTIPS, bCheck);

          // Status bar menu items
          bCheck =(dwStatusBarStyles & SBARS_SIZEGRIP);
          MenuCheckMark(hmenu, IDM_STAT_SIZEGRIP, bCheck);

          bCheck =((dwStatusBarStyles & 3) == CCS_TOP);
          MenuCheckMark(hmenu, IDM_STAT_TOP, bCheck);

          bCheck =((dwStatusBarStyles & 3) == CCS_BOTTOM);
          MenuCheckMark(hmenu, IDM_STAT_BOTTOM, bCheck);

          bCheck =((dwStatusBarStyles & 3) == CCS_NOMOVEY);
          MenuCheckMark(hmenu, IDM_STAT_NOMOVEY, bCheck);

          bCheck =(dwStatusBarStyles & CCS_NOPARENTALIGN);
          MenuCheckMark(hmenu, IDM_STAT_NOPARENTALIGN, bCheck);

          bCheck =(dwStatusBarStyles & CCS_NORESIZE);
          MenuCheckMark(hmenu, IDM_STAT_NORESIZE, bCheck);

          // View menu items
          bCheck = IsWindowVisible(hwndToolBar);
          MenuCheckMark(hmenu, IDM_VIEW_TOOLBAR, bCheck);

          bCheck = IsWindowVisible(hwndStatusBar);
          MenuCheckMark(hmenu, IDM_VIEW_STATUS, bCheck);

          bCheck =(hwndNotify != NULL);
          MenuCheckMark(hmenu, IDM_VIEW_NOTIFICATIONS, bCheck);
          return 0;
          }

     case WM_MENUSELECT :
          return Statusbar_MenuSelect(hwnd, wParam, lParam);
     case WM_DESTROY :
          PostQuitMessage(0);
          return 0;

     case WM_NOTIFY :
          {
          LPNMHDR pnmh =(LPNMHDR) lParam;
          int idCtrl =(int) wParam;

          // Display notification details in notify window
          DisplayNotificationDetails(wParam, lParam);
```

```
        // Toolbar notifications
        if((pnmh->code >= TBN_LAST) &&
           (pnmh->code <= TBN_FIRST))
             {
             return ToolBarNotify(hwnd, wParam, lParam);
             }

        // Fetch tooltip text
        if(pnmh->code == TTN_NEEDTEXT)
             {
             LPTOOLTIPTEXT lpttt =(LPTOOLTIPTEXT) lParam;
             CopyToolTipText(lpttt);
             }

        return 0;
        }

   case WM_SIZE :
        {
        int cx = LOWORD(lParam);
        int cy = HIWORD(lParam);
        int cyStatus;
        int cyTB;
        int x, y;
        DWORD dwStyle;
        RECT rWindow;

        // Ignore size message to allow auto-move and auto-size
        // features to be more clearly seen
        if(bIgnoreSize)
             return 0;

        // Adjust toolbar size
        if(IsWindowVisible(hwndToolBar))
             {
             dwStyle = GetWindowLong(hwndToolBar, GWL_STYLE);
             if(dwStyle & CCS_NORESIZE)
                  {
                  MoveWindow(hwndToolBar,
                             0, 0, cx, cyToolBar, FALSE);
                  }
             else
                  {
                  ToolBar_AutoSize(hwndToolBar);
                  }
             InvalidateRect(hwndToolBar, NULL, TRUE);
             GetWindowRect(hwndToolBar, &rWindow);
             ScreenToClient(hwnd,(LPPOINT) &rWindow.left);
             ScreenToClient(hwnd,(LPPOINT) &rWindow.right);
             cyTB = rWindow.bottom - rWindow.top;
             }
        else
             {
             cyTB = 0;
             }

        // Adjust status bar size
        if(IsWindowVisible(hwndStatusBar))
             {
             GetWindowRect(hwndStatusBar, &rWindow);
             cyStatus = rWindow.bottom - rWindow.top;
             MoveWindow(hwndStatusBar, 0, cy - cyStatus,
                                      cx, cyStatus, TRUE);
             }
```

```
            else
                {
                cyStatus = 0;
                }

            // Adjust client window size
            x = 0;
            y = cyTB;
            cy = cy -(cyStatus + cyTB);
            MoveWindow(hwndClient, x, y, cx, cy, TRUE);
            return 0;
            }

        default:
            return(DefWindowProc(hwnd, mMsg, wParam, lParam));
        }
    }

//-------------------------------------------------------------------
LRESULT CALLBACK
ClientWndProc(HWND hwnd, UINT mMsg, WPARAM wParam, LPARAM lParam)
    {
    static COLORREF crBack;
    static HBRUSH hbr;
    switch(mMsg)
        {
        case WM_CREATE :
            hwndNotify = ViewNotificationsToggle(hwnd);
            crBack = GetSysColor(COLOR_APPWORKSPACE);
            hbr = CreateSolidBrush(crBack);
            return 0;

        case WM_DESTROY :
            DeleteObject(hbr);
            return 0;

        case WM_CTLCOLORLISTBOX :
            {
            DefWindowProc(hwnd, mMsg, wParam, lParam);
            SetBkColor((HDC) wParam, crBack);
            SetBkMode((HDC) wParam, TRANSPARENT);
            return(LRESULT)(HBRUSH) hbr;
            }

        case WM_SIZE :
            {
            HWND hwndNotify = GetWindow(hwnd, GW_CHILD);
            int cx = LOWORD(lParam);
            int cy = HIWORD(lParam);

            // Ignore if notification window is absent
            if(hwndNotify != NULL)
                {
                MoveWindow(hwndNotify, 0, 0, cx, cy, TRUE);
                }

            return 0;
            }

        default :
            return(DefWindowProc(hwnd, mMsg, wParam, lParam));
        }
    }
```

TOOLBAR.C

```
/*---------------------------------------
   TOOLBAR.C -- Toolbar helper functions
               (c) Paul Yao, 1996
  ---------------------------------------*/
#include <windows.h>
#include <commctrl.h>
#include "comcthlp.h"
#include "gadgets.h"
#include "notifdef.h"
HWND hwndCombo;
HWND hwndEdit;
HWND hwndToolTip;
HWND hwndTB;

int cyToolBar;

BOOL bComboBox   = FALSE;
BOOL bStrings    = FALSE;
BOOL bLargeIcons = FALSE;

DWORD dwToolBarStyles = WS_CHILD | WS_VISIBLE | WS_CLIPSIBLINGS |
                        CCS_TOP | CCS_NODIVIDER | TBSTYLE_TOOLTIPS;

extern HINSTANCE hInst;

char szTbStrings[] = "New\0Open\0Save\0Print\0Preview\0"
                     "Cut\0Copy\0Paste\0Undo\0Properties\0"
                     "Help\0Delete\0";

TBBUTTON tbb[] =
     {
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     STD_FILENEW, IDM_FILE_NEW, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 0,
     STD_FILEOPEN, IDM_FILE_OPEN, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 1,
     STD_FILESAVE, IDM_FILE_SAVE, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 2,
     STD_PRINT, IDM_FILE_PRINT, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 3,
     STD_PRINTPRE, IDM_FILE_PREVIEW, TBSTATE_ENABLED, TBSTYLE_CHECK, 0, 0, 0, 4,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     STD_CUT, IDM_EDIT_CUT, TBSTATE_ENABLED, TBSTYLE_CHECKGROUP, 0, 0, 0, 5,
     STD_COPY, IDM_EDIT_COPY, TBSTATE_ENABLED, TBSTYLE_CHECKGROUP, 0, 0, 0, 6,
     STD_PASTE, IDM_EDIT_PASTE, TBSTATE_ENABLED, TBSTYLE_CHECKGROUP, 0, 0, 0, 7,
     STD_UNDO, IDM_EDIT_UNDO, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 8,
```

```
     STD_PROPERTIES, IDM_EDIT_PROP, TBSTATE_ENABLED, TBSTYLE_CHECK, 0, 0, 0, 9,
     STD_HELP, IDM_TB_HELP, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 10,
     STD_DELETE, IDM_TB_DELETE, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 11,
     };

int nCust[] = { 21, 22, 23, 24, 25, 27, 28, 29, 30, 31, 32, 33, -1};

TBADDBITMAP tbStdLarge[] =
     {
     HINST_COMMCTRL, IDB_STD_LARGE_COLOR,
     };

TBADDBITMAP tbStdSmall[] =
     {
     HINST_COMMCTRL, IDB_STD_SMALL_COLOR,
     };

//-------------------------------------------------------------------
LPSTR GetString(int iString)
     {
     int i, cb;
     LPSTR pString;

     // Cycle through to requested string
     pString = szTbStrings;
     for(i = 0; i < iString; i++)
          {
          cb = lstrlen(pString);
          pString +=(cb + 1);
          }

     return pString;
     }
//-------------------------------------------------------------------
LRESULT ToolBarNotify(HWND hwnd, WPARAM wParam, LPARAM lParam)
     {
     LPNMHDR pnmh =(LPNMHDR) lParam;
     int idCtrl =(int) wParam;

     // Allow toolbar to be customized
     if((pnmh->code == TBN_QUERYDELETE) ||
        (pnmh->code == TBN_QUERYINSERT))
          {
          return 1; // We always say "yes"
          }

     // Provide details of allowable toolbar buttons
     if(pnmh->code == TBN_GETBUTTONINFO)
          {
          LPTBNOTIFY ptbn =(LPTBNOTIFY) lParam;
          int iButton = nCust[ptbn->iItem];

          if(iButton != -1)
               {
               lstrcpy(ptbn->pszText, GetString(ptbn->iItem));
               memcpy(&ptbn->tbButton, &tbb[iButton], sizeof(TBBUTTON));
               return 1;
               }
          }

     return 0;
     }

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

```
HWND InitToolBar(HWND hwndParent)
     {
     int  iNumButtons;
     LPTBBUTTON ptbb;

     if(bComboBox)
          {
          ptbb = &tbb[0];
          iNumButtons = 31;
          }
     else
          {
          ptbb = &tbb[21];
          iNumButtons = 10;
          }

     UINT uiBitmap =(bLargeIcons) ? IDB_STD_LARGE_COLOR :
                                    IDB_STD_SMALL_COLOR;
     hwndTB = CreateToolbarEx(hwndParent,
                   dwToolBarStyles,
                   1, 15,
                   HINST_COMMCTRL,
                   uiBitmap,
                   ptbb,
                   iNumButtons,
                   0, 0, 0, 0,
                   sizeof(TBBUTTON));

     // If requested, add to string list
     if(bStrings)
          ToolBar_AddString(hwndTB, 0, szTbStrings);

     // Store handle to tooltip control
     hwndToolTip = ToolBar_GetToolTips(hwndTB);

     // Insert combo box into toolbar
     if(bComboBox)
          {
          RECT r;
          int x, y, cx, cy;

          // Calculate coordinates for combo box
          ToolBar_GetItemRect(hwndTB, 0, &r);
          x = r.left;
          y = r.top;
          cy = 100;
          ToolBar_GetItemRect(hwndTB, 18, &r);
          cx = r.right - x + 1;

          hwndCombo = CreateWindow("combobox",
                           NULL,
                           WS_CHILD | WS_VISIBLE |
                           CBS_DROPDOWN,
                           x, y, cx, cy,
                           hwndParent,
                          (HMENU) IDC_TB_COMBOBOX,
                           hInst,
                           0);

          // Set toolbar as combo box window parent
          SetParent(hwndCombo, hwndTB);

          SendMessage(hwndCombo, CB_ADDSTRING, 0,(LPARAM) "One");
          SendMessage(hwndCombo, CB_ADDSTRING, 0,(LPARAM) "Two");
```

```
          SendMessage(hwndCombo, CB_ADDSTRING, 0,(LPARAM) "Three");

          // Calculate toolbar height
          GetWindowRect(hwndCombo, &r);
          cyToolBar = r.bottom - r.top + 1;
          cyToolBar += y;
          cyToolBar +=(2 * GetSystemMetrics(SM_CYBORDER));
          ToolBar_GetItemRect(hwndTB, 0, &r);
          cyToolBar = max(cyToolBar, r.bottom+5);

          // If toolbar has tooltips, initialize
          if(dwToolBarStyles & TBSTYLE_TOOLTIPS)
               InitToolTip(hwndTB, hwndCombo);
          }

     return hwndTB;
     }

//-------------------------------------------------------------------
void static FlipStyleFlag(LPDWORD dwStyle, DWORD flag)
     {
     if(*dwStyle & flag)  // Flag on -- turn off
          {
          *dwStyle &=(~flag);
          }
     else                 // Flag off -- turn on
          {
          *dwStyle |= flag;
          }
     }

//-------------------------------------------------------------------
HWND RebuildToolBar(HWND hwndParent, WORD wFlag)
     {
     HWND hwndTB;
     RECT r;

     switch(wFlag)
          {
          case IDM_STRINGS :
               bStrings = !bStrings;
               break;

          case IDM_LARGEICONS :
               bLargeIcons = TRUE;
               break;

          case IDM_SMALLICONS :
               bLargeIcons = FALSE;
               break;

          case IDM_NODIVIDER :
               FlipStyleFlag(&dwToolBarStyles, CCS_NODIVIDER);
               break;

          case IDM_WRAPABLE :
               FlipStyleFlag(&dwToolBarStyles, TBSTYLE_WRAPABLE);
               break;

          case IDM_TOP :
               dwToolBarStyles &= 0xFFFFFFFC;
               dwToolBarStyles |= CCS_TOP;
               break;
```

```
          case IDM_BOTTOM :
               dwToolBarStyles &= 0xFFFFFFFC;
               dwToolBarStyles |= CCS_BOTTOM;
               break;

          case IDM_NOMOVEY :
               dwToolBarStyles &= 0xFFFFFFFC;
               dwToolBarStyles |= CCS_NOMOVEY;
               break;

          case IDM_NOPARENTALIGN :
               FlipStyleFlag(&dwToolBarStyles, CCS_NOPARENTALIGN);
               break;

          case IDM_NORESIZE :
               FlipStyleFlag(&dwToolBarStyles, CCS_NORESIZE);
               break;

          case IDM_ADJUSTABLE :
               FlipStyleFlag(&dwToolBarStyles, CCS_ADJUSTABLE);
               break;

          case IDM_ALTDRAG :
               FlipStyleFlag(&dwToolBarStyles, TBSTYLE_ALTDRAG);
               break;

          case IDM_TOOLTIPS :
               FlipStyleFlag(&dwToolBarStyles, TBSTYLE_TOOLTIPS);
               break;

          case IDM_COMBOBOX :
               bComboBox =(!bComboBox);
          }

     hwndTB = InitToolBar(hwndParent);

     // Post parent a WM_SIZE message to resize children
     GetClientRect(hwndParent, &r);
     PostMessage(hwndParent, WM_SIZE, 0,
                 MAKELPARAM(r.right, r.bottom));

     return hwndTB;
     }
//--------------------------------------------------------------------
void ToolBarMessage(HWND hwndTB, WORD wMsg)
     {
     switch(wMsg)
          {
          case IDM_TB_CHECK :
               {
               int nState = ToolBar_GetState(hwndTB, 1);
               BOOL bCheck =(!(nState & TBSTATE_CHECKED));
               ToolBar_CheckButton(hwndTB, 1, bCheck );
               break;
               }

          case IDM_TB_ENABLE :
               {
               int nState = ToolBar_GetState(hwndTB, 2);
               BOOL bEnabled =(!(nState & TBSTATE_ENABLED));
               ToolBar_EnableButton(hwndTB, 2, bEnabled);
               break;
               }
```

```
          case IDM_TB_HIDE :
               {
               int nState = ToolBar_GetState(hwndTB, 3);
               BOOL bShow =(!(nState & TBSTATE_HIDDEN));
               ToolBar_HideButton(hwndTB, 3, bShow);
               break;
               }

          case IDM_TB_INDETERMINATE :
               {
               int nState = ToolBar_GetState(hwndTB, 4);
               BOOL bInd =(!(nState & TBSTATE_INDETERMINATE));
               ToolBar_Indeterminate(hwndTB, 4, bInd);
               break;
               }

          case IDM_TB_PRESS :
               {
               int nState = ToolBar_GetState(hwndTB, 5);
               BOOL bPress =(!(nState & TBSTATE_PRESSED));
               ToolBar_PressButton(hwndTB, 5, bPress);
               break;
               }

          case IDM_TB_BUTTONCOUNT :
               {
               int nButtons = ToolBar_ButtonCount(hwndTB);
               char ach[80];
               wsprintf(ach, "Button Count = %d", nButtons);
               MessageBox(GetParent(hwndTB), ach,
                           "TB_BUTTONCOUNT", MB_OK);
               break;
               }

          case IDM_TB_GETROWS :
               {
               int nRows = ToolBar_GetRows(hwndTB);
               char ach[80];
               wsprintf(ach, "Row Count = %d", nRows);
               MessageBox(GetParent(hwndTB), ach,
                           "TB_GETROWS", MB_OK);
               break;
               }

          case IDM_TB_CUSTOMIZE :
               // ToolBar_Customize(hwndTB);
               SendMessage(hwndTB, TB_CUSTOMIZE,(LPARAM) &tbb[25], 5);
               break;
          }
     }
```

TOOLTIP.C

```
/*-----------------------------------------
   TOOLTIP.C -- Tooltip helper functions
               (c) Paul Yao, 1996
  -----------------------------------------*/
#include <windows.h>
#include <commctrl.h>
#include "comcthlp.h"
#include "notifdef.h"
#include "gadgets.h"

extern BOOL bComboBox;
extern char szTbStrings[];
```

```
extern HINSTANCE hInst;
extern HWND hwndEdit;
extern HWND hwndCombo;
extern HWND hwndEdit;
static HWND hwndTT;

// Map toolbar button command to string index
int CommandToString[] =
     { IDM_FILE_NEW, IDM_FILE_OPEN, IDM_FILE_SAVE, IDM_FILE_PRINT,
       IDM_FILE_PREVIEW, IDM_EDIT_CUT, IDM_EDIT_COPY, IDM_EDIT_PASTE,
       IDM_EDIT_UNDO, IDM_EDIT_PROP, IDM_TB_HELP, IDM_TB_DELETE, -1
     };

//-------------------------------------------------------------------
BOOL InitToolTip(HWND hwndToolBar, HWND hwndComboBox)
     {
     BOOL bSuccess;
     TOOLINFO ti;

     // Fetch handle to tooltip control
     hwndTT = ToolBar_GetToolTips(hwndToolBar);
     if(hwndTT == NULL)
          return FALSE;

     // Add tooltip for main combo box
     ZeroMemory(&ti, sizeof(TOOLINFO));
     ti.cbSize = sizeof(TOOLINFO);
     ti.uFlags = TTF_IDISHWND | TTF_CENTERTIP | TTF_SUBCLASS;
     ti.hwnd   = hwndToolBar;
     ti.uId    =(UINT)(HWND) hwndComboBox;
     ti.lpszText = LPSTR_TEXTCALLBACK;
     bSuccess = ToolTip_AddTool(hwndTT, &ti);
     if(!bSuccess)
          return FALSE;

     // Add tooltip for combo box's edit control
     hwndEdit = GetWindow(hwndComboBox, GW_CHILD);
     ti.uId    =(UINT)(HWND) hwndEdit;
     bSuccess = ToolTip_AddTool(hwndTT, &ti);

     return bSuccess;
     }

//-------------------------------------------------------------------
void CopyToolTipText(LPTOOLTIPTEXT lpttt)
     {
     int i;
     int iButton = lpttt->hdr.idFrom;
     int cb;
     int cMax;
     LPSTR pString;
     LPSTR pDest = lpttt->lpszText;

     // Check for combo box window handles
     if(lpttt->uFlags & TTF_IDISHWND)
          {
          if((iButton ==(int) hwndCombo) ||
             (iButton ==(int) hwndEdit))
               {
               lstrcpy(pDest, "1-2-3 ComboBox");
               return;
               }
          }
```

```
     // Map command ID to string index
     for(i = 0; CommandToString[i] != -1; i++)
          {
          if(CommandToString[i] == iButton)
               {
               iButton = i;
               break;
               }
          }

     // To be safe, count number of strings in text
     pString = szTbStrings;
     cMax = 0;
     while(*pString != '\0')
          {
          cMax++;
          cb = lstrlen(pString);
          pString +=(cb + 1);
          }

     // Check for valid parameter
     if(iButton > cMax)
          {
          pString = "Invalid Button Index";
          }
     else
          {
          // Cycle through to requested string
          pString = szTbStrings;
          for(i = 0; i < iButton; i++)
               {
               cb = lstrlen(pString);
               pString +=(cb + 1);
               }
          }

     lstrcpy(pDest, pString);
     }
```

STATBAR.C

```
/*-------------------------------------------
   STATBAR.C -- Status bar helper functions
               (c) Paul Yao, 1996
  -------------------------------------------*/
#include <windows.h>
#include <commctrl.h>
#include "comcthlp.h"
#include "gadgets.h"

typedef struct tagPOPUPSTRING
     {
     HMENU hMenu;
     UINT uiString;
     } POPUPSTRING;

#define MAX_MENUS 5

static POPUPSTRING popstr[MAX_MENUS];

DWORD dwStatusBarStyles = WS_CHILD | WS_VISIBLE |
                          WS_CLIPSIBLINGS | CCS_BOTTOM |
                          SBARS_SIZEGRIP;
extern HINSTANCE hInst;
extern HWND hwndStatusBar;
```

```
//-----------------------------------------------------------------------
HWND InitStatusBar(HWND hwndParent)
     {
     HWND hwndSB;

     // Initialize values for WM_MENUSELECT message handling
     HMENU hMenu = GetMenu(hwndParent);
     HMENU hMenuTB = GetSubMenu(hMenu, 2);
     HMENU hMenuSB = GetSubMenu(hMenu, 3);
     popstr[0].hMenu    = 0;
     popstr[0].uiString = 0;
     popstr[1].hMenu    = hMenu;
     popstr[1].uiString = IDS_MAIN_MENU;
     popstr[2].hMenu    = hMenuTB;
     popstr[2].uiString = IDS_TOOLBAR_MENU;
     popstr[3].hMenu    = hMenuSB;
     popstr[3].uiString = IDS_STATUSBAR_MENU;
     popstr[4].hMenu    = 0;
     popstr[4].uiString = 0;

     hwndSB = CreateStatusWindow(dwStatusBarStyles,
                                 "Ready",
                                 hwndParent,
                                 2);
     return hwndSB;
     }

//-----------------------------------------------------------------------
void static FlipStyleFlag(LPDWORD dwStyle, DWORD flag)
     {
     if(*dwStyle & flag)  // Flag on -- turn off
          {
          *dwStyle &=(~flag);
          }
     else                  // Flag off -- turn on
          {
          *dwStyle |= flag;
          }
     }
//-----------------------------------------------------------------------
HWND RebuildStatusBar(HWND hwndParent, WORD wFlag)
     {
     HWND hwndSB;
     RECT r;

     switch(wFlag)
          {
          case IDM_STAT_SIZEGRIP :
               FlipStyleFlag(&dwStatusBarStyles, SBARS_SIZEGRIP);
               break;

          case IDM_STAT_TOP :
               dwStatusBarStyles &= 0xFFFFFFFC;
               dwStatusBarStyles |= CCS_TOP;
               break;

          case IDM_STAT_BOTTOM :
               dwStatusBarStyles &= 0xFFFFFFFC;
               dwStatusBarStyles |= CCS_BOTTOM;
               break;

          case IDM_STAT_NOMOVEY :
               dwStatusBarStyles &= 0xFFFFFFFC;
```

```
              dwStatusBarStyles |= CCS_NOMOVEY;
              break;

         case IDM_STAT_NOPARENTALIGN :
              FlipStyleFlag(&dwStatusBarStyles, CCS_NOPARENTALIGN);
              break;

         case IDM_STAT_NORESIZE :
              FlipStyleFlag(&dwStatusBarStyles, CCS_NORESIZE);
              break;
         }

     hwndSB = InitStatusBar(hwndParent);

     // Post parent a WM_SIZE message to resize children
     GetClientRect(hwndParent, &r);
     PostMessage(hwndParent, WM_SIZE, 0,
                 MAKELPARAM(r.right, r.bottom));

     return hwndSB;
     }

//-------------------------------------------------------------------
void StatusBarMessage(HWND hwndSB, WORD wMsg)
     {
     switch(wMsg)
          {
          case IDM_ST_GETBORDERS :
               {
               char ach[180];
               int aiBorders[3];

               Status_GetBorders(hwndSB, &aiBorders);
               wsprintf(ach, "Horiz Width = %d\n"
                          "Vert Width = %d\n"
                          "Separator Width = %d",
                          aiBorders[0], aiBorders[1],
                          aiBorders[2]);
               MessageBox(GetParent(hwndSB), ach,
                            "SB_GETBORDERS", MB_OK);
               break;
               }

          case IDM_ST_GETPARTS :
               {
               char ach[80];
               int nParts = Status_GetParts(hwndSB, 0, 0);
               wsprintf(ach, "Part Count = %d", nParts);
               MessageBox(GetParent(hwndSB), ach,
                            "SB_GETPARTS", MB_OK);
               break;
               }

          case IDM_ST_SETTEXT :
               Status_SetText(hwndSB, 0, 0,
                                "SB_SETTEXT Message Sent");
               break;

          case IDM_ST_SIMPLE :
               {
               static BOOL bSimple = TRUE;
               Status_Simple(hwndSB, bSimple);
               bSimple =(!bSimple);
               break;
```

```
            }
        }
    }

//-------------------------------------------------------------------
LRESULT
Statusbar_MenuSelect(HWND hwnd, WPARAM wParam, LPARAM lParam)
    {
    UINT fuFlags =(UINT) HIWORD(wParam);
    HMENU hMainMenu = NULL;
    int iMenu = 0;
    // Handle non-system popup menu descriptions
    if((fuFlags & MF_POPUP) &&
       (!(fuFlags & MF_SYSMENU)))
        {
        for(iMenu = 1; iMenu < MAX_MENUS; iMenu++)
            {
            if((HMENU) lParam == popstr[iMenu].hMenu)
                {
                hMainMenu =(HMENU) lParam;
                break;
                }
            }
        }

    // Display helpful text in status bar
    MenuHelp(WM_MENUSELECT, wParam, lParam, hMainMenu, hInst,
            hwndStatusBar, &((UINT) popstr[iMenu].hMenu));

    return 0;
    }
```

NOTIFDEF.C

```
/*--------------------------------------------------
   NOTIFDEF.C -- Support notification detail window
                 (c) Paul Yao, 1996
  --------------------------------------------------*/
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include <prsht.h>
#include "notifdef.h"

CONTROLNOTIFICATIONS cnLookupTable[] =
    {
    NM_OUTOFMEMORY,        "NM_OUTOFMEMORY",
    NM_CLICK,              "NM_CLICK",
    NM_DBLCLK,             "NM_DBLCLK",
    NM_RETURN,             "NM_RETURN",
    NM_RCLICK,             "NM_RCLICK",
    NM_RDBLCLK,            "NM_RDBLCLK",
    NM_SETFOCUS,           "NM_SETFOCUS",
    NM_KILLFOCUS,          "NM_KILLFOCUS",
    LVN_ITEMCHANGING,      "LVN_ITEMCHANGING",
    LVN_ITEMCHANGED,       "LVN_ITEMCHANGED",
    LVN_INSERTITEM,        "LVN_INSERTITEM",
    LVN_DELETEITEM,        "LVN_DELETEITEM",
    LVN_DELETEALLITEMS,    "LVN_DELETEALLITEMS",
    LVN_BEGINLABELEDITA,   "LVN_BEGINLABELEDITA",
    LVN_BEGINLABELEDITW,   "LVN_BEGINLABELEDITW",
    LVN_ENDLABELEDITA,     "LVN_ENDLABELEDITA",
    LVN_ENDLABELEDITW,     "LVN_ENDLABELEDITW",
    LVN_COLUMNCLICK,       "LVN_COLUMNCLICK",
    LVN_BEGINDRAG,         "LVN_BEGINDRAG",
```

```
LVN_BEGINRDRAG,        "LVN_BEGINRDRAG",
LVN_GETDISPINFOA,      "LVN_GETDISPINFOA",
LVN_GETDISPINFOW,      "LVN_GETDISPINFOW",
LVN_SETDISPINFOA,      "LVN_SETDISPINFOA",
LVN_SETDISPINFOW,      "LVN_SETDISPINFOW",
LVN_KEYDOWN,           "LVN_KEYDOWN",
HDN_ITEMCHANGINGA,     "HDN_ITEMCHANGINGA",
HDN_ITEMCHANGINGW,     "HDN_ITEMCHANGINGW",
HDN_ITEMCHANGEDA,      "HDN_ITEMCHANGEDA",
HDN_ITEMCHANGEDW,      "HDN_ITEMCHANGEDW",
HDN_ITEMCLICKA,        "HDN_ITEMCLICKA",
HDN_ITEMCLICKW,        "HDN_ITEMCLICKW",
HDN_ITEMDBLCLICKA,     "HDN_ITEMDBLCLICKA",
HDN_ITEMDBLCLICKW,     "HDN_ITEMDBLCLICKW",
HDN_DIVIDERDBLCLICKA, "HDN_DIVIDERDBLCLICKA",
HDN_DIVIDERDBLCLICKW, "HDN_DIVIDERDBLCLICKW",
HDN_BEGINTRACKA,       "HDN_BEGINTRACKA",
HDN_BEGINTRACKW,       "HDN_BEGINTRACKW",
HDN_ENDTRACKA,         "HDN_ENDTRACKA",
HDN_ENDTRACKW,         "HDN_ENDTRACKW",
HDN_TRACKA,            "HDN_TRACKA",
HDN_TRACKW,            "HDN_TRACKW",
TVN_SELCHANGINGA,      "TVN_SELCHANGINGA",
TVN_SELCHANGINGW,      "TVN_SELCHANGINGW",
TVN_SELCHANGEDA,       "TVN_SELCHANGEDA",
TVN_SELCHANGEDW,       "TVN_SELCHANGEDW",
TVN_GETDISPINFOA,      "TVN_GETDISPINFOA",
TVN_GETDISPINFOW,      "TVN_GETDISPINFOW",
TVN_SETDISPINFOA,      "TVN_SETDISPINFOA",
TVN_SETDISPINFOW,      "TVN_SETDISPINFOW",
TVN_ITEMEXPANDINGA,    "TVN_ITEMEXPANDINGA",
TVN_ITEMEXPANDINGW,    "TVN_ITEMEXPANDINGW",
TVN_ITEMEXPANDEDA,     "TVN_ITEMEXPANDEDA",
TVN_ITEMEXPANDEDW,     "TVN_ITEMEXPANDEDW",
TVN_BEGINDRAGA,        "TVN_BEGINDRAGA",
TVN_BEGINDRAGW,        "TVN_BEGINDRAGW",
TVN_BEGINRDRAGA,       "TVN_BEGINRDRAGA",
TVN_BEGINRDRAGW,       "TVN_BEGINRDRAGW",
TVN_DELETEITEMA,       "TVN_DELETEITEMA",
TVN_DELETEITEMW,       "TVN_DELETEITEMW",
TVN_BEGINLABELEDITA,   "TVN_BEGINLABELEDITA",
TVN_BEGINLABELEDITW,   "TVN_BEGINLABELEDITW",
TVN_ENDLABELEDITA,     "TVN_ENDLABELEDITA",
TVN_ENDLABELEDITW,     "TVN_ENDLABELEDITW",
TVN_KEYDOWN,           "TVN_KEYDOWN",
TTN_NEEDTEXTA,         "TTN_NEEDTEXTA",
TTN_NEEDTEXTW,         "TTN_NEEDTEXTW",
TTN_SHOW,              "TTN_SHOW",
TTN_POP,               "TTN_POP",
TCN_KEYDOWN,           "TCN_KEYDOWN",
TCN_SELCHANGE,         "TCN_SELCHANGE",
TCN_SELCHANGING,       "TCN_SELCHANGING",
TBN_GETBUTTONINFOA,    "TBN_GETBUTTONINFOA",
TBN_GETBUTTONINFOW,    "TBN_GETBUTTONINFOW",
TBN_BEGINDRAG,         "TBN_BEGINDRAG",
TBN_ENDDRAG,           "TBN_ENDDRAG",
TBN_BEGINADJUST,       "TBN_BEGINADJUST",
TBN_ENDADJUST,         "TBN_ENDADJUST",
TBN_RESET,             "TBN_RESET",
TBN_QUERYINSERT,       "TBN_QUERYINSERT",
TBN_QUERYDELETE,       "TBN_QUERYDELETE",
TBN_TOOLBARCHANGE,     "TBN_TOOLBARCHANGE",
TBN_CUSTHELP,          "TBN_CUSTHELP",
UDN_DELTAPOS,          "UDN_DELTAPOS",
```

```
     PSN_SETACTIVE,        "PSN_SETACTIVE",
     PSN_KILLACTIVE,       "PSN_KILLACTIVE",
     PSN_APPLY,            "PSN_APPLY",
     PSN_RESET,            "PSN_RESET",
     PSN_HELP,             "PSN_HELP",
     PSN_WIZBACK,          "PSN_WIZBACK",
     PSN_WIZNEXT,          "PSN_WIZNEXT",
     PSN_WIZFINISH,        "PSN_WIZFINISH",
     PSN_QUERYCANCEL,      "PSN_QUERYCANCEL"
     };

int NOTIFY_COUNT = sizeof(cnLookupTable) / sizeof(CONTROLNOTIFICATIONS);
static HWND hwndNotify = NULL;

extern HINSTANCE hInst;

//-------------------------------------------------------------------
// QueryNotifyText: Convert notification codes into text
//-------------------------------------------------------------------
BOOL QueryNotifyText(UINT nNotifyCode, LPSTR *pName)
     {
     BOOL bFound = FALSE;
     int  iNotify;

     for(iNotify = 0; iNotify < NOTIFY_COUNT; iNotify++)
          {
          if(cnLookupTable[iNotify].nCode == nNotifyCode)
               {
               *pName = cnLookupTable[iNotify].pName;
               return TRUE;
               }
          }

     // Unknown notification code
     *pName = "** Unknown **";
     return FALSE;
     }
//-------------------------------------------------------------------
// ViewNotificationsToggle: Display/hide notification window
//-------------------------------------------------------------------
HWND ViewNotificationsToggle(HWND hwnd)
     {
     int x, y, cx, cy;
     RECT rWindow;

     if(hwndNotify)
          {
          DestroyWindow(hwndNotify);
          hwndNotify = NULL;
          }
     else
          {
          GetClientRect(hwnd, &rWindow);
          x = 0;
          y = 0;
          cx = rWindow.right  - rWindow.left;
          cy = rWindow.bottom - rWindow.top;
          hwndNotify = CreateWindowEx(0L,
                          "LISTBOX", NULL,
                          LBS_NOINTEGRALHEIGHT |
                          WS_CHILD   |
                          WS_VISIBLE |
                          WS_VSCROLL,
                          x, y, cx, cy,
```

```
                         hwnd,(HMENU) 1, hInst, NULL);
          }

     return hwndNotify;
     }

//-------------------------------------------------------------------
void DisplayText(LPSTR pText)
     {
     int iIndex;

     if(hwndNotify == NULL)
          return;

     iIndex = ListBox_AddString(hwndNotify, pText);
     ListBox_SetTopIndex(hwndNotify, iIndex);
     }

//-------------------------------------------------------------------
void DisplayNotificationDetails(WPARAM wParam, LPARAM lParam)
     {
     LPNMHDR pnmh;
     LPSTR   pName;
     if(hwndNotify == NULL)
          return;

     pnmh =(LPNMHDR) lParam;
     QueryNotifyText(pnmh->code, &pName);
     DisplayText(pName);
     }

//-------------------------------------------------------------------
void ClearNotificationList()
     {
     if(hwndNotify == NULL)
          return;

     ListBox_ResetContent(hwndNotify);
     }
```

GADGETS.RC

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

IDI_APP ICON    DISCARDABLE     "gadgets.ico"

MAIN MENU DISCARDABLE
     {
     POPUP "&File"
          {
          MENUITEM "&New",                        IDM_FILE_NEW
          MENUITEM "&Open",                       IDM_FILE_OPEN
          MENUITEM "&Save",                       IDM_FILE_SAVE
          MENUITEM "Save &As...",                 IDM_FILE_SAVEAS
          MENUITEM SEPARATOR
          MENUITEM "&Print...",                   IDM_FILE_PRINT
          MENUITEM "Print Pre&view...",           IDM_FILE_PREVIEW
          MENUITEM SEPARATOR
          MENUITEM "&Exit",                       IDM_FILE_EXIT
          }
     POPUP "&Edit"
          {
          MENUITEM "&Undo\tCtrl+Z",               IDM_EDIT_UNDO
          MENUITEM SEPARATOR
```

```
        MENUITEM "&Cut\tCtrl+X",                IDM_EDIT_CUT
        MENUITEM "&Copy\tCtrl+C",               IDM_EDIT_COPY
        MENUITEM "&Paste\tCtrl+V",              IDM_EDIT_PASTE
        MENUITEM SEPARATOR
        MENUITEM "Pr&operties",                 IDM_EDIT_PROP
        }
    POPUP "&Toolbar"
        {
        POPUP "St&yles"
            {
            MENUITEM "CCS_NODIVIDER",               IDM_NODIVIDER
            MENUITEM "TBSTYLE_WRAPABLE",            IDM_WRAPABLE
            MENUITEM SEPARATOR
            MENUITEM "CCS_TOP",                     IDM_TOP
            MENUITEM "CCS_BOTTOM",                  IDM_BOTTOM
            MENUITEM "CCS_NOMOVEY",                 IDM_NOMOVEY
            MENUITEM SEPARATOR
            MENUITEM "CCS_NOPARENTALIGN",           IDM_NOPARENTALIGN
            MENUITEM "CCS_NORESIZE",                IDM_NORESIZE
            MENUITEM SEPARATOR
            MENUITEM "CCS_ADJUSTABLE",              IDM_ADJUSTABLE
            MENUITEM "TBSTYLE_ALTDRAG",             IDM_ALTDRAG
            MENUITEM SEPARATOR
            MENUITEM "TBSTYLE_TOOLTIPS",            IDM_TOOLTIPS
            }
        MENUITEM "&Ignore WM_SIZE",                 IDM_IGNORESIZE
        MENUITEM SEPARATOR
        POPUP "&Messages"
            {
            MENUITEM "TB_CHECKBUTTON",              IDM_TB_CHECK
            MENUITEM "TB_ENABLEBUTTON",             IDM_TB_ENABLE
            MENUITEM "TB_HIDEBUTTON",               IDM_TB_HIDE
            MENUITEM "TB_INDETERMINATE",            IDM_TB_INDETERMINATE
            MENUITEM "TB_PRESSBUTTON",              IDM_TB_PRESS
            MENUITEM SEPARATOR
            MENUITEM "TB_BUTTONCOUNT",              IDM_TB_BUTTONCOUNT
            MENUITEM "TB_GETROWS",                  IDM_TB_GETROWS
            MENUITEM SEPARATOR
            MENUITEM "TB_CUSTOMIZE",                IDM_TB_CUSTOMIZE
            }
        MENUITEM SEPARATOR
        MENUITEM "&Large Icons",                    IDM_LARGEICONS
        MENUITEM "&Small Icons",                    IDM_SMALLICONS
        MENUITEM SEPARATOR
        MENUITEM "S&trings",                        IDM_STRINGS
        MENUITEM "&Combo Box",                      IDM_COMBOBOX
        }
    POPUP "&Statusbar"
        {
        POPUP "&Styles"
            {
            MENUITEM "SBARS_SIZEGRIP",              IDM_STAT_SIZEGRIP
            MENUITEM SEPARATOR
            MENUITEM "CCS_TOP",                     IDM_STAT_TOP
            MENUITEM "CCS_BOTTOM",                  IDM_STAT_BOTTOM
            MENUITEM "CCS_NOMOVEY",                 IDM_STAT_NOMOVEY
            MENUITEM SEPARATOR
            MENUITEM "CCS_NOPARENTALIGN",           IDM_STAT_NOPARENTALIGN
            MENUITEM "CCS_NORESIZE",                IDM_STAT_NORESIZE
            }
        MENUITEM "&Ignore WM_SIZE",                 IDM_STAT_IGNORESIZE
        MENUITEM SEPARATOR
        POPUP "&Messages"
            {
```

```
                MENUITEM "SB_GETBORDERS",           IDM_ST_GETBORDERS
                MENUITEM "SB_GETPARTS",             IDM_ST_GETPARTS
                MENUITEM "SB_SETTEXT",              IDM_ST_SETTEXT
                MENUITEM SEPARATOR
                MENUITEM "SB_SIMPLE",               IDM_ST_SIMPLE
                }
            }
        POPUP "&Notifications"
            {
            MENUITEM "&Clear",                      IDM_NOTIFICATIONS_CLEAR
            }
        POPUP "&View"
            {
            MENUITEM "&Toolbar",                    IDM_VIEW_TOOLBAR
            MENUITEM "&Status Bar",                 IDM_VIEW_STATUS
            MENUITEM "&Notifications",              IDM_VIEW_NOTIFICATIONS
            }
        }

STRINGTABLE DISCARDABLE
        {
        IDS_MAIN_MENU           "Create, open, save, print documents or quit program"
        IDS_MAIN_MENU1          "Undo, cut, copy, paste, and properties"
        IDS_MAIN_MENU2          "Toolbar styles, messages, and creation options"
        IDS_MAIN_MENU3          "Status bar styles and messages"
        IDS_MAIN_MENU4          "Clear notifications window"
        IDS_MAIN_MENU5          "Show or hide toolbar, status bar, and notifications window"
        IDS_TOOLBAR_MENU        "Set toolbar styles and re-create toolbar"
        IDS_TOOLBAR_MENU1       "placeholder"
        IDS_TOOLBAR_MENU2       "placeholder"
        IDS_TOOLBAR_MENU3       "Send messages to toolbar"
        IDS_STATUSBAR_MENU      "Set status bar styles and re-create status bar"
        IDS_STATUSBAR_MENU1     "placeholder"
        IDS_STATUSBAR_MENU2     "placeholder"
        IDS_STATUSBAR_MENU3     "Send messages to status bar"
        IDM_FILE_NEW            "Creates a new document"
        IDM_FILE_OPEN           "Open an existing document"
        IDM_FILE_SAVE           "Save the active document"
        IDM_FILE_SAVEAS         "Save the active document with a new name"
        IDM_FILE_PRINT          "Prints the active document"
        IDM_FILE_PREVIEW        "Displays full pages as they will be printed"
        IDM_FILE_EXIT           "Quits program"
        IDM_EDIT_UNDO           "Reverse the last action"
        IDM_EDIT_CUT            "Cuts the selection and puts it on the Clipboard"
        IDM_EDIT_COPY           "Copies the selection and puts it on the Clipboard"
        IDM_EDIT_PASTE          "Inserts the Clipboard contents at the insertion point"
        IDM_EDIT_PROP           "Opens property sheet for currently selected item"
        IDM_IGNORESIZE          "Toggle WM_SIZE handling to show auto-size/auto move"
        IDM_STRINGS             "Creates toolbar with strings"
        IDM_LARGEICONS          "Creates toolbar with large icons"
        IDM_SMALLICONS          "Creates toolbar with small icons"
        IDM_COMBOBOX            "Creates toolbar with combobox"
        IDM_NODIVIDER           "Toggle divider above toolbar"
        IDM_WRAPABLE            "Toggle toolbar resizing for narrow window"
        IDM_TOP                 "Toggle placing toolbar at top of parent"
        IDM_BOTTOM              "Toggle placing toolbar at bottom of parent"
        IDM_NOMOVEY             "Toggle inhibit moving window on Y-axis"
        IDM_NOPARENTALIGN       "Toggle inhibit aligning to parent"
        IDM_NORESIZE            "Toggle inhibit any sizing or moving"
        IDM_ADJUSTABLE          "Toggle ability for user to customize toolbar"
        IDM_ALTDRAG             "Toggle Alt-click to drag buttons"
        IDM_TOOLTIPS            "Toggle tooltip support"
        IDM_TB_CHECK            "Toggle button 0 checked state"
        IDM_TB_ENABLE           "Toggle button 1 enabled state"
```

```
IDM_TB_HIDE               "Toggle button 2 hidden state"
IDM_TB_INDETERMINATE      "Toggle button 3 indeterminate state"
IDM_TB_PRESS              "Toggle button 4 pressed state"
IDM_TB_BUTTONCOUNT        "Query button count"
IDM_TB_GETROWS            "Query row count"
IDM_TB_CUSTOMIZE          "Request customize dialog box"
IDM_STAT_IGNORESIZE       "Toggle WM_SIZE handling to show autosize/auto move"
IDM_STAT_SIZEGRIP         "Status bar to have sizing grip"
IDM_STAT_TOP              "Toggle placing status bar at top of parent"
IDM_STAT_BOTTOM           "Toggle placing status bar at bottom of parent"
IDM_STAT_NOMOVEY          "Toggle inhibit moving window on Y-axis"
IDM_STAT_NOPARENTALIGN    "Toggle inhibit aligning to parent"
IDM_STAT_NORESIZE         "Toggle inhibit any sizing or moving"
IDM_ST_GETBORDERS         "Query size of status bar borders"
IDM_ST_GETPARTS           "Query number of status bar parts"
IDM_ST_SETTEXT            "Set text in status bar"
IDM_ST_SIMPLE             "Toggle status bar simple state"
IDM_NOTIFICATIONS_CLEAR   "Clear contents of notification window"
IDM_VIEW_TOOLBAR          "Show/hide toolbar"
IDM_VIEW_STATUS           "Show/hide status bar"
IDM_VIEW_NOTIFICATIONS    "Show/hide notification window"
}
```

GADGETS.ICO

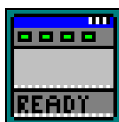


Рис. 12.8 Исходные тексты программы GADGETS

Программа GADGETS строит панель инструментов, строку состояния и рабочее окно, когда ее главное окно получает сообщение WM_CREATE. Местоположение каждого из окон во время создания значения не имеет, поскольку за сообщением WM_CREATE всегда следует сообщение WM_SIZE, и тогда программа GADGETS задаст местоположение своих дочерних окон.

Рабочее окно программы GADGETS содержит список для отображения подробностей уведомляющих сообщений WM_NOTIFY, получаемых главным окном. Эти сообщения посылаются из каждого из трех элементов управления общего пользования — панели инструментов, строки состояния, подсказки, когда возникает соответствующее событие. Панель инструментов посылает уведомляющие сообщения о нажатиях на кнопки мыши и запросах на изменение конфигурации; строка состояния тоже посылает уведомляющие сообщения; элемент управления "подсказка" запрашивает текст для окна подсказки, когда курсор мыши останавливается над инструментом панели инструментов.

Первые два всплывающих меню — File и Edit — обычные простые понятные пункты меню. Для них есть соответствующие кнопки. Четыре других пункта меню — Toolbar, Statusbar, Notifications и View управляют теми возможностями программы GADGETS, которые она призвана продемонстрировать. Некоторое число экспериментов покажет вам основные возможности панели инструментов, строки состояния и окон подсказки.

Управление панелью инструментов программы GADGETS

Программа GADGETS дает вам возможность понять взаимоотношения между тремя факторами, оказывающими влияние на панель инструментов и другие элементы управления: стили окна, сообщения, специфичные для элемента управления, и уведомляющие сообщения. Устанавливайте стиль окна, выбирая пункты подменю Styles меню Toolbar, посылайте сообщения, специфичные для элемента управления, выбирая пункты подменю Messages меню Toolbar, просматривайте информацию об уведомляющих сообщениях в списке рабочей области окна. Вы можете управлять различными свойствами панели инструментов: размером кнопок, наличием строк в кнопках и наличием комбинированного списка, используя другие пункты меню Toolbar.

В момент запуска панель инструментов программы GADGETS имеет маленькие кнопки, поддерживает окна подсказки, строки в кнопках отсутствуют, комбинированного списка нет. Панель инструментов расположена в верхней части родительского окна, а свойства автоматического изменения размеров и автоматического расположения могут быть добавлены путем установки различных стилей панели инструментов. Когда вы экспериментируете с местоположением панели инструментов, вы можете захотеть запретить программе GADGETS изменять местоположение панели инструментов, выбрав соответствующий пункт меню.

Программа GADGETS посылает простой набор сообщений своей панели инструментов. Большинство из них относятся к действиям с конкретными кнопками, такие как TB_CHECKBUTTON для установки нажатого (checking) или отжатого (unchecking) состояния кнопки, и TB_ENABLEBUTTON для разрешения или запрещения действий над кнопкой. Два сообщения с запросами позволяют вам узнать число кнопок и число строк.

Сообщение TB_CUSTOMIZE вызывает появление диалогового окна изменения конфигурации (Customize Toolbar dialog box), что дает пользователю возможность добавить или удалить кнопку из панели инструментов. Ключом к правильной работе диалогового окна изменения конфигурации является правильная обработка уведомляющих сообщений TBN_QUERYINSERT, TBN_QUERYDELETE и TBN_GETBUTTONINFO. При изменении конфигурации панели инструментов следите за потоком уведомляющих сообщений, чтобы прочувствовать последовательность этих запросов.

Управление строкой состояния программы GADGETS

Программа GADGETS содержит меню для управления стилями строки состояния и для посылки ей сообщений. Это меню имеет намного меньше команд, чем соответствующее меню панели инструментов, поскольку строка состояния значительно проще, чем панель инструментов.

Наиболее интересным аспектом управления строкой состояния в программе GADGETS является то, каким образом программа обрабатывает сообщение WM_MENUSELECT с целью отображения в строке состояния вспомогательной информации. Все строки, выводимые в строку состояния, расположены в таблице строк программы. Для упрощения установки соответствия между идентификаторами пунктов меню и идентификаторами текстовых строк они заданы одинаковыми. Например, идентификатор команды меню File Open задан равным IDM_FILE_OPEN; в то же время эта величина является идентификатором строки "Open an existing document".

Для того чтобы обеспечить отображение в строке состояния вспомогательной информации для всплывающих меню, программа GADGETS устанавливает соответствие описателей меню идентификаторам строк. В программе GADGETS имеется десять всплывающих меню, являющихся дочерними трех родительских меню: главного меню, всплывающего меню Toolbar и всплывающего меню Statusbar. Описания ресурсов-строк строки состояния для этих всплывающих меню в этих родительских меню начинаются со строк IDS_MAIN_MENU, IDS_TOOLBAR_MENU и IDS_STATUSBAR_MENU. Программа GADGETS устанавливает соответствие между описателями родительских меню и этими тремя значениями, передавая их в функцию *MenuHelp* для отображения необходимой строки вспомогательного текста.

Наборы страниц свойств

Наборы страниц свойств иногда также называют диалогами с закладками (tabbed dialogs). Они дают возможность объединить несколько диалоговых окон в одно, составное диалоговое окно. В наборе страниц свойств каждый отдельный диалог называется страницей свойств (property page). На рис. 12.9 показан набор страниц свойств дисплея Display Properties, доступный с рабочего стола (desktop) Windows 95. (Для доступа к этому набору страниц свойств активизируйте контекстное меню оболочки системы, щелкнув правой кнопкой мыши и выбрав пункт меню Properies — свойства.) Различные страницы в этом наборе страниц свойств позволяют пользователю удобно задать такие настройки дисплея, как обои (wallpaper), хранитель экрана (screen saver), цвета (colors), шрифты (fonts), а также изменить установки дисплейного драйвера, связанные с числом доступных цветов (available colors), разрешением устройства (device resolution), и даже сменить текущий драйвер дисплея.

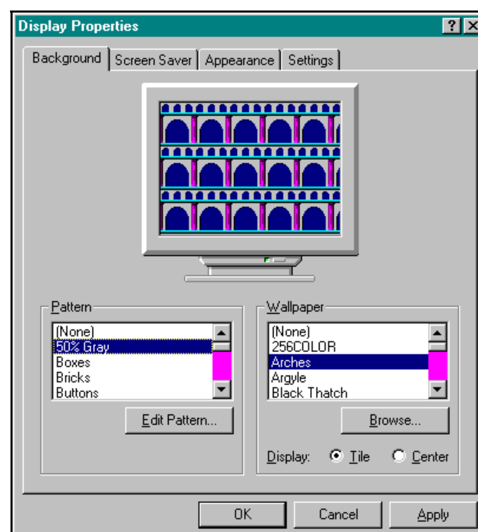


Рис. 12.9 Набор страниц свойств дисплея (Display Properties)

Набор страниц свойств позволяет модифицировать все установки сложного объекта, такого как, например, экран дисплея, путем объединения нескольких страниц свойств в единый элемент управления. Каждая страница имеет свою закладку (tab), щелчком на которой пользователь ее активизирует. Необязательная кнопка Apply позволяет пользователю посмотреть на изменения, им внесенные, без выхода из окна набора страниц свойств. Если результаты окажутся неудовлетворительными, то может быть сделан другой выбор, например, возврат к предыдущим установкам, которые доступны в наборе страниц свойств.

Другим типом составного диалогового окна, относящимся к набору страниц свойств, является мастер (*wizard*). Мастер — это набор диалогов, которые получают информацию от пользователя в определенном порядке, для выполнения какой-либо специфичной задачи. На рис. 12.10 показана первая из шести страниц мастера установки принтера в Windows 95 (Add Printer Wizard), который позволяет пользователю последовательно ввести всю детальную информацию для подключения нового системного принтера. Программный интерфейс для мастеров такой же как и для наборов страниц свойств, хотя и имеются небольшие отличия в интерфейсе пользователя: кнопки OK и Cancel заменены кнопками Back и Next. Эти кнопки предназначены для переходов между страницами мастера (вместо закладок в наборе страниц свойств). На последней странице мастера отображается кнопка Finish, которая при нажатии, разрешает выполнение той задачи, для решения которой и был активизирован мастер.

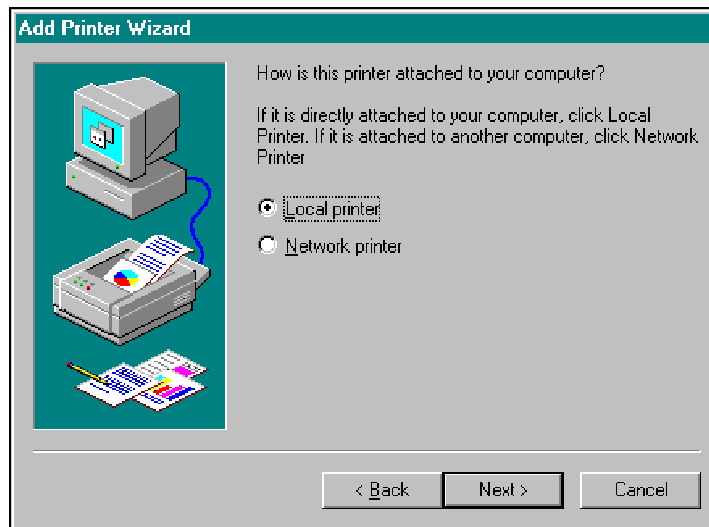


Рис. 12.10 Страница мастера установки принтера (Add Printer Wizard)

Создание набора страниц свойств

Для создания набора страниц свойств необходимы те же элементы, которые могли бы потребоваться для создания независимых диалоговых окон — шаблоны диалога и процедуры диалога. Они заносятся в некоторые структуры данных, а затем вызывается функция *PropertySheet*. Каждая страница свойств требует наличия своего собственного шаблона диалогового окна, описывающего расположение конкретных элементов управления. Каждая страница свойств требует также своей собственной диалоговой процедуры, обрабатывающей сообщение инициализации диалогового окна WM_INITDIALOG и уведомляющие сообщения от элементов управления WM_COMMAND. Диалоговая процедура страницы свойств обрабатывает также третье сообщение — WM_NOTIFY, которое используется для уведомлений о изменениях в состоянии самого набора страниц свойств.

Эти элементы элементизируются с использованием двух структур данных. Одна — для каждой страницы свойств, а вторая — заголовок, описывающий атрибуты набора страниц свойств целиком. Атрибуты каждой страницы свойств задаются в структуре типа PROPSHEETPAGE, а эти структуры объединяются в массив. Указатель на массив структур типа PROPSHEETPAGE включается в структуру данных типа PROPSHEETHEADER, содержащий атрибуты набора страниц свойств в целом.

Шаблоны диалога страниц свойств

Шаблоны диалога, используемые для страниц свойств, почти полностью идентичны тем шаблонам диалога, которые используются для обычных диалоговых окон. Шаблон диалога страницы свойств содержит обычные элементы управления, такие как окна редактирования (edits), списки (lists), кнопки (push buttons); он может также содержать элементы управления общего пользования, такие как, анимационное изображение (animation), списки типа Drag and Drop (drag lists), окна с движком для выбора значения из диапазона (trackbars) и полосы прокрутки, связанные с окном редактирования (up-down controls). Так же как и в случае обычного шаблона диалога, каждый элемент управления в шаблоне страницы свойств имеет уникальный идентификатор, указывая который при вызове функции *GetDlgItem* можно получить описатель окна.

Единственное отличие между обычным шаблоном диалога и шаблоном страницы свойств состоит в том, что последний обычно не содержит кнопок OK и Cancel. Набор страниц свойств создает и поддерживает разделяемый между всеми страницами свойств набор кнопок. Обычно, по умолчанию, набор страниц свойств имеет кнопки OK, Cancel и Apply, хотя возможно указание флага, удаляющего кнопку Apply, если в ней нет надобности. Возможно также задание флага, указывающего на необходимость вывода в наборе страниц свойств кнопки Help, которая будет доступна тем страницам свойств, которые поддерживают помощь. Мастера используют несколько отличающийся набор кнопок: Back и Next для пролистывания страниц мастера, и кнопку Finish на последней странице.

Для заданного набора страниц свойств вам следует так сформировать шаблоны страниц, чтобы они имели примерно одинаковые размеры и форму, поскольку набор страниц свойств приводит свои размеры в соответствие с самой широкой и высокой страницей свойств. Удачное расположение элементов управления в шаблонах диалоговых окон — задача, утомительная, требующая больших временных затрат. В контексте набора страниц свойств у вас еще прибавляется работы: необходимо скоординировать расположение различных шаблонов диалога.

Структура PROPSHEETPAGE

Как только будут созданы шаблоны диалоговых окон для каждой страницы свойств, и будет создан, как минимум, скелет процедуры диалогового окна, следующий шаг при создании набора страниц свойств состоит в заполнении структуры PROPSHEETPAGE для каждой страницы свойств. Структура PROPSHEETPAGE определена в файле PRSHT.H следующим образом:

```
typedef struct _PROPSHEETPAGE
{
       DWORD                       dwSize;    // = sizeof(PROPSHEETPAGE)
       DWORD                       dwFlags;
       HINSTANCE                   hInstance;
       union
       {
              LPCSTR               pszTemplate;  // по умолчанию
              LPCDLGTEMPLATE       pResource;    // PSP_DLGINDIRECT
       } DUMMYUNIONNAME;
       union
       {
              HICON                hIcon; // PSP_USEHICON
              LPCSTR               pszIcon;      // PSP_USEICONID
       } DUMMYUNIONNAME2;
       LPCSTR                      pszTitle;     // PSP_USETITLE
       DLGPROC                     pfnDlgProc;
       LPARAM                      lParam;
       LPFNPSPCALLBACKA            pfnCallback;  // PSP_USECALLBACK
       UINT FAR                    *pcRefParent;// PSP_USEREFPARENT
} PROPSHEETPAGE, FAR *LPPROPSHEETPAGE;
```

Комментарии в этом листинге описывают кратко каждый член структуры. Наличие флага с префиксом PSP_ означает, что этот флаг должен быть включен в поле *dwFlags* для активизации указанного элемента структуры данных. Ниже описаны детали о назначении конкретных полей структуры PROPSHEETPAGE:

- *dwSize* — текущая версия страницы свойств; должно быть равно *sizeof* (PROPSHEETPAGE).
- *dwFlags* — содержит один или несколько PSP-флагов, объединенных поразрядной оператором OR языка C. Большинство флагов влияют на то, будет или нет использоваться то или иное поле структуры PROPSHEETPAGE. Два дополнительных флага, неиспользуемых с этой целью, PSP_HASHELP и PSP_RTLREADING. Флаг PSP_HASHELP делает доступной кнопку Help, когда активна данная страница (флаг PSP_HASHELP должен быть указан в структуре PROPSHEETHEADER для того, чтобы кнопка Help присутствовала). Флаг PSP_RTLREADING заставляет страницу свойств использовать запись справа налево (для еврейского и арабского языков).
- *hInstance* — идентифицирует выполняемый файл, из которого загружены ресурсы диалоговых шаблонов (*pszTemplate*) и значков (*pszIcon*).
- Шаблон диалогового окна идентифицируется *pszTemplate* (по умолчанию) для шаблонов, загруженных из ресурсов, или *pResource* (необходимо задание флага PSP_DLGINDIRECT в поле *dwFlags*) для шаблонов, загруженных из памяти.
- На закладке страницы свойств дополнительно может быть отображен значок. Значок может быть идентифицирован описателем значка *hIcon* (при установленном флаге PSP_USEHICON в поле *dwFlags*) или идентификатором ресурса-значка *pszIcon* (при установленном флаге PSP_USEICONID в поле *dwFlags*).

- По умолчанию заголовок закладки страницы свойств — заголовок шаблона диалога, кроме тех случаев, когда в поле *dwFlags* установлен флаг PSP_USETITLE, что приводит к использованию заголовка *pszTitle*.
- *pfnDlgProc* — определяет диалоговую процедуру для страницы свойств.
- *lParam* — задает начальное значение, передаваемое в функцию обратного вызова страницы свойств, заданную параметром *pfnCallback*.
- *pfnCallback* — функция обратного вызова (при установленном в поле *dwFalgs* флаге PSP_USECALLBACK), вызываемая перед тем, как страница свойств создается, и после того, как уничтожается.
- *pcRefParent* идентифицирует счетчик ссылок для набора страниц свойств. Это указатель на величину типа UINT, которая увеличивается на единицу при создании страницы свойств, и уменьшается на единицу при удалении страницы свойств. В любой момент времени величина этого счетчика позволяет вам определить, сколько страниц свойств в данный момент присутствует.

Ниже показано, как в примере программы PROPERTY в этой главе заполняется массив структур типа PROPSHEETPAGE:

```
PROPSHEETPAGE pspage[2];

// Обнуляем данные
ZeroMemory(&pspage, 2 * sizeof(PROPSHEETPAGE));

// инициализируем данные первой страницы
pspage[0].dwSize      = sizeof(PROPSHEETPAGE);
pspage[0].dwFlags     = PSP_USECALLBACK | PSP_USEICONID;
pspage[0].hInstance   = hInst;
pspage[0].pszTemplate = MAKEINTRESOURCE(IDD_STYLES);
pspage[0].pszIcon     = MAKEINTRESOURCE(IDI_PAGE1);
pspage[0].pfnDlgProc  = StyleDlgProc;
pspage[0].lParam      =(LPARAM)&dwChildStyle;
pspage[0].pfnCallback = StylePageProc;

// инициализируем данные второй страницы
pspage[1].dwSize      = sizeof(PROPSHEETPAGE);
pspage[1].dwFlags     = PSP_USECALLBACK | PSP_USEICONID | PSP_HASHELP;
pspage[1].hInstance   = hInst;
pspage[1].pszTemplate = MAKEINTRESOURCE(IDD_EXSTYLES);
pspage[1].pszIcon     = MAKEINTRESOURCE(IDI_PAGE2);
pspage[1].pfnDlgProc  = ExStyleDlgProc;
pspage[1].lParam      =(LPARAM)&dwChildExStyle;
pspage[1].pfnCallback = ExStylePageProc;
```

Член структуры PROPSHEETPAGE *pfnDlgProc* содержит указатель на процедуру диалогового окна. В добавок к сообщениям WM_INITDIALOG и WM_COMMAND, которые обрабатываются в большинстве процедур диалоговых окон, диалоговые процедуры страниц свойств должны обрабатывать третье сообщение — WM_NOTIFY. Ниже приведен примерный шаблон диалоговой процедуры для страницы свойств:

```
BOOL CALLBACK EmptyPageDlgProc(HWND hwndDlg, UINT msg, WPARAM wParam, LPARAM lParam)
{
       switch(msg)
       {
       case WM_INITDIALOG:
              [инициализация диалогового окна]
              break;
       case WM_COMMAND:
              [уведомления от обычных диалоговых элементов управления]
              break;
       case WM_NOTIFY:
              {
                     LPNMHDR pnmh =(LPNMHDR)lParam;
                     [уведомления от набора страниц свойств]
              }
       }
}
```

Член *pfnCallback* структуры PROPSHEETPAGE ссылается на функцию обратного вызова, которая вызывается перед тем, как конкретная страница свойств создается, и сразу после того, как она уничтожается. Смысл использования этой функции состоит в том, чтобы иметь возможность прочитать данные члена *lParam* структуры

PROPSHEETPAGE, который содержит начальное значение, посланное создателем страницы свойств странице свойств (например, указатель на изменяемые данные). Ниже приведен шаблон функции обратного вызова страницы свойств:

```
static LPARAM InputData;

UINT CALLBACK EmptyPageProc(HWND hwnd, UINT uMsg, LPPROPSHEETPAGE ppsp)
{
	switch(uMsg)
	{
	case PSPCB_CREATE:
		InputData = ppsp->lParam;
		return TRUE;
	case PSPCB_RELEASE:
		return 0;
	}
}
```

В текущей реализации параметр описателя окна *hwnd* всегда равен NULL. Параметр *uMsg* равен PSPCB_CREATE для вызова перед созданием страницы свойств и равен PSPCB_RELEASE после того, как она будет уничтожена. Хотя уведомление при создании посылается только, если страница свойств действительно создается, уведомление при уничтожении всегда посылается каждой странице свойств, вне зависимости от того, была она создана или нет.

После заполнения массива структур типа PROPSHEETPAGE следующим шагом в создании набора страниц свойств является занесение указателя на этот массив в поле *ppsp* структуры заголовка набора страниц свойств PROPSHEETHEADER.

Структура PROPSHEETHEADER

Структура PROPSHEETHEADER определена в файле PRSHT.H следующим образом:

```
typedef struct _PROPSHEETHEADER
{
	DWORD		dwSize;	// = sizeof(PROPSHEETHEADER)
	DWORD		dwFlags;
	HWND		hwndParent;
	HINSTANCE	hInstance;
	union
	{
		HICON	hIcon; // PSH_USEHICON
		LPCSTR pszIcon;	// PSH_USEICONID
	} DUMMYUNIONNAME;
	LPCSTR		pszCaption;
	UINT		nPages;
	union
	{
		UINT	nStartPage;	//(по умолчанию)
		LPCSTR pStartPage;	// PSH_USEPSTARTPAGE
	} DUMMYUNIONNAME2;
	union
	{
		LPCPROPSHEETPAGE	ppsp;		// PSH_PROPSHEETPAGE
		HPROPSHEETPAGE FAR	*phpage;
	} DUMMYUNIONNAME3;
	PFNPROPSHEETCALLBACK		pfnCallback;	// PSH_USECALLBACK
} PROPSHEETHEADER, FAR *LPPROPSHEETHEADER;
```

Комментарии в этом листинге описывают кратко каждый член структуры. Наличие флага с префиксом PSH_ означает, что этот флаг должен быть включен в поле *dwFlags* для активизации указанного члена структуры данных. Ниже описаны детали о назначении конкретных полей структуры PROPSHEETHEADER:

- *dwSize* — текущая версия набора страниц свойств; должно быть равно *sizeof* (PROPSHEETHEADER).
- *dwFlags* — содержит один или несколько PSH-флагов, объединенных поразрядной оператором OR языка C. Некоторые флаги определяют, будет или нет использоваться то или иное поле структуры PROPSHEETHEADER, как показано в предыдущем листинге.
- *hwndParent* — идентифицирует родительское окно набора страниц свойств. Для модального набора страниц свойств родительское окно недоступно.

- В наборах страниц свойств, использующих ресурсы (такие как значки), параметр *hInstance* идентифицирует модуль, из которого загружаются ресурсы.
- Набор страниц свойств может иметь, а может и не иметь, значок, идентифицируемый либо описателем *hIcon* (требует установки флага PSH_USEHICON в поле *dwFlags*), либо ресурсом значка *pszIcon* (требует установки флага PSH_USEICONID в поле *dwFlags*).
- Заголовок набора страниц свойств задается строкой *pszCaption*.
- Число страниц свойств в наборе (т. е. размер массива страниц свойств) равен *nPages*.
- Первая страница свойств для отображения задается или *nStartPage* (индекс с нулевой базой), или заголовком первой отображаемой страницы *pStartPage* (требует установки флага PSH_USEPSTARTPAGE в поле *dwFlags*).
- Страницы свойств набора определяются с использованием массива одного из двух типов. Первый — массив описателей страниц свойств, возвращаемых функциями *CreatePropertySheetPage* (по умолчанию), идентифицируемый *phpage*. Второй — массив структур типа PROPSHEETPAGE (требует установки флага PSH_PROPSHEETPAGE в поле *dwFlags*), идентифицируемый *ppsp*.
- В добавок к диалоговым процедурам каждой страницы свойств набор страниц свойств в целом может иметь функцию обратного вызова, которая вызывается перед тем, как создается любая из страниц свойств. Эта функция идентифицируется *pfnCallback*.

Параметр *dwFlags* обеспечивает тонкую настройку набора страниц свойств. Некоторые флаги показывают, будет ли тот или иной член структуры PROPSHEETHEADER использоваться или игнорироваться. В предыдущем листинге этой структуры конкретные члены помечены именами соответствующих флагов. Другие флаги приведены в следующей таблице:

Флаг	**Назначение**
PSH_HASHELP	Отображает кнопку Help в наборе страниц свойств, которая будет доступной, если страница свойств имеет флаг PSP_HASHELP. При нажатии, она посылает сообщение WM_NOTIFY с кодом уведомления PSN_HELP.
PSH_MODELESS	Строит немодальный набор страниц свойств.
PSH_NOAPPLYNOW	Скрывает кнопку Apply.
PSH_PROPTITLE	Добавляет слово "Properties" в строку заголовка после заголовка, заданного членом *pszCaption* структуры PROPSHEETHEADER.
PSH_WIZARD	Создает мастера вместо набора страниц свойств.

Функция PropertySheet

Функция, которая действительно создает набор страниц свойств, называется *PropertySheet*. Она принимает единственный параметр: указатель на структуру типа PROPSHEETHEADER. Набор страниц свойств, по умолчанию, является модальным, поэтому функция *PropertySheet* возвращает TRUE, когда пользователь нажимает кнопку OK, и FALSE, когда пользователь нажимает кнопку Cancel. Вызов функции *PropertySheet* выглядит как обычный вызов функции *DialogBox*:

```
if(PropertySheet(&pshead))
{
        [пользователь нажал OK – сохранить изменения]
}
else
{
        [пользователь нажал Cancel – отменить изменения]
}
```

Для создания немодального набора страниц свойств используется функция *PropertySheet* с установленным флагом PSH_MODELESS в структуре заголовка набора страниц свойств. Вместо возврата значения типа Boolean функция *PropertySheet* возвращает описатель окна набора страниц свойств:

```
// Создание немодального набора страниц свойств. Возвращается описатель окна
HWND hwndPropSheet =(HWND) PropertySheet(&pshead);
```

Процедуры диалогового окна страницы свойств

Каждая страница свойств имеет собственную диалоговую процедуру. При создании страницы свойств диалоговая процедура получает сообщение WM_INITDIALOG и инициализирует элементы управления страницы. Набор страниц свойств не создает страницу свойств до тех пор, пока она не понадобится. Поэтому, конкретная страница

свойств не получает сообщение WM_INITDIALOG при создании набора страниц свойств, и не получит его совсем, если эта страница не будет активизирована.

Как и диалоговая процедура обычного диалогового окна, диалоговая процедура страницы свойств получает уведомляющие сообщения WM_COMMAND от своих элементов управления. Однако, поскольку кнопок OK и Cancel обычного диалогового окна нет, на странице свойств, вам не следует вызывать функцию *EndDialog* для завершения страницы свойств или всего набора страниц свойств. Вместо этого, сам набор страниц свойств удаляет каждую страницу и самого себя при завершении своей работы.

Сообщение WM_NOTIFY

Набор страниц свойств посылает сообщение WM_NOTIFY для уведомления диалоговых процедур о значимых изменениях своего состояния. Эти сообщения возникают, когда одна из страниц активизируется, а другая теряет активность. Для страницы, в которой требуется проверка корректности введенных данных, уведомление о потере активности дает возможность проверить необходимые значения, и, возможно, предотвратить потерю активности, если на странице имеются некорректные данные. Сообщение WM_NOTIFY — запросы на применение внесенных изменений — посылается при нажатии пользователем кнопок OK или Apply, а запросы на игнорирование изменений — при нажатии пользователем кнопки Cancel. Когда набор страниц свойств создается как мастер, третий набор сообщений WM_NOTIFY информирует процедуру диалога о нажатии пользователем кнопок: Back, Next и Finish.

Параметр *lParam* сообщения WM_NOTIFY содержит указатель на структуру типа NMHDR. Поле *code* этой структуры данных идентифицирует причину, вызвавшую посылку уведомления. Шаблон программы, обрабатывающей уведомления набора страниц свойств, приведен ниже:

```
case WM_NOTIFY:
       {
              LPNMHDR pnmh =(LPNMHDR)lParam;
              switch(pnmh->code)
              {
              case PSN_SETACTIVATE:
                     [активизируется страница свойств]
                     break;
              case PSN_KILLACTIVE:
                     [страница свойств теряет активность]
                     break;
              case PSN_APPLY:
                     [пользователь нажал кнопку OK или Apply]
                     break;

              case PSN_RESET:
                     [пользователь нажал кнопку Cancel]
                     break;
              case PSN_QUERYCANCEL:
                     [пользователь нажал кнопку Cancel. Продолжать?]
                     break;
              case PSN_HELP:
                     [пользователь нажал кнопку Help]
                     break;
              case PSN_WIZBACK:
                     [пользователь нажал кнопку Back в окне мастера]
                     break;
              case PSN_WIZNEXT:
                     [пользователь нажал кнопку Next в окне мастера]
                     break;
              case PSN_WIZFINISH:
                     [пользователь нажал кнопку Finish в окне мастера]
                     break;
              }
       }
```

Смысл конкретных уведомлений набора страниц свойств описан в следующей таблице:

Код уведомления	Действие пользователя	Описание
PSN_APPLY	Пользователь нажал кнопку OK (страница свойств закрывается) или Apply (страница свойств не закрывается). Активные и неактивные страницы получают это уведомление, только если они были созданы.	Запрос принятия изменений.

Код уведомления	Действие пользователя	Описание
PSN_HELP	Нажата кнопка Help.	
PSN_KILLACTIVE	Пользователь либо переходит от одной страницы к другой, либо нажал кнопку Apply.	Уведомление для проверки содержания страницы; можно предотвратить закрытие страницы, если данные некорректны.
PSN_QUERYCANCEL	Нажата кнопка Cancel.	Запрос о том, может ли пользователь закрыть окно набора страниц свойств.
PSN_RESET	Нажата кнопка Cancel.	Уведомление об игнорировании всех изменений, сделанных после последнего нажатия кнопки Apply.
PSN_SETACTIVE	Пользователь активизирует страницу первый раз после открытия набора, или переключается на другую страницу, или нажал кнопку Apply, что приводит к потере активности и установке активности вновь.	Уведомление о том, что страница активизируется.
PSN_WIZBACK	Пользователь нажал кнопку Back в окне мастера.	
PSN_WIZNEXT	Пользователь нажал кнопку Next в окне мастера.	
PSN_WIZFINISH	Пользователь нажал кнопку Finish в окне мастера.	

Важнейшим из этих уведомлений является PSN_APPLY, означающее, что пользователь нажал кнопку OK или Apply. Внесенные изменения должны быть приняты. Это уведомление заменяет сообщение WM_COMMAND с идентификатором IDOK в обычном окне диалога. В отличие от обычного диалога, однако, функция *EndDialog* не вызывается для диалогового окна страницы свойств (это могло бы привести к порче окна набора страниц свойств). Сам набор страниц свойств отвечает за создание и удаление страниц свойств.

Уведомление PSN_APPLY посылается при нажатии на кнопку OK или Apply. В большинстве случаев пользователь почти не различает эти кнопки по смыслу. Однако, ситуация меняется, если окно набора страниц свойств — немодальное. При нажатии на кнопку OK набор страниц свойств должен быть уничтожен. Ниже приведен соответствующий фрагмент программы:

```
// Проверка при работе с немодальным окном набора страниц свойств
if( IsWindowEnabled(hwndMain) )
{
       LPPSHNOTIFY psh =(LPPSHNOTIFY)lParam;
       HWND hwndPropSheet;

       // Нажатие на кнопку Apply не означает удаления
       if(pnmh->code == PSN_APPLY && psh->lParam == 0)
              return TRUE;

       // Нажатие на кнопку OK или Cancel означает удаление
       hwndPropSheet = GetParent(hwndDlg);
       DestroyWindow(hwndPropSheet);
       hwndModeless =  NULL;
}
```

Уведомляющие сообщения PSN_APPLY и PSN_RESET являются широковещательными, т. е. они посылаются всем активным и неактивным страницам свойств набора, с одним только ограничением: страница свойств должна уже быть создана. Поскольку страница свойств в наборе создается только тогда, когда пользователь ее активизирует, возможно, и достаточно часто, встречается случай, что диалоговая процедура страницы свойств не получает сообщения WM_NOTIFY. На практике это означает, что диалоговое окно страницы свойств будет всегда получать сообщение WM_INITDIALOG до того, как получит сообщение WM_NOTIFY.

Управление набором страниц свойств

Управление набором страниц свойств состоит в посылке ему сообщений. Для этого необходимо получить описатель окна набора страниц свойств. Диалоговые процедуры страниц свойств получают описатель конкретной страницы, которая является дочерним окном всего набора страниц свойств. Для получения описателя окна набора страниц свойств необходимо вызвать функцию *GetParent*:

```
// Получить описатель окна набора страниц свойств
HWND hwndPropSheet = GetParent(hwndDlg);
```

Используя этот описатель, можно посылать сообщения набору страниц свойств с помощью функции *SendMessage* или с использованием макросов сообщений из файла PRSHT.H.

В следующей таблице приведены описания сообщений набора страниц свойств:

Категория	Сообщение	Описание
Активизация страницы	PSM_SETCURSEL	Активизация указанной страницы по индексу или описателю.
	PSM_SETCURSELID	Активизация указанной страницы по ее идентификатору в ресурсах.
Состояния кнопок	PSM_APPLY	Имитирует нажатие пользователем кнопки Apply, что вызывает получение всеми созданными страницами сообщения WM_NOTIFY с кодом уведомления PSN_APPLY.
	PSM_CANCELTOCLOSE	Запрещает кнопку Cancel и изменяет текст кнопки OK на Close.
	PSM_CHANGED	Устанавливает флаг доступности кнопки Apply. Если флаг установлен для любой страницы, то кнопка Apply доступна; в противном случае — нет.
	PSM_UNCHANGED	Сбрасывает флаг доступности кнопки Apply. (См. описание сообщения PSM_CHANGED выше.)
	PSM_PRESSBUTTON	Имитирует нажатие пользователем кнопок.
	PSM_SETFINISHTEXT	Делает доступной кнопку Finish в мастере и задает ее текст.
	PSM_SETWIZBUTTONS	Разрешает и запрещает кнопки в мастере.
Редактирование списка	PSM_ADDPAGE	Добавляет страницу в набор страниц.
	PSM_REMOVEPAGE	Удаляет страницу из набора.
Запрос	PSM_GETTABCONTROL	Получает описатель элемента управления набор закладок (tab).
	PSM_GETCURRENTPAGEHWND	Получает описатель окна текущей страницы свойств.
	PSM_QUERYSIBLINGS	При получении этого сообщения набор страниц свойств посылает сообщение PSM_QUERYSIBLINGS диалоговым процедурам всех открытых страниц свойств. Термин "открытая" означает, что пользователь уже открывал эту страницу свойств, т. е. ее диалоговая процедура получила сообщение WM_INITDIALOG.
Изменения, требующие перезагрузки	PSM_REBOOTSYSTEM	Приводит к тому, что функция *PropertySheet* вернет ID_PSREBOOTSYSTEM.
	PSM_RESTARTWINDOWS	Приводит к тому, что функция *PropertySheet* вернет ID_PSRESTARTWINDOWS.
Другие	PSM_ISDIALOGMESSAGE	Разрешает интерфейс клавиатуры для немодального набора страниц свойств (см. подробнее ниже).
	PSM_SETTITLE	Устанавливает заголовок набора страниц свойств.

Разрешение и запрещение кнопки Apply

Кнопка Apply запрещена, когда набор страниц свойств создается, и делается разрешенной (доступной) из процедуры диалогового окна конкретной страницы свойств, если пользователь изменяет какие-либо данные. Для уведомления набора страниц свойств об изменении данных используется сообщение PSM_CHANGED. Отмена изменений и запрещение кнопки Apply осуществляется путем посылки сообщения PSM_UNCHANGED набору страниц свойств. Ниже показано, как с использованием макросов из файла PRSHT.H, посылаются сообщения набору страниц свойств:

```
hwndSheet = GetParent(hwndDlg);
PropSheet_Changed(hwndSheet, hwndDlg);
```

Набор страниц свойств хранит бит изменения данных для каждой страницы свойств, поэтому кнопка Apply становится доступной, когда *любая* из страниц свойств сообщает, что ее данные изменились, независимо от того, какая из страниц видима в данный момент.

Применение изменений

Когда пользователь нажимает кнопку OK или Apply, наступает момент, когда необходимо применить сделанные изменения. Оба события обрабатываются одинаково, за исключением того, что при нажатии кнопки OK окно набора страниц свойств удаляется, а при нажатии кнопки Apply — нет. Один путь объединения обработки этих

изменений состоит в посылке сообщения родительскому окну с запросом о применении изменений. Этот путь отличается от пути, которым обрабатывается модальное диалоговое окно в обычной Windows-программе, когда изменения применяются только после завершения функции *DialogBox*. Фактически, это более похоже на метод обработки изменений, применяемый при работе с немодальными диалоговыми окнами.

Уведомления о нажатии на кнопку OK и кнопку Apply — одинаковы. Диалоговые процедуры открытых страниц свойств — страниц, которые были хоть раз активизированы пользователем и получили сообщение WM_INITDIALOG — получают сообщение WM_NOTIFY с кодом уведомления PSN_APPLY. Разумным ответом является сохранение, когда-либо сделанных изменений, и посылка сообщения своему родительскому окну.

Чтобы сделать окно набора страниц свойств более похожим на модальное диалоговое окно, делайте кнопку Apply недоступной, создавая набор страниц свойств с установленным флагом PSH_NOAPPLYNOW. (Раньше кнопка имела название "Apply Now".) Применение изменений требует, в этом случае, анализа значения, возвращаемого функцией *PropertySheet*: TRUE — при нажатии кнопки OK, FALSE — при нажатии кнопки Cancel.

Помощь при работе с набором страниц свойств

Существует два механизма для запроса помощи при работе с набором страниц свойств: кнопка Help и клавиша <F1>. Кнопка Help отображается только в наборах страниц свойств, созданных с установленным флагом PSH_HASHELP, но остается недоступной до тех пор, пока не будет активизирована страница свойств с установленным флагом PSP_HASHELP. Нажатие на кнопку Help приводит к посылке сообщения WM_NOTIFY с кодом уведомления PSN_HELP диалоговой процедуре страницы свойств. Вам следует реагировать на это сообщение, отображая окно помощи со вспомогательной информацией о конкретной странице свойств.

Любое Win32-окно поддерживает клавишу помощи <F1>. Оконная процедура по умолчанию отвечает на необработанные нажатия клавиши <F1>, посылая сообщение WM_HELP родительскому окну. Поэтому, никакой специальной обработки нажатия клавиши <F1> не требуется, кроме обработки сообщения WM_HELP.

Обработка немодального набора страниц свойств

Также как большинство диалоговых окон, большинство наборов страниц свойств — модальные. Если вы решите создать немодальный набор страниц свойств, потребуются небольшие изменения. Во-первых, необходимо указать флаг PSH_MODELESS в структуре PROPSHEETHEADER. Все другие изменения аналогичны тем, какие надо сделать для создания немодального диалогового окна.

Чтобы разрешить интерфейс клавиатуры для немодального диалогового окна, вы вызываете функцию *IsDialogMessage*. Эта функция фильтрует клавиатурные сообщения в цикле обработки сообщений. При получении клавиатурного сообщения элемент управления, имеющий фокус ввода, получает сообщение WM_GETDLGCODE. Смысл сообщения состоит в запросе: "Какой тип клавиатурного ввода вам нужен?" На основе ответа элемент управления может получить ввод от клавиатуры или, в ответ на нажатие клавиш <Tab>, <Alt> + мнемоника, управления курсором, потерять фокус ввода, который перейдет к следующему по порядку элементу управления диалогового окна.

В немодальных наборах страниц свойств вместо вызова функции *IsDialogMessage* родительскому окну окна набора страниц свойств посылается сообщение PSM_ISDIALOGMESSAGE. Программа, имеющая немодальный набор страниц свойств, должна иметь примерно такой цикл обработки сообщений:

```
while(GetMessage(&msg, NULL, 0, 0))
{
        if((hwndModeless) &&(!(PropSheet_IsDialogMessage(hwndModeless, &msg))))
                continue;

        TranslateMessage(&msg);
        DispatchMessage(&msg);
}
```

В этом фрагменте предполагается, что *hwndModeless* либо равен описателю немодального окна набора страниц свойств, если он существует, либо равен NULL, если набор страниц свойств не существует.

Установка кнопок мастера

При создании мастера вместо набора страниц свойств диалоговая процедура для каждой страницы свойств должна управлять кнопками мастера. Набор страниц свойств имеет кнопки OK и Cancel, а мастер — кнопки Back и Next. Обе кнопки остаются доступными до тех пор, пока вы их явно не запретите. На последней странице мастера пользователь ожидает увидеть кнопку Finish на месте кнопки Next.

Для управления установкой кнопок мастера, посылайте ему сообщение PSM_SETWIZBUTTONS с параметром *lParam*, равным комбинации трех значений: PSWIZB_NEXT, PSWIZB_BACK, PSWIZB_FINISH. Удобный момент

для посылки этого сообщения — когда страница мастера становится активной, о чем узнает диалоговая процедура страницы свойств при получении уведомляющего сообщения WM_NOTIFY с кодом уведомления PSN_SETACTIVE. В ответ на уведомление PSN_SETACTIVE мастер может предотвратить установку новой активной страницы, возвращая 1; установить активность нужной страницы, возвращая ее идентификатор ресурса; разрешить активизацию страницы, возвращая 0.

Когда первая страница мастера становится активной, запретите кнопку Back путем посылки сообщения PSM_SETWIZBUTTONS с установленным флагом PSWIZB_NEXT в параметре *lParam*. Макрос *PropSheet_SetWizButtons* обеспечивает один путь для посылки этого сообщения:

```
// разрешить кнопку Next мастера
if( bWizard && pnmh->code == PSN_SETACTIVE )
{
        HWND hwndSheet = GetParent(hwndDlg);
        PropSheet_SetWizButtons(hwndSheet, PSWIZB_NEXT);
}
```

На последней странице мастера необходимо разрешить кнопку Back и изменить название кнопки Next на "Finish". Это делается путем посылки сообщения PSM_SETWIZBUTTONS с параметром *lParam* равным результату применения операции OR языка С к флагам PSWIZB_BACK и PSWIZB_FINISH.

```
// разрешить кнопки Back и Finish
if( bWizard && pnmh->code == PSN_SETACTIVE )
{
        HWND hwndSheet = GetParent(hwndDlg);
        PropSheet_SetWizButtons(hwndSheet, PSWIZB_BACK | PSWIZB_FINISH);
}
```

Программа PROPERTY

На рис. 12.11 и рис. 12.12 приведены две страницы свойств из набора страниц свойств, созданного в программе PROPERTY. Взятые вместе эти страницы свойств определяют все основные флаги стиля окна (WS_ и WS_EX_), которые могут быть переданы в функцию *CreateWindowEx*. Главное окно программы PROPERTY показывает два окна: первое ("First Window") создается с использованием флагов, заданных в наборе страниц свойств.

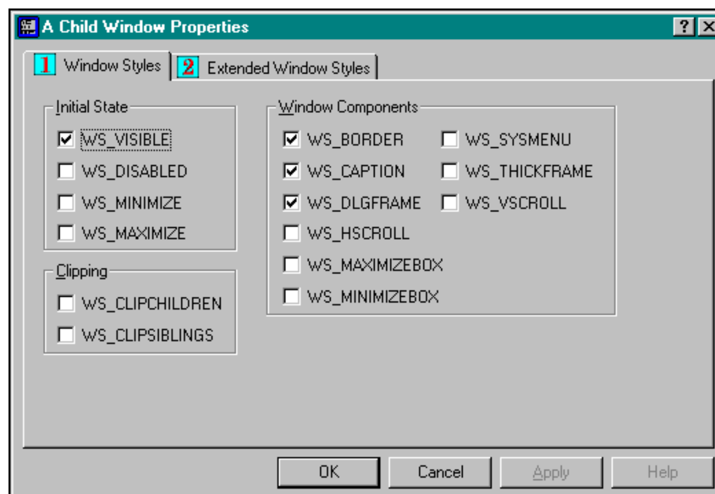


Рис. 12.11 Страница свойств Window Styles

Другое окно ("Second Window") существует только для того, чтобы показать влияния конкретных флагов отсечения (WS_CLIPSIBLINGS) на предотвращение конкуренции между окнами, имеющего общее родительское окно, когда они оба разделяют пиксели на экране. Кроме иллюстрации основ создания наборов страниц свойств, программа PROPERTY иллюстрирует работу большинства флагов стиля создания окна. Листинги программы PROPERTY приведены на рис. 12.13.

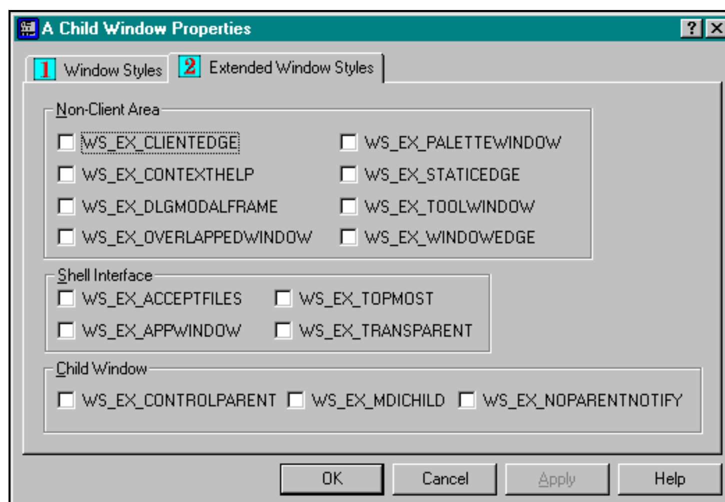


Рис. 12.12 Страница свойств Extended Window Styles

PROPERTY.MAK

```
#------------------------
# PROPERTY.MAK make file
#------------------------

property.exe : property.obj helper.obj notify.obj \
               page1.obj page2.obj sheet.obj property.res
     $(LINKER) $(GUIFLAGS) -OUT:property.exe property.obj \
         helper.obj notify.obj page1.obj page2.obj sheet.obj \
         property.res $(GUILIBS)

property.obj : property.c comcthlp.h property.h
     $(CC) $(CFLAGS) property.c

helper.obj : helper.c property.h
     $(CC) $(CFLAGS) helper.c

notify.obj : notify.c notify.h
     $(CC) $(CFLAGS) notify.c

page1.obj : page1.c helper.h notify.h property.h
     $(CC) $(CFLAGS) page1.c
page2.obj : page2.c helper.h notify.h property.h
     $(CC) $(CFLAGS) page2.c

sheet.obj : sheet.c comcthlp.h property.h
     $(CC) $(CFLAGS) sheet.c

property.res : property.rc property.ico
     $(RC) $(RCVARS) property.rc
```

PROPERTY.H

```
// Menu item identifiers
#define IDM_OVERLAPPED    100
#define IDM_POPUP         101
#define IDM_CHILD         102
#define IDM_WIZARD        200
#define IDM_HASHELP       201
#define IDM_MODELESS      202
#define IDM_MULTILINETABS 203
#define IDM_NOAPPLYNOW    204
#define IDM_PROPTITLE     205
#define IDM_RTLREADING    206
```

```
// Dialog template IDs
#define IDD_STYLES                        101
#define IDD_EXSTYLES                      102

// Icon IDs
#define IDI_PAGE1                         103
#define IDI_PAGE2                         104

// Dialog Control IDs
#define IDC_BORDER                        1000
#define IDC_CAPTION                       1001
#define IDC_VISIBLE                       1005
#define IDC_DISABLED                      1006
#define IDC_DLGFRAME                      1007
#define IDC_MINIMIZE                      1008
#define IDC_MAXIMIZE                      1009
#define IDC_HSCROLL                       1010
#define IDC_MAXIMIZEBOX                   1011
#define IDC_MINIMIZEBOX                   1012
#define IDC_SYSMENU                       1013
#define IDC_THICKFRAME                    1014
#define IDC_VSCROLL                       1015
#define IDC_CLIPCHILDREN                  1016
#define IDC_CLIPSIBLINGS                  1017
#define IDC_CLIENTEDGE                    1018
#define IDC_CONTEXTHELP                   1019
#define IDC_DLGMODALFRAME                 1020
#define IDC_EXOVERLAPPED                  1021
#define IDC_PALETTE                       1022
#define IDC_STATICEDGE                    1023
#define IDC_TOOLWINDOW                    1024
#define IDC_WINDOWEDGE                    1025
#define IDC_ACCEPTFILES                   1026
#define IDC_APPWINDOW                     1027
#define IDC_TOPMOST                       1028
#define IDC_TRANSPARENT                   1029
#define IDC_CONTROLPARENT                 1030
#define IDC_MDICHILD                      1031
#define IDC_NOPARENTNOTIFY                1032
#define IDM_WINDOW_PROPERTIES             40001
#define IDC_STATIC                        -1
#define IDI_APP                           1000

// Private message
#define PM_CREATEWINDOW   WM_APP

// Property Sheet Functions(in SHEET.C)
BOOL CreatePropertySheet(HWND hwndParent);

// Property Page Functions(in PAGE1.C and PAGE2.C)
UINT CALLBACK StylePageProc(HWND, UINT, LPPROPSHEETPAGE);
BOOL CALLBACK StyleDlgProc(HWND, UINT, WPARAM, LPARAM);
UINT CALLBACK ExStylePageProc(HWND, UINT, LPPROPSHEETPAGE);
BOOL CALLBACK ExStyleDlgProc(HWND, UINT, WPARAM, LPARAM);
```

HELPER.H

```
void SetButtonCheck(HWND hwndDlg, int CtrlID, BOOL bCheck);
BOOL QueryButtonCheck(HWND hwndDlg, int CtrlID);
```

PROPERTY.C

```
/*--------------------------------------
   PROPERTY.C -- Property sheet example
                 (c) Paul Yao, 1996
  --------------------------------------*/
```

```
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include "comcthlp.h"
#include "property.h"

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

char szAppName[]  = "Property Sheet";
HINSTANCE hInst;
HWND hwndMain = NULL;
HWND hwndChild = NULL;
HWND hwndModeless = NULL;
HICON hiconApp;  // Application icon
BOOL bWizard;    // Flag whether PSH_WIZARD is set

// Values modified by property sheet
DWORD dwChildStyle = WS_CHILD | WS_VISIBLE | WS_BORDER | WS_CAPTION;
DWORD dwChildExStyle = 0L;

// Value modified by menu item selection
DWORD dwSheetStyles = PSH_PROPTITLE;

//-------------------------------------------------------------------
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR lpszCmdLine, int cmdShow)
     {
     HWND       hwnd;
     MSG        msg;
     WNDCLASSEX wc;

     hInst = hInstance;
     hiconApp = LoadIcon(hInst, MAKEINTRESOURCE(IDI_APP));

     ZeroMemory(&wc, sizeof(WNDCLASSEX));
     wc.cbSize        = sizeof(wc);
     wc.lpszClassName = "MAIN";
     wc.hInstance     = hInstance;
     wc.lpfnWndProc   = WndProc;
     wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wc.hIcon         = hiconApp;
     wc.lpszMenuName  = "MAIN";
     wc.hbrBackground =(HBRUSH)(COLOR_APPWORKSPACE + 1);
     wc.hIconSm       = hiconApp;

     RegisterClassEx(&wc);

     wc.lpszClassName = "CHILD";
     wc.lpfnWndProc   = DefWindowProc;
     wc.hCursor       = LoadCursor(NULL, IDC_IBEAM);
     wc.hIcon         = NULL;
     wc.lpszMenuName  = NULL;
     wc.hbrBackground =(HBRUSH)(COLOR_WINDOW + 1);
     wc.hIconSm       = NULL;

     RegisterClassEx(&wc);

     hwndMain =
     hwnd = CreateWindowEx(WS_EX_OVERLAPPEDWINDOW, "MAIN",
                           szAppName, WS_OVERLAPPEDWINDOW,
                           CW_USEDEFAULT, CW_USEDEFAULT,
                           CW_USEDEFAULT, CW_USEDEFAULT,
                           NULL, NULL, hInstance, NULL);
     ShowWindow(hwnd, cmdShow);
```

```
     UpdateWindow(hwnd);

     InitCommonControls();

     while(GetMessage(&msg, NULL, 0, 0))
          {
          if((hwndModeless) &&
             (!(PropSheet_IsDialogMessage(hwndModeless, &msg))))
              continue;

          TranslateMessage(&msg);
          DispatchMessage(&msg);
          }
     return msg.wParam;
     }

//-------------------------------------------------------------------
void MenuCheckMark(HMENU hmenu, int id, BOOL bCheck)
     {
     int iState;
     iState =(bCheck) ? MF_CHECKED : MF_UNCHECKED;
     CheckMenuItem(hmenu, id, iState);
     }

//-------------------------------------------------------------------
void static FlipFlag(LPDWORD dwStyle, DWORD flag)
     {
     if(*dwStyle & flag)  // Flag on -- turn off
          {
          *dwStyle &=(~flag);
          }
     else                  // Flag off -- turn on
          {
          *dwStyle |= flag;
          }
     }

//-------------------------------------------------------------------
LRESULT CALLBACK
WndProc(HWND hwnd, UINT mMsg, WPARAM wParam, LPARAM lParam)
     {
     switch(mMsg)
          {
          case WM_CREATE :
               hwndChild = CreateWindowEx(dwChildExStyle, "CHILD",
                                          "First Window", dwChildStyle,
                                          0, 0, 0, 0, hwnd,(HMENU) 1,
                                                     hInst, NULL);

               CreateWindowEx(dwChildExStyle, "CHILD", "Second Window",
                                WS_CLIPSIBLINGS | dwChildStyle,
                                10, 10, 200, 50, hwnd,
                               (HMENU) 2, hInst, NULL);
               return 0;

          case WM_COMMAND :
               {
               switch(LOWORD(wParam))
                    {
                    case IDM_WINDOW_PROPERTIES :
                         {
                         BOOL bRet;

                         // If modeless, active existing property sheet
```

```
                if(hwndModeless)
                     {
                     SetActiveWindow(hwndModeless);
                     break;
                     }

                // Are we creating a wizard?
                bWizard =(dwSheetStyles & PSH_WIZARD);

                // Create actual property sheet
                bRet = CreatePropertySheet(hwnd);

                // Store handle if modeless
                if(dwSheetStyles & PSH_MODELESS)
                     {
                     hwndModeless =(HWND) bRet;
                     break;
                     }
                break;
                }

           case IDM_WIZARD :
                FlipFlag(&dwSheetStyles, PSH_WIZARD);
                break;

           case IDM_HASHELP :
                FlipFlag(&dwSheetStyles, PSH_HASHELP);
                break;

           case IDM_MODELESS :
                FlipFlag(&dwSheetStyles, PSH_MODELESS);
                break;

           case IDM_NOAPPLYNOW :
                FlipFlag(&dwSheetStyles, PSH_NOAPPLYNOW);
                break;
           case IDM_PROPTITLE :
                FlipFlag(&dwSheetStyles, PSH_PROPTITLE);
                break;

           case IDM_RTLREADING :
                FlipFlag(&dwSheetStyles, PSH_RTLREADING);
                break;
           }
           return 0;
}

case WM_DESTROY :
     PostQuitMessage(0);
     return 0;

case WM_INITMENU :
     {
     BOOL bCheck;
     HMENU hmenu =(HMENU) wParam;

     bCheck =(dwSheetStyles & PSH_WIZARD);
     MenuCheckMark(hmenu, IDM_WIZARD, bCheck);

     bCheck =(dwSheetStyles & PSH_HASHELP);
     MenuCheckMark(hmenu, IDM_HASHELP, bCheck);

     bCheck =(dwSheetStyles & PSH_MODELESS);
     MenuCheckMark(hmenu, IDM_MODELESS, bCheck);
```

```
            bCheck =(dwSheetStyles & PSH_NOAPPLYNOW);
            MenuCheckMark(hmenu, IDM_NOAPPLYNOW, bCheck);

            bCheck =(dwSheetStyles & PSH_PROPTITLE);
            MenuCheckMark(hmenu, IDM_PROPTITLE, bCheck);

            bCheck =(dwSheetStyles & PSH_RTLREADING);
            MenuCheckMark(hmenu, IDM_RTLREADING, bCheck);

            return 0;
            }

        case WM_SETFOCUS :
            SetFocus(hwndChild);
            return 0;

        case WM_SIZE :
            {
            int cxWidth  = LOWORD(lParam);
            int cyHeight = HIWORD(lParam);
            int x, y, cx, cy;
            x  = cxWidth  / 4;
            cx = cxWidth  / 2;
            y  = cyHeight / 4;
            cy = cyHeight / 2;

            MoveWindow(hwndChild, x, y, cx, cy, TRUE);
            return 0;
            }

        case PM_CREATEWINDOW :
            {
            RECT rClient;
            LPARAM l;

            DestroyWindow(hwndChild);
            hwndChild = CreateWindowEx(dwChildExStyle, "CHILD",
                                       "First Window", dwChildStyle,
                                       0, 0, 0, 0, hwnd,(HMENU) 1,
                                                   hInst, NULL);

            // Send ourselves a WM_SIZE to resize child window
            GetClientRect(hwnd, &rClient);
            l = MAKELPARAM(rClient.right, rClient.bottom);
            SendMessage(hwnd, WM_SIZE, 0, l);
            return 0;
            }

        default :
            return(DefWindowProc(hwnd, mMsg, wParam, lParam));
        }
    }
```

SHEET.C

```
/*-----------------------------------------
   SHEET.C -- Property sheet page functions
              (c) Paul Yao, 1996
  -----------------------------------------*/
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include "comcthlp.h"
#include "property.h"
```

```
extern DWORD dwChildStyle;
extern DWORD dwChildExStyle;
extern DWORD dwSheetStyles;
extern char szAppName[];
extern HINSTANCE hInst;
extern HICON hiconApp;
//-------------------------------------------------------------------
int CALLBACK PropSheetProc(HWND hwndDlg, UINT uMsg, LPARAM lParam)
     {
     switch(uMsg)
          {
          case PSCB_INITIALIZED :
               // Process PSCB_INITIALIZED
               break;

          case PSCB_PRECREATE :
               // Process PSCB_PRECREATE
               break;

          default :
               // Unknown message
               break;
          }

     return 0;
     }

//-------------------------------------------------------------------
BOOL CreatePropertySheet(HWND hwndParent)
     {
     PROPSHEETHEADER pshead;
     PROPSHEETPAGE   pspage[2];

     // Initialize property sheet HEADER data
     ZeroMemory(&pshead, sizeof(PROPSHEETHEADER));
     pshead.dwSize  = sizeof(PROPSHEETHEADER);
     pshead.dwFlags = dwSheetStyles     |
                      PSH_PROPSHEETPAGE |
                      PSH_USECALLBACK   |
                      PSH_USEHICON;
     pshead.hwndParent  = hwndParent;
     pshead.hInstance   = hInst;
     pshead.hIcon       = hiconApp;
     pshead.pszCaption  = "A Child Window";
     pshead.nPages      = 2;
     pshead.nStartPage  = 0;
     pshead.ppsp        = pspage;
     pshead.pfnCallback = PropSheetProc;

     // Zero out property PAGE data
     ZeroMemory(&pspage, 2 * sizeof(PROPSHEETPAGE));

     // PAGE 1 -- window style page
     pspage[0].dwSize      = sizeof(PROPSHEETPAGE);
     pspage[0].dwFlags     = PSP_USECALLBACK | PSP_USEICONID;
     pspage[0].hInstance   = hInst;
     pspage[0].pszTemplate = MAKEINTRESOURCE(IDD_STYLES);
     pspage[0].pszIcon     = MAKEINTRESOURCE(IDI_PAGE1);
     pspage[0].pfnDlgProc  = StyleDlgProc;
     pspage[0].lParam      =(LPARAM) &dwChildStyle;
     pspage[0].pfnCallback = StylePageProc;

     // PAGE 2 -- extended window style page
```

```
     pspage[1].dwSize      = sizeof(PROPSHEETPAGE);
     pspage[1].dwFlags     = PSP_USECALLBACK | PSP_USEICONID |
                             PSP_HASHELP;
     pspage[1].hInstance   = hInst;
     pspage[1].pszTemplate = MAKEINTRESOURCE(IDD_EXSTYLES);
     pspage[1].pszIcon     = MAKEINTRESOURCE(IDI_PAGE2);
     pspage[1].pfnDlgProc  = ExStyleDlgProc;
     pspage[1].lParam      =(LPARAM) &dwChildExStyle;
     pspage[1].pfnCallback = ExStylePageProc;

     // --------- Create & display property sheet ---------

     return PropertySheet(&pshead);
     }
```

PAGE1.C

```
/*---------------------------------
   PAGE1.C -- Property sheet page 1
             (c) Paul Yao, 1996
  ---------------------------------*/
#include <windows.h>
#include <prsht.h>
#include "property.h"
#include "notify.h"
#include "helper.h"

static LPDWORD pTheStyles;
extern BOOL bWizard;
extern HWND hwndMain;
extern HWND hwndModeless;

DWORD FetchStyles(HWND hwndDlg);

//-------------------------------------------------------------------
UINT CALLBACK
StylePageProc(HWND  hwnd, UINT uMsg, LPPROPSHEETPAGE ppsp)
     {
     switch(uMsg)
          {
          case PSPCB_CREATE :
               // Store pointer to style data
               pTheStyles =(LPDWORD) ppsp->lParam;
               return TRUE;
          case PSPCB_RELEASE :
               return 0;
          }

     return 0;
     }

//-------------------------------------------------------------------
BOOL CALLBACK
StyleDlgProc(HWND hwndDlg, UINT msg, WPARAM wParam, LPARAM lParam)
     {
     switch(msg)
          {
          case WM_INITDIALOG :
               {
               BOOL bCheck;
               DWORD dwOrigStyle = *pTheStyles;

               bCheck =(dwOrigStyle & WS_VISIBLE);
               SetButtonCheck(hwndDlg, IDC_VISIBLE, bCheck);
```

```
            bCheck =(dwOrigStyle & WS_DISABLED);
            SetButtonCheck(hwndDlg, IDC_DISABLED, bCheck);

            bCheck =(dwOrigStyle & WS_MINIMIZE);
            SetButtonCheck(hwndDlg, IDC_MINIMIZE, bCheck);

            bCheck =(dwOrigStyle & WS_MAXIMIZE);
            SetButtonCheck(hwndDlg, IDC_MAXIMIZE, bCheck);

            bCheck =(dwOrigStyle & WS_CLIPCHILDREN);
            SetButtonCheck(hwndDlg, IDC_CLIPCHILDREN, bCheck);

            bCheck =(dwOrigStyle & WS_CLIPSIBLINGS);
            SetButtonCheck(hwndDlg, IDC_CLIPSIBLINGS, bCheck);

            bCheck =(dwOrigStyle & WS_BORDER);
            SetButtonCheck(hwndDlg, IDC_BORDER, bCheck);

            bCheck =(dwOrigStyle & WS_CAPTION);
            SetButtonCheck(hwndDlg, IDC_CAPTION, bCheck);

            bCheck =(dwOrigStyle & WS_DLGFRAME);
            SetButtonCheck(hwndDlg, IDC_DLGFRAME, bCheck);

            bCheck =(dwOrigStyle & WS_HSCROLL);
            SetButtonCheck(hwndDlg, IDC_HSCROLL, bCheck);

            bCheck =(dwOrigStyle & WS_MAXIMIZEBOX);
            SetButtonCheck(hwndDlg, IDC_MAXIMIZEBOX, bCheck);
            bCheck =(dwOrigStyle & WS_MINIMIZEBOX);
            SetButtonCheck(hwndDlg, IDC_MINIMIZEBOX, bCheck);

            bCheck =(dwOrigStyle & WS_SYSMENU);
            SetButtonCheck(hwndDlg, IDC_SYSMENU, bCheck);

            bCheck =(dwOrigStyle & WS_THICKFRAME);
            SetButtonCheck(hwndDlg, IDC_THICKFRAME, bCheck);

            bCheck =(dwOrigStyle & WS_VSCROLL);
            SetButtonCheck(hwndDlg, IDC_VSCROLL, bCheck);

            return TRUE;
        }

        case WM_COMMAND :
            {
            WORD wNotifyCode = HIWORD(wParam);
            WORD wID = LOWORD(wParam);
            HWND hwndSheet;

            switch(wID)
                {
                case IDC_VISIBLE :
                case IDC_DISABLED :
                case IDC_MINIMIZE :
                case IDC_MAXIMIZE :
                case IDC_CLIPCHILDREN :
                case IDC_CLIPSIBLINGS :
                case IDC_BORDER :
                case IDC_CAPTION :
                case IDC_DLGFRAME :
                case IDC_HSCROLL :
                case IDC_MAXIMIZEBOX :
                case IDC_MINIMIZEBOX :
```

```
            case IDC_SYSMENU :
            case IDC_THICKFRAME :
            case IDC_VSCROLL :
                 hwndSheet = GetParent(hwndDlg);
                 PropSheet_Changed(hwndSheet, hwndDlg);
                 break;
            }
       return TRUE;
       }

  case WM_HELP :
       // Catch F1 key strike
       MessageBox(hwndDlg, "WM_HELP Message Received",
                  "StyleDlgProc", MB_OK);
       return TRUE;
  case WM_NOTIFY :
       {
       LPNMHDR pnmh =(LPNMHDR) lParam;

       // Handle OK and Apply buttons
       if(pnmh->code == PSN_APPLY)
            {
            HWND hwndPS;
            HWND hwndActive;

            // Overwrite current style value
            *pTheStyles = FetchStyles(hwndDlg);

            // Tell main window to re-create child window
            hwndPS = GetParent(hwndDlg);
            hwndActive = PropSheet_GetCurrentPageHwnd(hwndPS);

            // Only re-create if we're the active page
            if(hwndDlg == hwndActive)
                 PostMessage(hwndMain, PM_CREATEWINDOW, 0, 0L);
            }

       // Destroy modeless dialog on OK or Cancel
       if((IsWindowEnabled(hwndMain)) &&
          (pnmh->code == PSN_APPLY || pnmh->code == PSN_RESET))
            {
            LPPSHNOTIFY psh =(LPPSHNOTIFY) lParam;
            HWND hwndPropSheet;

            // Ignore Apply button
            if(pnmh->code == PSN_APPLY && psh->lParam == 0)
                 return TRUE;

            // Clicking OK or Cancel, destroy property sheet
            hwndPropSheet = GetParent(hwndDlg);
            DestroyWindow(hwndPropSheet);
            hwndModeless = NULL;
            }

       // Enable Next button on wizard page
       if(bWizard && pnmh->code == PSN_SETACTIVE)
            {
            HWND hwndSheet = GetParent(hwndDlg);
            PropSheet_SetWizButtons(hwndSheet, PSWIZB_NEXT);
            }
       return TRUE;
       }

  default :
```

```
                return FALSE;
           }
      }

//-----------------------------------------------------------------------
DWORD FetchStyles(HWND hwndDlg)
     {
     DWORD dwStyle = WS_CHILD;

     if(QueryButtonCheck(hwndDlg, IDC_VISIBLE))
          {
          dwStyle |= WS_VISIBLE;
          }

     if(QueryButtonCheck(hwndDlg, IDC_DISABLED))
          {
          dwStyle |= WS_DISABLED;
          }

     if(QueryButtonCheck(hwndDlg, IDC_MINIMIZE))
          {
          dwStyle |= WS_MINIMIZE;
          }

     if(QueryButtonCheck(hwndDlg, IDC_MAXIMIZE))
          {
          dwStyle |= WS_MAXIMIZE;
          }

     if(QueryButtonCheck(hwndDlg, IDC_CLIPCHILDREN))
          {
          dwStyle |= WS_CLIPCHILDREN;
          }

     if(QueryButtonCheck(hwndDlg, IDC_CLIPSIBLINGS))
          {
          dwStyle |= WS_CLIPSIBLINGS;
          }

     if(QueryButtonCheck(hwndDlg, IDC_BORDER))
          {
          dwStyle |= WS_BORDER;
          }

     if(QueryButtonCheck(hwndDlg, IDC_CAPTION))
          {
          dwStyle |= WS_CAPTION;
          }

     if(QueryButtonCheck(hwndDlg, IDC_DLGFRAME))
          {
          dwStyle |= WS_DLGFRAME;
          }

     if(QueryButtonCheck(hwndDlg, IDC_HSCROLL))
          {
          dwStyle |= WS_HSCROLL;
          }

     if(QueryButtonCheck(hwndDlg, IDC_MAXIMIZEBOX))
          {
          dwStyle |= WS_MAXIMIZEBOX;
          }
```

```
     if(QueryButtonCheck(hwndDlg, IDC_MINIMIZEBOX))
          {
          dwStyle |= WS_MINIMIZEBOX;
          }

     if(QueryButtonCheck(hwndDlg, IDC_SYSMENU))
          {
          dwStyle |= WS_SYSMENU;
          }

     if(QueryButtonCheck(hwndDlg, IDC_THICKFRAME))
          {
          dwStyle |= WS_THICKFRAME;
          }

     if(QueryButtonCheck(hwndDlg, IDC_VSCROLL))
          {
          dwStyle |= WS_VSCROLL;
          }

     return dwStyle;
     }
```

PAGE2.C

```
/*-----------------------------------
   PAGE2.C -- Property sheet page 2
             (c) Paul Yao, 1996
  -----------------------------------*/
#include <windows.h>
#include <commctrl.h>
#include "property.h"
#include "notify.h"
#include "helper.h"

static LPDWORD pTheExStyles;
extern BOOL bWizard;
extern HWND hwndMain;
extern HWND hwndModeless;

DWORD FetchExStyles(HWND hwndDlg);

//-------------------------------------------------------------------
UINT CALLBACK
ExStylePageProc(HWND  hwnd, UINT uMsg, LPPROPSHEETPAGE ppsp)
     {
     switch(uMsg)
          {
          case PSPCB_CREATE :
               // Store pointer to extended style data
               pTheExStyles =(LPDWORD) ppsp->lParam;
               return TRUE;

          case PSPCB_RELEASE :
               break;
          }

     return 0;
     }

//-------------------------------------------------------------------
BOOL CALLBACK
ExStyleDlgProc(HWND hwndDlg, UINT msg, WPARAM wParam, LPARAM lParam)
     {
     switch(msg)
```

```
{
case WM_INITDIALOG :
     {
     BOOL bCheck;
     DWORD dwOrigStyle = *pTheExStyles;

     bCheck =(dwOrigStyle & WS_EX_CLIENTEDGE);
     SetButtonCheck(hwndDlg, IDC_CLIENTEDGE, bCheck);

     bCheck =(dwOrigStyle & WS_EX_CONTEXTHELP);
     SetButtonCheck(hwndDlg, IDC_CONTEXTHELP, bCheck);

     bCheck =(dwOrigStyle & WS_EX_DLGMODALFRAME);
     SetButtonCheck(hwndDlg, IDC_DLGMODALFRAME, bCheck);

     bCheck =(dwOrigStyle & WS_EX_OVERLAPPEDWINDOW);
     SetButtonCheck(hwndDlg, IDC_EXOVERLAPPED, bCheck);

     bCheck =(dwOrigStyle & WS_EX_PALETTEWINDOW);
     SetButtonCheck(hwndDlg, IDC_PALETTE, bCheck);

     bCheck =(dwOrigStyle & WS_EX_STATICEDGE);
     SetButtonCheck(hwndDlg, IDC_STATICEDGE, bCheck);
     bCheck =(dwOrigStyle & WS_EX_TOOLWINDOW);
     SetButtonCheck(hwndDlg, IDC_TOOLWINDOW, bCheck);

     bCheck =(dwOrigStyle & WS_EX_WINDOWEDGE);
     SetButtonCheck(hwndDlg, IDC_WINDOWEDGE, bCheck);

     bCheck =(dwOrigStyle & WS_EX_ACCEPTFILES);
     SetButtonCheck(hwndDlg, IDC_ACCEPTFILES, bCheck);

     bCheck =(dwOrigStyle & WS_EX_APPWINDOW);
     SetButtonCheck(hwndDlg, IDC_APPWINDOW, bCheck);

     bCheck =(dwOrigStyle & WS_EX_TOPMOST);
     SetButtonCheck(hwndDlg, IDC_TOPMOST, bCheck);

     bCheck =(dwOrigStyle & WS_EX_TRANSPARENT);
     SetButtonCheck(hwndDlg, IDC_TRANSPARENT, bCheck);

     bCheck =(dwOrigStyle & WS_EX_CONTROLPARENT);
     SetButtonCheck(hwndDlg, IDC_CONTROLPARENT, bCheck);

     bCheck =(dwOrigStyle & WS_EX_MDICHILD);
     SetButtonCheck(hwndDlg, IDC_MDICHILD, bCheck);

     bCheck =(dwOrigStyle & WS_EX_NOPARENTNOTIFY);
     SetButtonCheck(hwndDlg, IDC_NOPARENTNOTIFY, bCheck);

     return TRUE;
     }

case WM_COMMAND :
     {
     WORD wNotifyCode = HIWORD(wParam);
     WORD wID = LOWORD(wParam);
     HWND hwndSheet;

     switch(wID)
          {
          case IDC_CLIENTEDGE :
          case IDC_CONTEXTHELP :
          case IDC_DLGMODALFRAME :
```

```
            case IDC_EXOVERLAPPED :
            case IDC_PALETTE :
            case IDC_STATICEDGE :
            case IDC_TOOLWINDOW :
            case IDC_WINDOWEDGE :
            case IDC_ACCEPTFILES :
            case IDC_APPWINDOW :
            case IDC_TOPMOST :
            case IDC_TRANSPARENT :
            case IDC_CONTROLPARENT :
            case IDC_MDICHILD :
            case IDC_NOPARENTNOTIFY :
                 hwndSheet = GetParent(hwndDlg);
                 PropSheet_Changed(hwndSheet, hwndDlg);
                 break;
            }
       return TRUE;
       }

  case WM_HELP :
       // Catch F1 key strike
       MessageBox(hwndDlg, "WM_HELP Message Received",
                   "ExStyleDlgProc", MB_OK);
       return TRUE;

  case WM_NOTIFY :
       {
       LPNMHDR pnmh =(LPNMHDR) lParam;

        // Handle Finish button on wizard page
                  if(pnmh->code == PSN_WIZFINISH)
                       {
                       HWND hwndPS;

                       hwndPS = GetParent(hwndDlg);
                       PropSheet_Apply(hwndPS);
                       return TRUE;
                       }

       // Handle OK and Apply buttons
       if(pnmh->code == PSN_APPLY || pnmh->code == PSN_RESET)
            {
            HWND hwndPS;
            HWND hwndActive;

            // Overwrite current style value
            *pTheExStyles = FetchExStyles(hwndDlg);

            // Tell main window to re-create child window
            hwndPS = GetParent(hwndDlg);
            hwndActive = PropSheet_GetCurrentPageHwnd(hwndPS);

            // Only re-create if we're the active page
            if(hwndDlg == hwndActive)
                 PostMessage(hwndMain, PM_CREATEWINDOW, 0, 0L);
            }

       // Destroy modeless dialog on OK or Cancel
       if((IsWindowEnabled(hwndMain)) &&
          (pnmh->code == PSN_APPLY || pnmh->code == PSN_RESET))
            {
            LPPSHNOTIFY psh =(LPPSHNOTIFY) lParam;
            HWND hwndPropSheet;
```

```
                    // Ignore Apply button
                    if(pnmh->code == PSN_APPLY && psh->lParam == 0)
                         return TRUE;

                    // Clicking OK or Cancel, destroy property sheet
                    hwndPropSheet = GetParent(hwndDlg);
                    DestroyWindow(hwndPropSheet);
                    hwndModeless = NULL;
               }

               // Enable Back and Finish buttons on wizard page
               if(bWizard && pnmh->code == PSN_SETACTIVE)
                    {
                    HWND hwndSheet = GetParent(hwndDlg);
                    PropSheet_SetWizButtons(hwndSheet, PSWIZB_BACK |
                                                    PSWIZB_FINISH);
                    }

               // Support for Help button
               if(pnmh->code == PSN_HELP)
                    {
                    MessageBox(hwndDlg, "PSN_HELP Notification Received",
                                "ExStyleDlgProc", MB_OK);
                    }

               return TRUE;
               }

          default :
               return FALSE;
          }
     }

//--------------------------------------------------------------------
DWORD FetchExStyles(HWND hwndDlg)
     {
     DWORD dwStyle = 0L;

     if(QueryButtonCheck(hwndDlg, IDC_CLIENTEDGE))
          {
          dwStyle |= WS_EX_CLIENTEDGE;
          }

     if(QueryButtonCheck(hwndDlg, IDC_CONTEXTHELP))
          {
          dwStyle |= WS_EX_CONTEXTHELP;
          }

     if(QueryButtonCheck(hwndDlg, IDC_DLGMODALFRAME))
          {
          dwStyle |= WS_EX_DLGMODALFRAME;
          }

     if(QueryButtonCheck(hwndDlg, IDC_EXOVERLAPPED))
          {
          dwStyle |= WS_EX_OVERLAPPEDWINDOW;
          }

     if(QueryButtonCheck(hwndDlg, IDC_PALETTE))
          {
          dwStyle |= WS_EX_PALETTEWINDOW;
          }

     if(QueryButtonCheck(hwndDlg, IDC_STATICEDGE))
```

```
          {
          dwStyle |= WS_EX_STATICEDGE;
          }

     if(QueryButtonCheck(hwndDlg, IDC_TOOLWINDOW))
          {
          dwStyle |= WS_EX_TOOLWINDOW;
          }

     if(QueryButtonCheck(hwndDlg, IDC_WINDOWEDGE))
          {
          dwStyle |= WS_EX_WINDOWEDGE;
          }

     if(QueryButtonCheck(hwndDlg, IDC_ACCEPTFILES))
          {
          dwStyle |= WS_EX_ACCEPTFILES;
          }

     if(QueryButtonCheck(hwndDlg, IDC_APPWINDOW))
          {
          dwStyle |= WS_EX_APPWINDOW;
          }

     if(QueryButtonCheck(hwndDlg, IDC_TOPMOST))
          {
          dwStyle |= WS_EX_TOPMOST;
          }

     if(QueryButtonCheck(hwndDlg, IDC_TRANSPARENT))
          {
          dwStyle |= WS_EX_TRANSPARENT;
          }

     if(QueryButtonCheck(hwndDlg, IDC_CONTROLPARENT))
          {
          dwStyle |= WS_EX_CONTROLPARENT;
          }

     if(QueryButtonCheck(hwndDlg, IDC_MDICHILD))
          {
          dwStyle |= WS_EX_MDICHILD;
          }

     if(QueryButtonCheck(hwndDlg, IDC_NOPARENTNOTIFY))
          {
          dwStyle |= WS_EX_NOPARENTNOTIFY;
          }


     return dwStyle;
     }
```

HELPER.C

```
/*--------------------------------
   HELPER.C -- Helper routines
              (c) Paul Yao, 1996
  --------------------------------*/
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include "property.h"
```

```
//-------------------------------------------------------------------
void SetButtonCheck(HWND hwndDlg, int CtrlID, BOOL bCheck)
     {
     HWND hwndCtrl = GetDlgItem(hwndDlg, CtrlID);
     if(bCheck)
          {
          Button_SetCheck(hwndCtrl, BST_CHECKED);
          }
     }

//-------------------------------------------------------------------
BOOL QueryButtonCheck(HWND hwndDlg, int CtrlID)
     {
     HWND hwndCtrl = GetDlgItem(hwndDlg, CtrlID);
     int  nCheck = Button_GetCheck(hwndCtrl);
     return(nCheck == BST_CHECKED);
     }
```

PROPERTY.RC

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

IDI_APP                 ICON    DISCARDABLE     "PROPERTY.ICO"
IDI_PAGE1               ICON    DISCARDABLE     "page1.ico"
IDI_PAGE2               ICON    DISCARDABLE     "page2.ico"

MAIN MENU DISCARDABLE
     {
     POPUP "&Property-Sheet-Styles"
          {
          MENUITEM "PSH_WIZARD",                 IDM_WIZARD
          MENUITEM SEPARATOR
          MENUITEM "PSH_HASHELP",                IDM_HASHELP
          MENUITEM "PSH_MODELESS",               IDM_MODELESS
          MENUITEM "PSH_NOAPPLYNOW",             IDM_NOAPPLYNOW
          MENUITEM "PSH_PROPTITLE",              IDM_PROPTITLE
          MENUITEM "PSH_RTLREADING",             IDM_RTLREADING
          }
     POPUP "&Window"
          {
          MENUITEM "Properties",                 IDM_WINDOW_PROPERTIES
          }
     }

IDD_STYLES DIALOG DISCARDABLE  0, 0, 292, 127
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Window Styles"
FONT 8, "MS Sans Serif"
     {
     GROUPBOX        "&Initial State",IDC_STATIC,7,7,94,69
     CONTROL         "WS_VISIBLE",IDC_VISIBLE,"Button",BS_AUTOCHECKBOX |
                     WS_TABSTOP,14,21,55,10
     CONTROL         "WS_DISABLED",IDC_DISABLED,"Button",BS_AUTOCHECKBOX |
                     WS_TABSTOP,14,35,64,10
     CONTROL         "WS_MINIMIZE",IDC_MINIMIZE,"Button",BS_AUTOCHECKBOX |
                     WS_TABSTOP,14,49,60,10
     CONTROL         "WS_MAXIMIZE",IDC_MAXIMIZE,"Button",BS_AUTOCHECKBOX |
                     WS_TABSTOP,14,63,62,10
     GROUPBOX        "&Clipping",IDC_STATIC,7,80,94,42
     CONTROL         "WS_CLIPCHILDREN",IDC_CLIPCHILDREN,"Button",
                     BS_AUTOCHECKBOX | WS_TABSTOP,14,94,79,10
     CONTROL         "WS_CLIPSIBLINGS",IDC_CLIPSIBLINGS,"Button",
                     BS_AUTOCHECKBOX | WS_TABSTOP,14,108,75,10
     GROUPBOX        "&Window Components",IDC_STATIC,115,7,170,98
```

```
    CONTROL         "WS_BORDER",IDC_BORDER,"Button",BS_AUTOCHECKBOX |
                    WS_TABSTOP,123,21,60,10
    CONTROL         "WS_CAPTION",IDC_CAPTION,"Button",BS_AUTOCHECKBOX |
                    WS_TABSTOP,123,35,60,10
    CONTROL         "WS_DLGFRAME",IDC_DLGFRAME,"Button",BS_AUTOCHECKBOX |
                    WS_TABSTOP,123,49,68,10
    CONTROL         "WS_HSCROLL",IDC_HSCROLL,"Button",BS_AUTOCHECKBOX |
                    WS_TABSTOP,123,63,71,10
    CONTROL         "WS_MAXIMIZEBOX",IDC_MAXIMIZEBOX,"Button",
                    BS_AUTOCHECKBOX | WS_TABSTOP,123,77,93,10
    CONTROL         "WS_MINIMIZEBOX",IDC_MINIMIZEBOX,"Button",
                    BS_AUTOCHECKBOX | WS_TABSTOP,123,91,90,10
    CONTROL         "WS_SYSMENU",IDC_SYSMENU,"Button",BS_AUTOCHECKBOX |
                    WS_TABSTOP,200,21,72,10
    CONTROL         "WS_THICKFRAME",IDC_THICKFRAME,"Button",BS_AUTOCHECKBOX |
                    WS_TABSTOP,200,35,77,10
    CONTROL         "WS_VSCROLL",IDC_VSCROLL,"Button",BS_AUTOCHECKBOX |
                    WS_TABSTOP,200,49,74,10
    }

IDD_EXSTYLES DIALOG DISCARDABLE  0, 0, 330, 161
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Extended Window Styles"
FONT 8, "MS Sans Serif"
    {
    GROUPBOX        "&Non-Client Area",IDC_STATIC,7,7,263,71
    CONTROL         "WS_EX_CLIENTEDGE",IDC_CLIENTEDGE,"Button",
                    BS_AUTOCHECKBOX | WS_TABSTOP,14,21,113,10
    CONTROL         "WS_EX_CONTEXTHELP",IDC_CONTEXTHELP,"Button",
                    BS_AUTOCHECKBOX | WS_TABSTOP,14,35,116,10
    CONTROL         "WS_EX_DLGMODALFRAME",IDC_DLGMODALFRAME,"Button",
                    BS_AUTOCHECKBOX | WS_TABSTOP,14,49,121,10
    CONTROL         "WS_EX_OVERLAPPEDWINDOW",IDC_EXOVERLAPPED,"Button",
                    BS_AUTOCHECKBOX | WS_TABSTOP,14,63,127,10
    CONTROL         "WS_EX_PALETTEWINDOW",IDC_PALETTE,"Button",
                    BS_AUTOCHECKBOX | WS_TABSTOP,149,21,113,10
    CONTROL         "WS_EX_STATICEDGE",IDC_STATICEDGE,"Button",
                    BS_AUTOCHECKBOX | WS_TABSTOP,149,35,102,10
    CONTROL         "WS_EX_TOOLWINDOW",IDC_TOOLWINDOW,"Button",
                    BS_AUTOCHECKBOX | WS_TABSTOP,149,49,101,10
    CONTROL         "WS_EX_WINDOWEDGE",IDC_WINDOWEDGE,"Button",
                    BS_AUTOCHECKBOX | WS_TABSTOP,149,63,102,10
    GROUPBOX        "&Shell Interface",IDC_STATIC,8,80,214,39
    CONTROL         "WS_EX_ACCEPTFILES",IDC_ACCEPTFILES,"Button",
                    BS_AUTOCHECKBOX | WS_TABSTOP,14,90,97,10
    CONTROL         "WS_EX_APPWINDOW",IDC_APPWINDOW,"Button",
                        BS_AUTOCHECKBOX | WS_TABSTOP,14,104,96,10
    CONTROL         "WS_EX_TOPMOST",IDC_TOPMOST,"Button",BS_AUTOCHECKBOX |
                    WS_TABSTOP,118,90,88,10
    CONTROL         "WS_EX_TRANSPARENT",IDC_TRANSPARENT,"Button",
                    BS_AUTOCHECKBOX | WS_TABSTOP,118,104,102,10
    GROUPBOX        "&Child Window",IDC_STATIC,7,121,316,33
    CONTROL         "WS_EX_CONTROLPARENT",IDC_CONTROLPARENT,"Button",
                    BS_AUTOCHECKBOX | WS_TABSTOP,14,135,105,10
    CONTROL         "WS_EX_MDICHILD",IDC_MDICHILD,"Button",BS_AUTOCHECKBOX |
                    WS_TABSTOP,124,135,77,10
    CONTROL         "WS_EX_NOPARENTNOTIFY",IDC_NOPARENTNOTIFY,"Button",
                    BS_AUTOCHECKBOX | WS_TABSTOP,206,135,111,10
    }
```

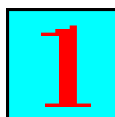
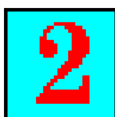
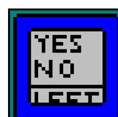
PAGE1.ICO	**PAGE2.ICO**	**PROPERTY.ICO**
		

Рис. 12.13 Исходный текст программы PROPERTY

В ответ на сообщение WM_CREATE главное окно программы PROPERTY создает два дочерних окна. Первое окно с заголовком "First Window" представляет для нас основной интерес. Это окно изменяет размеры при изменении размеров родительского окна с тем, чтобы занимать 1/9 от размера его рабочей области. При изменении флагов стиля окна, родительское окно получает частное сообщение PM_CREATEWINDOW, в ответ на которое родительское окно уничтожает открытое дочернее окно, создает новое дочернее окно с указанными стилями, а затем посылает самому себе сообщение WM_SIZE для форсирования изменения размеров дочернего окна.

Флаги стиля для создания дочернего окна запоминаются в двух глобальных переменных, определенных в файле PROPERTY.C:

```
DWORD dwChildStyle = WS_CHILD | WS_VISIBLE | WS_BORDER | WS_CAPTION;
DWORD dwChildExStyle = 0L;
```

Указатели на каждую из этих переменных передаются каждой странице свойств в поле *lParam* структуры PRPSHEETPAGE. Например, ниже показано, как *dwChildStyle* передается первой странице свойств (см. файл SHEET.C):

```
pspage[0].lParam =(LPARAM) &dwChildStyle;
```

Первая страница свойств извлекает это значение в своей оконной процедуре (*StylePageProc*). Расположенная в файле PAGE1.C, эта функция вызывается непосредственно перед созданием страницы свойств с кодом сообщения PSPCB_CREATE. Этот указатель запоминается в глобальной переменной, объявленной как *static*, и поэтому, имеющей область видимости, ограниченную этим исходным файлом:

```
// Сохранить указатель на данные о стиле
pTheStyles =(LPDWORD) ppsp->lParam;
```

Здесь *ppsp* — указатель на структуру страницы свойств PROPSHEETPAGE, переданную в функцию страницы свойств.

Каждая страница свойств имеет указатель на данные, которые могут быть изменены в ней. Поэтому, когда становится необходимым применить сделанные изменения, каждая диалоговая процедура страницы свойств записывает эти изменения обратно в источник данных. Кажется, что было бы проще использовать глобальные переменные. Это общая практика при работе с обычными диалоговыми окнами. Но мы используем параметр *lParam* для того, чтобы показать, как "удаленная" страница свойств — такая как расположенная в динамически подключаемой библиотеке — может быть инициализирована без использования чего-либо кроме содержимого структуры PROPSHEETPAGE.

Внутри диалоговых процедур страниц свойств наиболее интересным является обработка сообщения WM_NOTIFY. При нажатии кнопок OK или Apply диалоговые процедуры получают уведомление PSN_APPLY. Как уже упоминалось ранее, страницы свойств создаются по мере их активизации пользователем. Только страницы, которые были реально созданы, получают уведомления. Это означает, что когда набор страниц свойств уничтожается, только одна или обе диалоговые процедуры могут получить уведомление PSN_APPLY.

Для того, чтобы быть уверенным в том, что изменения данных будут применены только один раз, каждая диалоговая процедура проверяет, являлась ли ее страница активной перед посылкой запроса PM_CREATEWINDOW главному окну программы. Это делается с помощью посылки сообщения PSM_GETCURRENTPAGEHWND окну набора страниц свойств, в ответ на которое возвращается описатель окна активной страницы. Запрос на воссоздание дочернего окна посылается функцией *PostMessage* с целью избежать временных проблем при посылке сообщения из диалоговой процедуры активной страницы:

```
HWND hwndPS = GetParent(hwndDlg);
HWND hwndActive = PropSheet_GetCurrentPageHwnd(hwndPS);

// Только для активной страницы
if(hwndDlg == hwndActive)
       PostMessage(hwndMain, PM_CREATEWINDOW, 0, 0L);
```

Программа PROPERTY показывает, что все, что вы знаете о работе с диалоговыми окнами применимо к наборам страниц свойств. Имеется несколько различий, связанных с большей сложностью. Это касается мастеров, немодальных наборов страниц свойств и использования различных наборов кнопок.