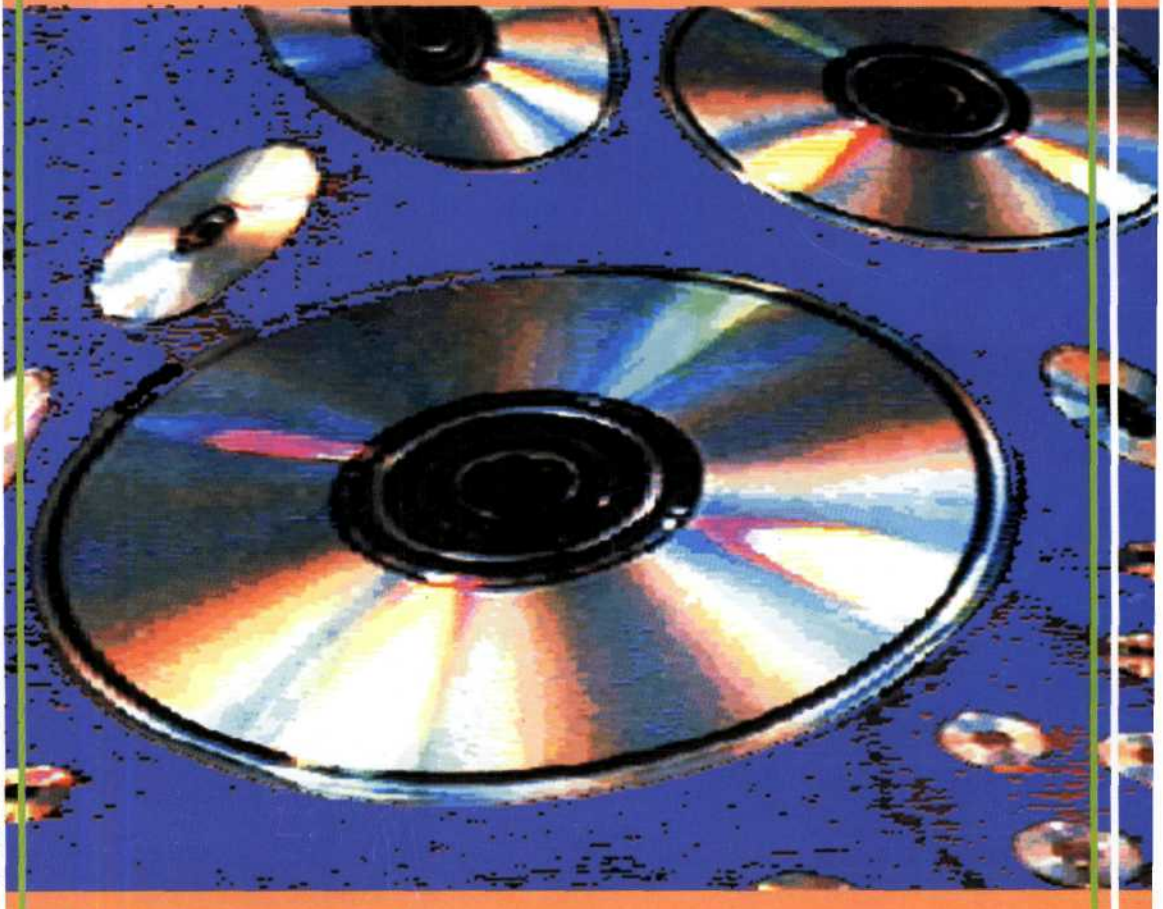
К. Г. ФИНОГЕНОВ

WIN32

ОСНОВЫ ПРОГРАММИРОВАНИЯ



ДИАЛОГ-МИФИ

К. Г. Финогенов

WIN32
ОСНОВЫ ПРОГРАММИРОВАНИЯ

Издание второе, исправленное и дополненное

МОСКВА ДИАЛОГ-МИФИ 2006

УДК 32.973.1
ББК 681.3
Ф59

Финогенов К. Г.

Ф59 Win32. Основы программирования. – 2-е изд., испр. и дополн. – М.: ДИАЛОГ-МИФИ, 2006. – 416 с.

ISBN 5-86404-173-4

Книга является простым и доступным для широкого круга читателей пособием по разработке 32-разрядных программ для систем Windows на языке C++. Рассмотрены основные особенности архитектуры защищенного режима процессоров Intel, дано введение в язык C++, описаны принципы составления прикладных программ для Windows. Особое внимание уделяется новым средствам, включенным в Win32, – потокам, процессам, синхронизации, передаче данных, отображению файлов в память и др.

Книга предназначена главным образом для начинающих программистов, студентов вузов, аспирантов и преподавателей, однако может быть полезна и опытным программистам, желающим освоить специальные средства Win32.

ББК 32.973.1

Учебно-справочное издание

Финогенов Кирилл Григорьевич

Win32. Основы программирования

Издание второе, исправленное и дополненное

Редактор О. А. Голубев
Корректор В. С. Кустов
Макет Н. В. Дмитриевой

Подписано в печать 12.12.2005.
Формат 70x100/16. Бум. газетная. Печать офс. Гарнитура Таймс.
Усл. печ. л. 33,54. Уч.-изд. л. 25,03. Тираж 2 000 экз. Заказ 5532.

ЗАО "ДИАЛОГ-МИФИ", ООО "Д и М"
115409, Москва, ул. Москворечье, 31, корп. 2. Т.: 320-43-55, 320-43-77
Http://www.dialog-mifi.ru E-mail:zakaz@dialog-mifi.ru

Отпечатано в ОАО ордена Трудового Красного Знамени
«Чеховский полиграфический комбинат»
142300 г. Чехов Московской области
Т/ф (501) 443-92-17, т/ф (272) 6-25-36
E-mail: chpk_marketing@chehov.ru

ISBN 5-86404-173-4

Предисловие

Эта книга рассчитана на начинающих программистов, которые хотели бы за короткое время освоить принципы разработки прикладных программ, выполняемых в среде операционных систем Windows. В настоящее время те или иные варианты системы Windows используются на большинстве персональных компьютеров, и при разработке нового программного обеспечения вполне естественно ориентироваться именно на эту операционную систему.

Современные графические операционные системы, к которым принадлежит и система Windows, предоставляют прикладному программисту много удобных и полезных возможностей: средства управления в виде меню, диалоговых окон, линеек прокрутки, списков, кнопок и прочих инструментов; сопровождение работы программы выводом цветных графических изображений и музыкальных фрагментов; средства организации контекстных справочников и системы подсказок и многое другое.

С другой стороны, разработка приложений Windows оказывается довольно сложным делом. Программист должен не только достаточно свободно владеть огромным арсеналом изобразительных средств Windows, но и хорошо понимать многочисленные внутренние концепции этой системы. В то время как сервис, предоставляемый пользователю системой MS-DOS, – обслуживание файловой системы, запуск и завершение задач, работа с векторами прерываний и др. – вполне естественно вытекает из общих особенностей архитектуры и функционирования компьютера, понятия, используемые в системе Windows, – очереди сообщений, файлы ресурсов, контексты устройств, дескрипторы объектов и т. д. – совсем не столь очевидны. В настоящей книге сделана попытка изложения на базе относительно простых и имеющих прикладной характер примеров основных концепций Windows и методики разработки программ, предназначенных для работы под управлением этой системы.

Как известно, для системы Windows можно разрабатывать как 16-разрядные, так и 32-разрядные программы; 32-разрядную операционную среду Windows, в которой выполняются 32-разрядные программы, называют Win32, а сами программы – Win32-приложениями; соответственно для 16-разрядных программ иногда используют обозначение Win16.

Не следует думать, что 16- и 32-разрядные приложения должны обязательно сильно отличаться друг от друга. Бóльшая часть базовых средств Windows, служащих для организации таких непременных элементов интерфейса пользователя, как окна, меню, диалоги и пр., одинаково используется в программах обоих типов (впрочем, для многих функций в Win32 имеются расширенные варианты). Поэтому простую Windows-программу, применяющую лишь эти базовые средства, можно с равным успехом оформить и как 16-разрядное, и как 32-разрядное приложение, для чего достаточно при построении приложения в интегрированной среде разработки установить соответствующий флажок. Исходные тексты двух таких вариантов, как и их функционирование, будут полностью идентичны, хотя машинные коды этих программ, т. е. состав загрузочных модулей, как и процесс их выполнения, будут различаться довольно существенно.

Однако Win32-приложения обладают целым рядом несомненных достоинств. Операционная среда Win32 позволяет работать с огромными массивами данных, создавать иерархические программные комплексы, организовывать параллельное выполнение многих процессов со взаимной передачей данных и синхронизацией. В ряде случаев (например,

если программа интенсивно работает с файлами на магнитных дисках) 32-разрядная программа оказывается проще 16-разрядной. В технической документации законодателя мод в системном программном обеспечении – корпорации Microsoft декларируется, что вновь разрабатываемые приложения должны быть только 32-разрядными.

В настоящей книге, адресуемой главным образом начинающему программисту, уделяется внимание как базовым средствам программирования, так и средствам, допступным только в Win32. При этом, учитывая, что 16-разрядные приложения уходят в прошлое, мы не будем специально выделять эти средства, полагая, что читатель и не станет пытаться работать с примерами из этой книги в 16-разрядной среде.

Так же как и приложения MS-DOS, программы для Windows можно писать на разных языках. Среди наиболее употребительных языков можно отметить C++, Delphi, C#, Java; в некоторых случаях (например, при разработке драйверов) используется язык ассемблера. Естественно, ни внутренние возможности Windows, ни тем более концепции этой системы никак не зависят от используемого языка, однако практические руководства по программированию, к каковым относится и эта книга, должны опираться на какой-то определенный язык программирования и даже на определенный вариант компилятора, или, точнее говоря, варнант инструментальной системы разработки программного обеспечения. При разработке примеров этой книги использовались главным образом интегрированные среды разработки Borland C++ 5.02 и Visual C++ 6.0. Для начинающего программиста первый пакет заметно удобнее; второй предоставляет больше возможностей.

Относительно несложные программы обычно разрабатываются с использованием традиционной методики программирования, иногда называемой процедурным программированием. По мере возрастания объема и сложности создаваемых программных продуктов все более привлекательной оказывается более современная и чрезвычайно популярная идеология, получившая название объектно-ориентированного программирования (ООП). Сущность ООП заключается в том, что программа выглядит не как последовательность вызовов подпрограмм – процедур обработки тех или иных данных, а как система взаимодействия самостоятельно живущих объектов, внутренний состав и правила функционирования которых определены заранее в описаниях классов этих объектов. Применение ООП позволяет сделать программный комплекс существенно более компактным и логичным и многократно сокращает затраты на проектирование. Разработанная однажды библиотека объектов, если она достаточно универсальна, может затем использоваться в самых различных программных проектах.

В современные системы программирования входят готовые библиотеки объектов, разработанные для реализации в прикладных программах возможностей системы Windows. Эти библиотеки (под названием OWL от Object Windows Library в пакете Borland C++ и MFC от Microsoft Foundation Classes в пакете Microsoft Visual C++) позволяют сократить объем интерфейсной части программы иногда в 10 и более раз и, главное, повысить ее качество, поскольку объекты этих библиотек разрабатываются высококвалифицированными программистами с использованием наиболее оптимальных алгоритмов. Однако работа с этими библиотеками требует основательного изучения состава и возможностей описанных в них объектов, а это, в свою очередь, предполагает наличие у программиста опыта процедурного программирования в системе Windows. Действительно, объекты библиотек OWL или MFC сами состоят из вызовов "обычных" функций Windows и для того, чтобы разобраться в функционировании объектов, необходимо знать основные концепции Windows и назначение ее функций, предназначенных для использования в прикладных программах, что и являтся предметом настоящей книги.

Разработка Windows-приложений требует понимания особенностей функционирования различных вариантов системы Windows. Большая часть приводимых в книге примеров будет одинаково хорошо работать во всех версиях Windows – от Windows 95 (или даже Windows 3.1) до Windows NT, 2000 или XP. Любопытно отметить, что платформы Windows 9x и NT весьма сильно отличаясь внутренними алгоритмами и даже концепциями, используют одни и те же базовые интерфейсные функции. Приложение, отлаженное в одной системе, будет так же хорошо работать и в любой другой. Последнее, однако, в полной мере относится только к 16-разрядным приложениям. Некоторые из средств Win32 реализованы только в Windows NT. В системах Windows 95 или 98 эти средства или не будут работать вообще, или их функциональные возможности будут ограничены.

Очевидно, что для составления прикладных программ требуется знание используемого языка программирования и основных понятий операционной системы. Начинающими программистами менее осознается тот факт, что как разработка, так и отладка более или менее сложных программ на языке высокого уровня, например C++, невозможна без достаточно глубокого знакомства с архитектурой процессора и языком ассемблера. Действительно, использование, например, такого мощного средства Win32, как файлы, проецируемые в память, являющегося основой работы с большими массивами данных и организации передачи данных между параллельными процессами, требует понимания весьма непростых механизмов адресации физической памяти, заложенных в процессоре, а также особенностей использования этих механизмов различными системами Windows. При организации параллельно выполняемых потоков приходится сталкиваться с особенностями использования потоками своих стеков – областей памяти с особыми свойствами, поддержка которых осуществляется с помощью специально выделенных для этого аппаратных регистров процессора. Понимание особенностей взаимодействия прикладной программы с системой Windows или с внешними устройствами, могущих отразиться на эффективности выполнения программы, требует знакомства с механизмами защиты, заложенными в процессоре, – системой уровней привилегий, ограничениями, накладываемыми процессором на обращение к портам, и др.

С архитектурой процессора и языком ассемблера программист постоянно сталкивается и при отладке программ, написанных на языке C++ или любом другом языке высокого уровня. Хотя многие ошибки программы можно обнаружить и исправить на уровне используемого языка, контролируя, например, значения программных переменных или значения, возвращаемые вызываемыми функциями, однако в столь же многих случаях понять сущность ошибки (приводящей, например, к "затиранию" какого-либо поля данных) можно, только обратившись к интерактивному отладчику, включаемому в состав любой современной системы программирования. Отладчик выводит на экран фактическое содержимое загрузочного модуля отлаживаемой программы (машинные команды), сопровождая эти данные мнемоническими обозначениями команд на языке ассемблера, а также указаниями, к каким операторам исходного языка программирования относятся те или иные группы машинных команд. Выполняя программу под управлением такого отладчика по шагам, можно "залезть" внутрь предложений исходного языка и более детально проконтролировать содержимое регистров процессора и ячеек памяти на каждом этапе ее выполнения.

Современные процессоры чрезвычайно сложны, и даже беглое описание их архитектуры может занять не один том. Однако со многими архитектурными особенностями процессора, важными, может быть, для его производительности, прикладной программист никогда не сталкивается. Другие средства, заложенные в процессоре, не использует сама система Windows (хотя, возможно, будет использовать в дальнейшем). Учитывая

важность всех этих вопросов, настоящая книга начинается главой, кратко освещающей те особенности архитектуры процессоров Intel, которые, на наш взгляд, имеют особое значение для прикладного программиста.

Книга является переработанным изложением лекционных курсов, читаемых автором в течение ряда лет студентам различных специальностей МИФИ, а также специалистам промышленности и "продвинутым" школьникам. Это обусловило ее характер – она предназначена прежде всего для начального (хотя и достаточно глубокого) ознакомления с предметом. Более детальные сведения по затронутым вопросам читатель может найти в специальной литературе, интерактивных справочниках, входящих в системы программирования, а также на многочисленных программистских сайтах Интернета.

Глава 1
Архитектура процессоров Intel

Как уже отмечалось, в процессе написания и особенно отладки программ мы постоянно сталкиваемся с элементами архитектуры процессора и системными особенностями организации программ. Профессиональный программист должен быть хорошо знаком с такими архитектурными вопросами, как назначение и использование регистров процессора, способы адресации памяти, сегментная структура программ, особенности реального и защищенного режимов и многими другими. При этом в зависимости от задач, решаемых разрабатываемым программным обеспечением, особую важность приобретают те или иные аспекты организации компьютера. Так, при разработке программ управления автоматизированным производством требуется глубокое понимание систем прерываний и ввода-вывода, особенностей управления аппаратными средствами, возможностей временной синхронизации процессов. Программы обслуживания измерительных систем могут предъявлять высокие требования к скорости выполнения, и тогда на первый план выступают такие вопросы, как время выполнения команд и средства повышения эффективности разрабатываемых программных фрагментов. Компьютерное моделирование экономических или других процессов часто сопряжено с использованием очень больших массивов данных, реализация которых потребует знакомства с организацией физической и виртуальной памяти и алгоритмами трансляции адресов.

В настоящей главе будут рассмотрены наиболее важные из этих вопросов применительно к разработке 32-разрядных приложений Windows.

Память и процессор

Среди всех многочисленных устройств, входящих в состав компьютера, важнейшими, несомненно, являются центральный процессор и оперативная память. В памяти хранится выполняемая программа с принадлежащими ей данными; процессор выполняет вычисления и другие действия, описанные в программе.

Программы составляются на том или ином языке программирования: C++, Паскале, языке ассемблера и др. В любом случае исходный текст программы состоит из условных обозначений – операторов и других ключевых слов используемого языка. Часто эти обозначения представляют собой английские слова или их сокращения (add, call, for, switch и т. д.); для выполнения арифметических, логических и прочих операций используются привычные нам знаки сложения (+), умножения (*), присваивания (=) и др.

Для того чтобы сделать программу выполнимой, ее исходный текст следует обработать специальной программой транслятора (компилятора), который преобразует предложения исходного языка в машинные коды команд процессора. Результат работы транслятора обычно называют объектным модулем программы; файл с объектным модулем всегда имеет расширение .OBJ. Хотя объектный модуль состоит из машинных кодов, выполнить его еще нельзя. Для преобразования объектного модуля в выполнимый, или, как говорят, загрузочный, модуль с расширением .EXE или .COM, его следует обработать еще одной служебной программой – компоновщиком, или редактором, связей. Основное назначение компоновщика заключается в подключении к основной программе всех используемых в ней подпрограмм. В простых случаях подпрограмм может

и не быть, однако использование компоновщика является обязательным во всех случаях, так как помимо сборки программы в единое целое он еще придает программе необходимый для выполнения формат.

Машинные коды команд процессора представляют собой числа, в которых содержится информация о роде выполняемой операции (сложение, пересылка, проверка и т. д.), а также о местоположении используемых в данной команде операндов. В ряде случаев в код команды включаются сами операнды.

Обычно загрузочные модули оттранслированных и готовых к выполнению программ хранятся в виде файлов на жестких или гибких магнитных дисках. Для того чтобы программа могла выполняться, ее следует загрузить с диска в память и, как говорят, передать ей управление. Эти действия невозможно выполнить вручную; для запуска программы необходимо тем или иным способом (командой, вводимой с клавиатуры и включающей в себя имя программы, или щелчком клавиши мыши по значку, олицетворяющему собой программу) сообщить операционной системе компьютера, какую именно программу мы хотим выполнить. Операционная система отыщет файл этой программы на диске, загрузит его в память и сообщит процессору адрес первой команды программы. С этого момента процессор начинает последовательно выполнять команды загруженной программы.

Еще раз подчеркнем, что программа, загруженная в память, представляет собой набор чисел. Некоторые из этих чисел являются кодами машинных команд, другие – данными, над которыми выполняются требуемые действия. Команды и данные сами по себе, в отрыве от их местонахождения, абсолютно неразличимы. Так, шестнадцатеричное число 40 является кодом машинной команды увеличения содержимого регистра AX на единицу:

```
inc     AX
```

Однако в другом контексте то же число 40 может обозначать индекс в массиве или номер порта (конкретно номер 40 присвоен одному из регистров системного таймера). Разделение команд и данных осуществляется по тому месту в программе, где они встречаются. Так, процессор, начав выполнять фрагмент программы, ожидает встретить в этом фрагменте исключительно коды команд, но не, например, массив с данными. Массивы данных, как и вообще любые данные, располагаются в отдельном месте программы, и обращаться к этому участку надо только с целью извлечения оттуда данных, но не для выполнения содержащихся там кодов.

Минимальный объем информации, к которому имеется доступ в памяти, составляет 1 байт (8 двоичных разрядов, или битов). Некоторые данные (например, коды букв, цифр и других символов) требуют для своего хранения 1 байта; для других данных этого места не хватает, и под них в памяти выделяется 2, 4, 8 или еще большее число байтов. Обычно пары байтов называют словами, а четверки – двойными словами (рис. 1.1), хотя иногда термином "слово" обозначают любую порцию машинной информации.

Помимо ячеек оперативной памяти, для хранения данных используются еще запоминающие ячейки, расположенные в самом процессоре и называемые *регистрами*. Достоинство регистров заключается в их высоком быстродействии, гораздо большем, чем у оперативной памяти, а недостаток в том, что их очень мало – всего около десятка. Поэтому регистры используются лишь для кратковременного хранения данных. Например, при выполнении процессором команды умножения двух чисел, процессор всегда помещает произведение в один из своих регистров, откуда при необходимости его можно перенести в память для более длительного хранения. Почти все регистры процессора име-

ют длину 32 бит, или 4 байта. За каждым регистром закреплено определенное имя (например, EAX или ESP), по которому к нему можно обращаться в программе.

Любая информация хранится в компьютере в виде *двоичных* чисел. Двоичная система счисления, естественная для цифровых электронных устройств, неудобна для человека. Для удобства представления (в книгах или в устной речи) двоичного содержимого ячеек памяти или регистров процессора обычно используют шестнадцатеричную систему счисления.

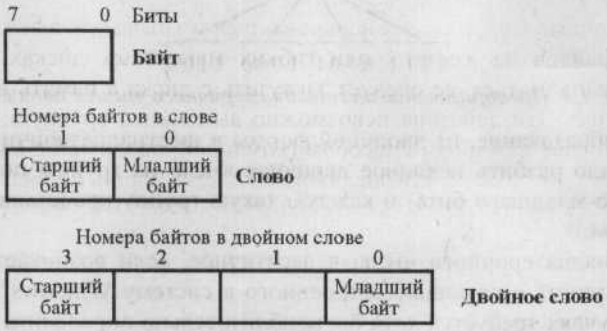


Рис. 1.1. Байт, слово и двойное слово

Каждый разряд шестнадцатеричного числа может принимать 16 значений, из которых первые 10 обозначаются обычными десятичными цифрами, а последние 6 – буквами латинского алфавита от A до F (рис. 1.2).

Представление чисел

двоичное	десятичное	шестнадцатеричное
0 0 0 0	0	0
0 0 0 1	1	1
0 0 1 0	2	2
0 0 1 1	3	3
0 1 0 0	4	4
0 1 0 1	5	5
0 1 1 0	6	6
0 1 1 1	7	7
1 0 0 0	8	8
1 0 0 1	9	9
1 0 1 0	10	A
1 0 1 1	11	B
1 1 0 0	12	C
1 1 0 1	13	D
1 1 1 0	14	E
1 1 1 1	15	F
8 4 2 1	← Значения двоичных разрядов (битов)	
2^3 2^2 2^1 2^0		

Рис. 1.2. Представление первых 16 чисел в разных системах

Поскольку одна шестнадцатеричная цифра требует для записи ее в память компьютера четырех двоичных разрядов, то содержимое байта описывается двумя шестнадцатеричными цифрами (от 00 до FF), содержимое слова – четырьмя (от 0000 до FFFF), а содержимое двойного слова – восемью (от 00000000 до FFFFFFFF). В программах шестнадцатеричные числа указываются по правилам, диктуемым транслятором с данного языка. Так, в программах на языке ассемблера шестнадцатеричное число должно закан-

чиваться буквой h (например, 1B7Fh), а в программах на языке C++ начинаться с символов 0x (например, 0x1B7F). Какие буквы при этом использовать – строчные или прописные, – значения не имеет.

Перевод шестнадцатеричного числа в двоичное осуществляется весьма просто. Для этого достаточно каждую цифру шестнадцатеричного числа представить в виде четверки двоичных цифр (рис. 1.3).

Рис. 1.3. Преобразование шестнадцатеричного числа в двоичное

Обратное преобразование, из двоичной формы в шестнадцатеричную, тоже не представляет труда. Надо разбить исходное двоичное число на группы по 4 бита (с правой стороны, от самого младшего бита) и каждую такую группу представить в виде шестнадцатеричной цифры.

Перевод шестнадцатеричного числа в десятичное, если возникает такая необходимость, лучше выполнять с помощью встроенного в систему Windows калькулятора. Однако во многих случаях требуется хотя бы приблизительно определить величину шестнадцатеричного числа. Для этого полезно помнить (или уметь быстро определить) значения последовательных разрядов шестнадцатеричного числа (рис. 1.4). Напомним, что в вычислительной технике величина 2^{10} = 1024 обозначается буквой К (от кило); величина 2^{20} = 1024 x 1024 = 1048576 обозначается буквой М (от мега); величина 2^{30} = 1024 x x 1024 x 1024 = 1073741824 обозначается буквой Г (от гига). Таким образом, 1 Кбайт несколько больше 1 000 байт, 1 Мбайт немного превышает 1 млн. байт, а 1 Гбайт составляет приблизительно 1,07 млрд. байт.

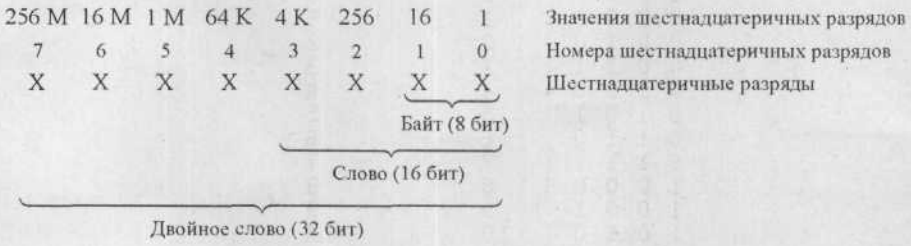


Рис. 1.4. Значения шестнадцатеричных разрядов

Имея перед глазами табличку, изображенную на рис. 1.4, нетрудно сообразить, что, например, число 0x402074 составляет приблизительно 4 М, а число 0xC39200E8 имеет порядок 3 Г.

Вернемся к вопросу об оперативной памяти и ее содержимом. Оперативная память компьютера представляет собой электронное устройство, состоящее из большого числа двоичных запоминающих элементов, а также схем управления ими. На системной плате компьютера всегда предусматривается определенное количество плоских разъемов для подключения модулей памяти. Таким образом, память компьютера можно наращивать, увеличивая количество установленных модулей или заменяя их модулями большей емкости. Обычно на индивидуальных персональных компьютерах устанавливают память объемом 256...512 Мбайт и более. Необходимо иметь в виду, что производительность компьютера сильно зависит от объема установленной оперативной памяти, так как при уве-

личении объема памяти операционной системе приходится реже обращаться к магнитному диску, который является одним из самых медленных устройств компьютера.

Как уже отмечалось, оперативная память компьютера состоит из байтов, нумеруемых начиная с нуля. Нужные байты отыскиваются в памяти по их номерам, выполняющим функции адресов, при этом многие данные (например, целые или дробные числа) занимают в памяти более одного байта.

При обсуждении содержимого многобайтового данного приходится ссылаться на составляющие его байты; эти байты условно нумеруются от нуля и располагаются (при их изображении на бумаге) в порядке *уменьшения* номера слева направо, так что слева оказываются байты с бóльшими номерами, а справа – с меньшими, как это показано, например, на рис. 1.1. Крайний слева байт называют старшим, крайний справа – младшим. Такой порядок расположения байтов связан с привычной для нас формой записи чисел: в многоразрядном числе слева указываются старшие разряды, а справа – младшие.

Однако в памяти компьютера (при использовании процессоров Intel) данные располагаются в более естественном порядке непрерывного *возрастания* номеров байтов и, таким образом, каждое слово или двойное слово в памяти начинается с его младшего байта и заканчивается старшим (рис. 1.5).

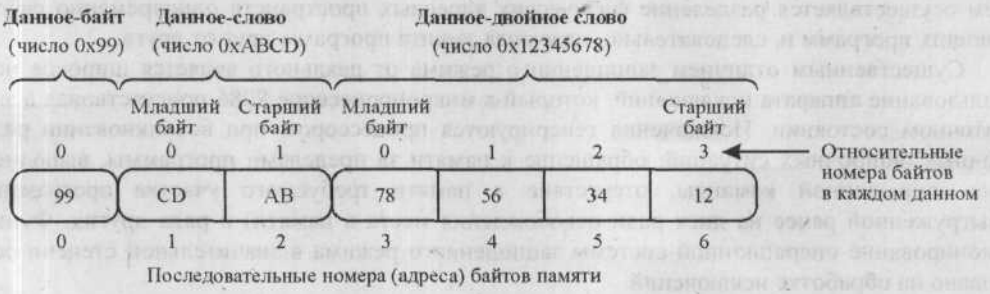


Рис. 1.5. Расположение в памяти байтов многобайтовых данных

Остановимся теперь на некоторых особенностях процессоров Intel, решающим образом повлиявших на ход развития всего современного программного обеспечения.

Первые персональные компьютеры корпорации IBM, получившие название IBM PC, использовали в качестве центрального вычислительного узла 16-разрядный микропроцессор Intel 8088 с 8-разрядной внешней шиной. В дальнейшем в персональных компьютерах стал использоваться другой вариант микропроцессора, 8086, который отличался от 8088 тем, что его внешняя шина имела ширину 16 разрядов. С тех пор его имя стало нарицательным и о программах, использующих только возможности процессоров 8088 или 8086, говорят, что они работают в режиме 86-го процессора.

В 1983 г. корпорацией Intel был предложен микропроцессор 80286, в котором был реализован принципиально новый режим работы, получивший название защищенного. Однако процессор 80286 мог работать и в режиме 86-го процессора, который стали называть реальным.

В дальнейшем на смену процессору 80286 пришли модели 80386, i486 и, наконец, различные варианты процессора Pentium. Все они могут работать и в реальном, и в защищенном режиме. Хотя каждая следующая модель была значительно совершеннее предыдущей (в частности, почти на 3 порядка возросла скорость работы процессора, начиная с модели 80386 процессор стал 32-разрядным, а в современных процессорах типа Pentium реализован даже 64-разрядный обмен данными с системной шиной), однако с

точки зрения программиста все эти процессоры весьма схожи. Основным их качеством является наличие трех режимов работы – реального, защищенного, а также режима виртуального 86-го процессора, или V86, который используется для эмуляции реального режима в рамках защищенного.

Реальный и защищенный режимы прежде всего принципиально различаются способом обращения к оперативной памяти компьютера. В реальном режиме программа может напрямую обратиться к любому участку физической оперативной памяти, однако процессору доступен лишь 1 Мбайт адресного пространства. В защищенном режиме используется совершенно другой механизм формирования физических адресов, который, с одной стороны, запрещает прямое обращение программы к физической памяти, а с другой – расширяет адресное пространство процессора до весьма внушительной величины – 4 Гбайт. Суть этого механизма заключается в том, что программа работает не в физическом адресном пространстве, а в пространстве виртуальных адресов, назначаемых командам и переменным программы в процессе ее компиляции. Отображение виртуальных адресов на физические осуществляется процессором по ходу программы с помощью специальных таблиц страничной трансляции, которые строятся операционной системой. При этом для каждой программы выделяется свой набор таблиц преобразования адресов, чем осуществляется разделение физических адресных пространств одновременно работающих программ и, следовательно, надежная защита программ друг от друга.

Существенным отличием защищенного режима от реального является широкое использование аппарата исключений, который в микропроцессоре 8086 присутствовал в зачаточном состоянии. Исключения генерируются процессором при возникновении различных ошибочных ситуаций: обращение к памяти за пределами программы, выполнение запрещенной команды, отсутствие в памяти требуемого участка программы (выгруженной ранее на диск ради освобождения места в памяти) и ряда других. Функционирование операционной системы защищенного режима в значительной степени основано на обработке исключений.

В защищенном режиме процессор реализует полностью отсутствующую в реальном режиме систему уровней привилегий. Каждому сегменту программы назначается определенный уровень привилегий, после чего процессор следит за тем, чтобы программы, имеющие низкий уровень привилегий, не вмешивались в работу программ более высокого уровня. Обычно системным сегментам назначается уровень максимальных привилегий, а прикладным – минимальных. Тем самым реализуется еще один слой защиты системных программ от их непреднамеренного повреждения прикладными.

В реальном режиме любая программа может безо всяких ограничений обращаться к любому устройству компьютера – видеосистеме, клавиатуре, таймеру, последовательным или параллельным портам и т. д. В защищенном режиме система формирует специальную таблицу – карту разрешения ввода-вывода, в которой указывается, какие порты компьютера открыты для прикладных программ. При обращении прикладной программы к запрещенному порту процессор формирует исключение общей защиты, которое передает управление операционной системе для принятия соответствующих мер. Таким образом процессор, работая в защищенном режиме, обеспечивает не только защиту программ друг от друга, но и защиту аппаратных средств компьютера.

Следует заметить, что перевод процессора из реального режима в защищенный и обратно осуществляется весьма просто – всего лишь установкой соответствующего бита в одном из управляющих регистров процессора. Однако команды чтения или записи этого регистра являются привилегированными и обычная прикладная программа их выполнить

не может – процессор тут же сформирует исключение, что приведет к передаче управления операционной системе и прекращению выполнения программы.

Реальный и защищенный режимы имеют прямое отношение к работе операционной системы, установленной на компьютере.

В настоящее время на персональных компьютерах типа IBM PC используются в основном два класса операционных систем (оба – разработки корпорации Microsoft): однозадачная текстовая система MS-DOS и многозадачная графическая система Windows. Операционная система MS-DOS является системой реального режима; другими словами, она использует только средства процессора 8086, даже если она установлена на компьютере с процессором Pentium. Система Windows – это система защищенного режима; она значительно более полно использует возможности современных процессоров, в частности многозадачность и расширенное адресное пространство. Разумеется, система Windows не могла бы работать с процессором 8086, так как в нем не был реализован защищенный режим.

Соответственно двум типам операционных систем все программное обеспечение персональных компьютеров подразделяется на два класса: программы, предназначенные для работы под управлением MS-DOS (приложения DOS), и программы, предназначенные для системы Windows (приложения Windows). Естественно, приложения DOS могут работать только в реальном режиме, а приложения Windows – только в защищенном.

Таким образом, выражения "программирование в системе MS-DOS", "программирование в реальном режиме" и "программирование 86-го процессора" фактически являются синонимами. При этом следует подчеркнуть, что, хотя процессор 8086 как микросхема уже давно не используется, его архитектура и система команд целиком вошли в современные процессоры. Когда на процессоре Pentium выполняется приложение DOS, оно использует лишь ту часть средств Pentium, которая соответствует архитектуре реального режима. Эту часть средств мы будем условно называть процессором МП 86. При рассмотрении архитектуры современных процессоров естественно затронуть и средства МП 86.

Системная шина и передача данных

Одной из важнейших концепций, лежащих в основе современных компьютеров, является идея системной шины (магистрали), которая связывает процессор со всеми остальными устройствами компьютера (рис. 1.6).

Системная шина представляет собой, в сущности, набор линий – проводов, к которым единообразно подключаются схемы управления всех устройств компьютера: клавиатуры, видеосистемы, последовательного и параллельного интерфейсов и т. д. В более широком плане в понятие системной шины включают электрические и логические характеристики сигналов, действующих на линиях шины, их назначение, а также правила взаимодействия этих сигналов при выполнении тех или иных операций на шине – то, что обычно называют протоколом шины. Сигналы, распространяющиеся по шине, доступны всем подключенным к ней устройствам, и в задачу каждого устройства входит выбор предназначенных ему сигналов и обеспечение реакции на них, соответствующей протоколу обмена.

В состав шины входит большое количество различных линий и их групп; наиболее важными из них являются группы (подшины) адресов и данных. Часть линий отводится для передачи управляющих и синхронизирующих сигналов, часть – для подачи питания и заземления. Часто группу линий адресов называют шиной адресов, а группу линий

данных – шиной данных, хотя обе они в действительности являются не самостоятельными шинами, а компонентами системной шины.

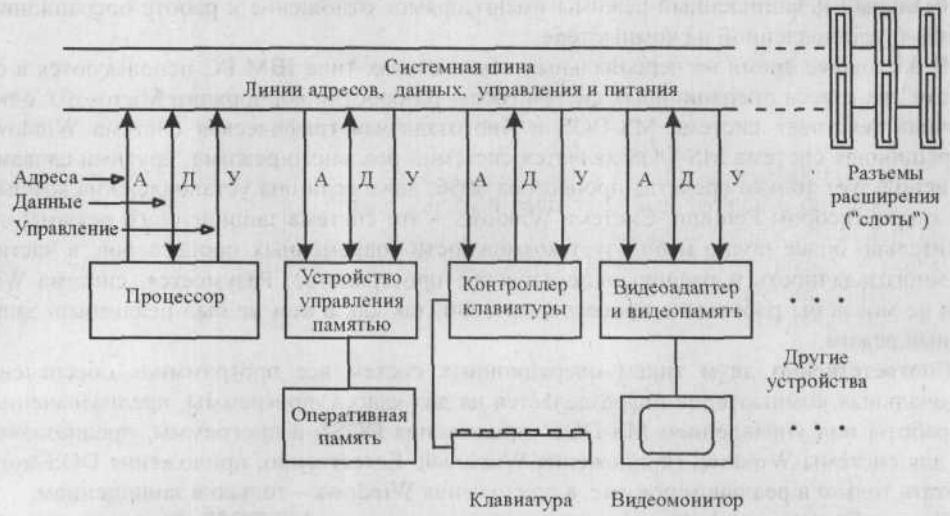


Рис. 1.6. Подключение устройств компьютера к системной шине

Цифровые электронные схемы работают исключительно с двоичными сигналами, т. е. с сигналами, которые могут принимать лишь два значения – "единица" и "нуль". Практически это означает, что во всех цепях компьютера, в том числе и на линиях магистрали, могут устанавливаться только два уровня напряжения, например 0 и 5 В (высокий уровень сигнала может быть и другим, например 3,3 В и менее). Один из этих уровней принимается за сигнал "нуль", другой – за сигнал "единица".

Рассмотрим случай, когда шины адресов и данных содержат всего по 8 линий. Тогда говорят, что их ширина составляет 8 двоичных разрядов (8 бит). Если мы хотим передать по шине адрес, равный 12, а в качестве данного – число 129, то сигналы на линиях адресов и данных следует установить так, как это показано на рис. 1.7.

Подшина адресов

Обозначение линии адреса	Номер бита	Значение бита	Значение сигнала для конкретного примера
A0	0	$2^0=1$	0
A1	1	$2^1=2$	0
A2	2	$2^2=4$	1
A3	3	$2^3=8$	1
A4	4	$2^4=16$	0
A5	5	$2^5=32$	0
A6	6	$2^6=64$	0
A7	7	$2^7=128$	0

Адрес = 4+8=12

Подшина данных

Обозначение линии данных	Номер бита	Значение бита	Значение сигнала для конкретного примера
D0	0	$2^0=1$	1
D1	1	$2^1=2$	0
D2	2	$2^2=4$	0
D3	3	$2^3=8$	0
D4	4	$2^4=16$	0
D5	5	$2^5=32$	0
D6	6	$2^6=64$	0
D7	7	$2^7=128$	1

Данное = 1+128=129

Рис. 1.7. Пример комбинации сигналов на линиях адресов и данных

Адреса служат для выделения требуемого устройства среди всей их совокупности. Каждому устройству компьютера, подключенному к магистрали, назначается некоторый диапазон неперекрывающихся адресов, на которые это устройство отзывается. Процессор, выполняя операцию записи данного в устройство, устанавливает на шине адресов один из его адресов, а на шине данных – передаваемое данное. Схема управления устройством (эта схема часто носит название контроллера или адаптера) расшифровывает поступивший адрес и, если этот адрес принадлежит устройству, принимает с линий данных поступившее данное. В случае выполнения обратной операции чтения из устройства процессор выставляет на линиях адресов требуемый адрес и ожидает поступления данных. Устройство, расшифровав поступивший адрес, выставляет имеющееся в нем данное на линии данных, а процессор, подождав некоторое время, снимает данное с шины.

Следует подчеркнуть, что если не говорить о специальном способе пересылки данных – прямом доступе к памяти, используемом главным образом для обмена данными с магнитными дисками, то инициатором пересылки данных всегда является процессор. Устройство может принимать и посылать данные только при обращении к нему процессора; инициировать процедуру пересылки данных устройство не может. Соответственно устройства могут общаться только с процессором, но не друг с другом.

В каком случае процессор выполняет процедуру пересылки данных? Инициатором этой процедуры всегда является выполняемая в настоящий момент программа. Следует заметить, что процессор компьютера, несмотря на свою поистине фантастическую сложность (современные процессоры содержат десятки миллионов транзисторов), фактически умеет делать только одно – выполнять команды, составляющие ту или иную программу. При этом некоторые команды целиком выполняются внутри процессора и не требуют обращения к магистрали. К таким командам относятся операции с регистрами процессора (регистры будут подробнее рассмотрены в следующем разделе):

```
mov     ECX,EAX     ;Пересылка содержимого регистра EAX в регистр ECX
inc     EDX         ;Увеличение содержимого регистра EDX на 1
add     AX,BX       ;Сложение содержимого регистров AX и BX
```

Если, однако, в качестве одного из операндов команды выступает ячейка оперативной памяти или регистр какого-либо устройства компьютера, то при выполнении такой команды происходит передача данного по магистрали либо в процессор (операция чтения), либо из процессора в память или устройство (операция записи):

```
mov     ECX,mem1    ;Пересылка данного из ячейки mem1 в регистр ECX
dec     mem2        ;Уменьшение содержимого ячейки mem2 на 1
in      AL,60h      ;Чтение данного в регистр AL из порта 60h
out     21h,AL      ;Запись данного из регистра AL в порт 21h
```

Для того чтобы процессор мог выполнить команду, ее код должен попасть в процессор. Однако команды выполняемой программы (как и ее данные) хранятся в оперативной памяти. Следовательно, перед выполнением команды она должна быть передана по магистрали из памяти в процессор. Эту операцию инициирует процессор, как только он освободится от выполнения предшествующей команды.

Таким образом, процесс выполнения любой программы заключается в бесконечной (до завершения программы) последовательности следующих операций:

- чтение из памяти в процессор кода очередной команды вместе с ее операндами;
- расшифровка кода команды;
- выполнение команды, возможно с пересылкой результата в память;
- подготовка к чтению следующей команды из памяти.

Схема связи процессора и устройств в том виде, как она изображена на рис. 1.6, использовалась в ранних компьютерах IBM PC и IBM PC/XT с процессорами Intel 8088 и 8086. Системная магистраль для этих компьютеров получила название ISA (от Industrial

Standard Architecture, стандартная промышленная архитектура). Эта магистраль использовалась в двух вариантах: с 8-разрядной шиной данных (для процессоров 8088) и с 16-разрядной шиной данных (для процессоров 8086). Ширина шины адресов и в том и в другом случае составляла 20 разрядов. Это давало возможность адресовать память общим объемом до 2^{20} байт = 1 Мбайт. Что же касается разрядности данных, то в обоих процессорах внутренние регистры имели размер 16 разрядов, что позволяло обрабатывать числа в диапазоне до 2^{16} = 64 К. Однако в компьютерах с 8-разрядной шиной ISA данные приходилось передавать по магистрали (в память или из памяти) за два цикла, сначала младший байт данных, затем старший. В компьютерах с 16-разрядной шиной ISA данные можно передавать по шине целиком, за один цикл.

Дальнейшее развитие микропроцессорной техники привело к появлению процессора Intel 80286 – первого процессора защищенного режима, который, оставаясь 16-разрядным, мог в защищенном режиме адресовать память объемом до 16 Мбайт. Это потребовало расширения адресной шины, и в компьютерах IBM PC/AT, построенных на базе этого микропроцессора, стал использоваться модифицированный вариант системной шины ISA с 24 адресными линиями (2^{24} = 16 Мбайт).

Для того чтобы в состав компьютера можно было включать дополнительное оборудование (например, звуковую плату, модем для телефонной связи или плату для подключения компьютера к локальной сети), на системной плате компьютера всегда монтируют несколько разъемов расширения, к контактам которых подводят линии системной шины (см. рис. 1.6). В просторечии эти разъемы называют слотами. В компьютерах, о которых шла речь выше, к разъемам расширения фактически подходила системная шина, в результате чего состав сигналов и скоростные характеристики самой шины и разъемов расширения совпадали.

С выпуском 32-разрядного процессора Intel 80386, позволяющего работать с физической памятью объемом до 2^{32} = 4 Гбайт, началась новая эпоха в развитии компьютеров типа IBM PC. Во-первых, 32-разрядная архитектура процессора потребовала расширения системной магистрали: ширина шин адресов и данных увеличилась до 32 линий. Во-вторых, для эффективного использования возможностей 32-разрядной архитектуры процессора потребовалась разработка новых операционных систем и нового прикладного программного обеспечения. В качестве операционной системы в персональных компьютерах стала преимущественно применяться система Windows; вновь выпускаемые прикладные программы общего назначения (графические и текстовые редакторы, программы управления базами данных и пр.) разрабатывались в 32-разрядном варианте.

Разработка компьютеров на базе процессоров 80386 с 32-разрядной системной шиной вступила в противоречие с использованием разъемов расширения, предназначенных для 16-разрядной шины ISA. В течение длительного времени компьютеры комплектовались (и зачастую комплектуются до сих пор) именно этими разъемами, для которых было выпущено огромное количество разнообразных плат расширения – звуковых, графических, модемных и пр. Однако для реализации преимуществ 32-разрядной архитектуры процессора передача данных между процессором и памятью должна осуществляться по 32-разрядной шине. Это противоречие было разрешено введением вместо общей для всего компьютера системной магистрали двух шин – внутренней, связывающей процессор с памятью, и внешней, служащей для подключения разъемов расширения (рис. 1.8).

Вместе с появлением 32-разрядных плат расширения (в первую очередь графических и сетевых) встал вопрос о замене 16-разрядных разъемов ISA каким-либо другим стандартом, допускающим 32-разрядный обмен данными. Дело усугубилось еще и тем, что появившиеся вслед за i486 процессоры Pentium имеют 64-разрядную шину данных (при прежней 32-разрядной внутренней архитектуре процессора). Кроме того, существенное повышение тактовой частоты процессоров (от 5 МГц до 1...2 ГГц и более) вступило в противоречие со ско-

ростными возможностями шины ISA, работающей на частоте 8 МГц. Решение этой проблемы было найдено в реализации так называемых локальных шин. Наиболее распространенной в настоящее время является шина PCI (от Peripheral Component Interconnect, межсоединение периферийных компонентов).

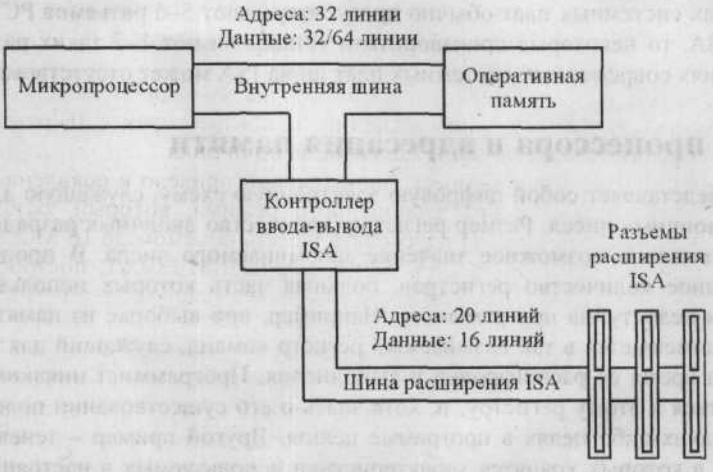


Рис. 1.8. Системные шины 32-разрядного компьютера с разъемами расширения ISA

Шина PCI подключается к процессору через специальное устройство – контроллер ввода-вывода, или PCI-мост, обеспечивающий разделение управляющих сигналов шины процессора и PCI-шины (рис. 1.9). При этом, в отличие от шины ISA, шина PCI содержит широкие подшины адресов и данных (по 32 разряда), что позволяет подключаемым устройствам полностью использовать адресное пространство процессора. Кроме того, шина PCI работает на значительно более высокой частоте – 33 МГц, что существенно увеличивает ее пропускную способность. Дальнейшие усовершенствования привели к появлению модификации PCI 2.0 с шириной шины 64 разряда, а вариант PCI 2.1 обеспечивает к тому же частоту работы шины 66 МГц.

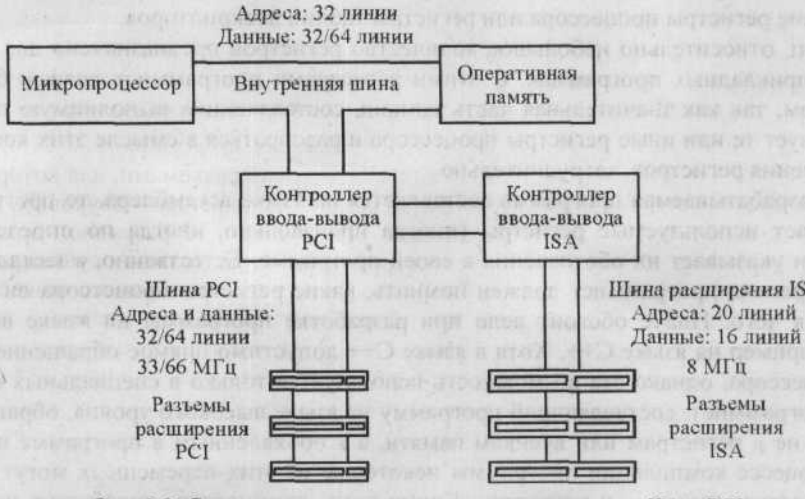


Рис. 1.9. Системные шины компьютера с разъемами ISA и PCI

Наличие на системной плате компьютера шины PCI не исключает возможности одновременного использования шины расширения ISA с ее разъемами, как это показано на рис. 1.9. Конструктивно разъемы PCI и ISA различаются (и по числу контактов, и по их конфигурации), что предотвращает ошибки при установке в них плат расширения. В новых разработках системных плат обычно предусматривают 5–6 разъемов PCI; что касается разъемов ISA, то некоторые производители устанавливают 1–2 таких разъема, в других конструкциях современных системных плат шина ISA может отсутствовать вовсе.

Регистры процессора и адресация памяти

Регистр представляет собой цифровую электронную схему, служащую для временного хранения двоичных чисел. Размер регистра (количество двоичных разрядов в нем) определяет максимально возможное значение запоминаемого числа. В процессорах Intel имеется огромное количество регистров, большая часть которых используется самим процессором и недоступна программисту. Например, при выборке из памяти очередной команды она помещается в так называемый регистр команд, служащий для хранения кода команды на время ее расшифровки и выполнения. Программист никаким образом не может обратиться к этому регистру, и, хотя знать о его существовании полезно, использовать его в каких-либо целях в программе нельзя. Другой пример – теневые регистры дескрипторов, в которых хранятся характеристики используемых в настоящий момент в программе сегментов памяти, в частности их адреса и размеры. Когда в программе осуществляется настройка на тот или иной программный сегмент, процессор сам заполняет соответствующий теневой регистр характеристиками этого сегмента (которые он берет из таблицы дескрипторов) и при дальнейших обращениях к сегменту получает из теневого регистра требуемую информацию. Программист не может ни записать ничего в теневой регистр, ни прочитать из него.

Большая группа регистров процессора в принципе программно доступна, однако обращение к ним осуществляется исключительно из самой операционной системы; другими словами, эти регистры используют системные программисты на этапе разработки программных модулей операционной системы. К таким регистрам относятся, например, управляющие регистры процессора или регистры таблиц дескрипторов.

Наконец, относительно небольшое количество регистров предназначено для использования в прикладных программах. С этими регистрами программист должен быть хорошо знаком, так как значительная часть команд, составляющих выполнимую программу, использует те или иные регистры процессора и разобраться в смысле этих команд, не зная назначения регистров, затруднительно.

Если разрабатываемая программа составляется на языке ассемблера, то программист сам выбирает используемые регистры (иногда произвольно, иногда по определенным правилам) и указывает их обозначения в своей программе. Естественно, в каждом фрагменте программы программист должен помнить, какие регистры процессора он использовал и для чего. Иначе обстоит дело при разработке программы на языке высокого уровня, например на языке C++. Хотя в языке C++ допустимо прямое обращение к регистрам процессора, однако эта возможность используется только в специальных случаях; обычно программист, составляющий программу на языке высокого уровня, обращается в программе не к регистрам или ячейкам памяти, а к объявленным в программе переменным. В процессе компиляции программы некоторые из этих переменных могут разместиться в памяти, другие – в регистрах. Кроме того, компилятор, преобразуя исходный текст программы в машинные коды, использует ряд регистров по своему усмотрению.

В результате результирующий текст программы так же насыщен обращениями к регистрам процессора, как если бы мы написали эту программу на языке ассемблера (см. рис. 2.17, где можно заметить использование регистров esi, ebx, esp, ebp, eax и edx).

Архитектура реального режима

Состав регистров процессора, служащих для использования в прикладных программах, фактически был определен еще в процессоре 8086 и перешел в дальнейшем в современные 32-разрядные процессоры, образуя в них часть гипотетического процессора МП 86. Как уже отмечалось выше, именно эти регистры используются программами реального режима (а также, между прочим, и 16-разрядными приложениями Windows). В соответствии с 16-разрядной архитектурой процессора 8086 все эти регистры имеют длину 16 разрядов (рис. 1.10).

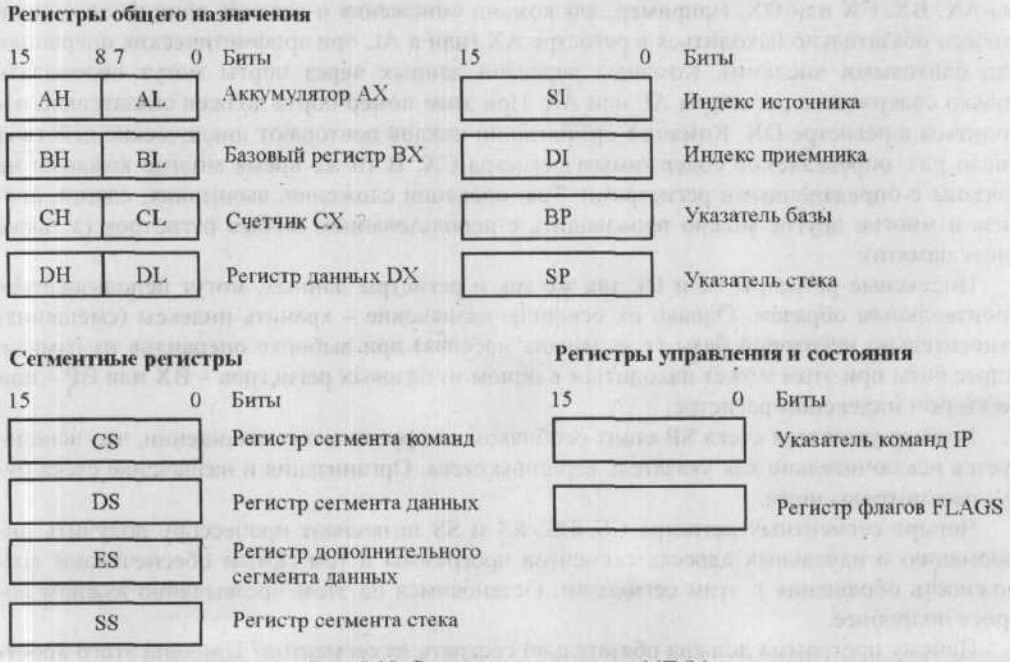


Рис. 1.10. Регистры процессора МП 86

Приведенные на рисунке названия регистров имеют большое значение, так как это не просто условные имена, а мнемонические обозначения. Мнемоническими принято называть такие имена, которые распознаются транслятором с языка ассемблера и могут использоваться в программах, написанных на этом языке. Язык C++ непосредственно мнемонических обозначений не понимает; для обращения к регистрам процессора напрямую в ряде случаев используются обозначения вида _AX, _BX и т. д.

Регистры указателя команд и флагов не имеют мнемонических обозначений и, следовательно, не являются программно адресуемыми. Приведенные на рисунке условные обозначения указателя команд IP (от Instruction Pointer) и регистра флагов FLAGS широко используются в литературе, однако являются просто аббревиатурами, которыми нельзя пользоваться в программах.

Рассмотрим кратко состав и назначение регистров МП 86.

Регистры AX, BX, CX и DX иногда называют регистрами данных. Они могут использоваться для временного хранения любой информации, хотя, как мы увидим позже, в ряде случаев их применение обусловлено определенными жесткими правилами. Все эти 4 регистра имеют длину по 16 разрядов, что позволяет записывать в них целые числа от 0 до $2^{16} - 1 = 64\ К - 1 = 65535$.

Во многих случаях диапазон чисел, используемых каким-то фрагментом программы, оказывается не столь велик. Например, коды отображаемых на экране символов занимают всего по 1 байту. Многие устройства компьютера (клавиатура, мышь, модем) также передают информацию отдельными байтами. Для упрощения работы с данными-байтами в регистрах AX, BX, CX и DX младшие и старшие байты имеют отдельные мнемонические обозначения: AL, BL, CL и DL для младших байтов и AH, BH, CH и DH для старших.

Многие команды процессора однозначно используют при своем выполнении регистры AX, BX, CX или DX. Например, для команд умножения и деления один из операндов должен обязательно находиться в регистре AX (или в AL при арифметических операциях над байтовыми числами). Команды передачи данных через порты могут пересылать только содержимое регистров AL или AX. При этом номер порта должен обязательно находиться в регистре DX. Команды организации циклов повторяют циклические действия число раз, определяемое содержимым регистра CX. В то же время многие команды не связаны с определенными регистрами. Так, операции сложения, вычитания, сдвига, анализа и многие другие можно производить с использованием любых регистров (а также ячеек памяти).

Индексные регистры SI и DI, так же как и регистры данных, могут использоваться произвольным образом. Однако их основное назначение – хранить индексы (смещения) относительно некоторой базы (т. е. начала массива) при выборке операндов из памяти. Адрес базы при этом может находиться в одном из базовых регистров – BX или BP – или во втором индексном регистре.

Регистр указателя стека SP стоит особняком от других в том отношении, что используется исключительно как указатель вершины стека. Организация и назначение стека будут рассмотрены ниже.

Четыре сегментных регистра CS, DS, ES и SS позволяют процессору получить информацию о начальных адресах сегментов программы и тем самым обеспечивают возможность обращения к этим сегментам. Остановимся на этом чрезвычайно важном вопросе подробнее.

Почему программа должна обязательно состоять из сегментов? Причина этого кроется в недостаточной длине регистров процессора МП 86. Действительно, при 16-разрядных регистрах максимальное целое число (данное или адрес), с которым может работать процессор, составляет 2^{16}-1 = 65535 (64 К–1). Если задавать адрес в виде одного числа, то адресное пространство процессора составило бы всего лишь 64 Кбайт. Для расширения адресного пространства (до 1 Мбайт) и предусмотрена сегментная адресация памяти, реализуемая с помощью четырех сегментных регистров.

Суть сегментной адресации заключается в следующем. Физический адрес любой ячейки памяти вычисляется процессором по следующей формуле:

```
Физический адрес = сегментный адрес * 16 + смещение
```

И сегментный адрес и смещение имеют размер 16 разрядов. При умножении сегментного адреса на 16 получается 20-разрядное число – начальный (базовый) физический адрес сегмента в памяти.

Сегментный адрес хранится в одном из сегментных регистров. Каждый раз при загрузке в сегментный регистр сегментного адреса процессор автоматически умножает его на 0x10 =16 и полученный таким образом базовый адрес сегмента сохраняет в одном из своих внутренних регистров (конкретно – в теневом регистре, упоминавшемся ранее). При необходимости обратиться к той или иной ячейке памяти процессор прибавляет к этому базовому адресу смещение ячейки, в результате чего образуется физический адрес ячейки в памяти. Умножение 16-разрядного сегментного адреса на 16 увеличивает диапазон адресуемых ячеек до величины 64 Кбайт * 16 = 1 Мбайт (рис. 1.11).

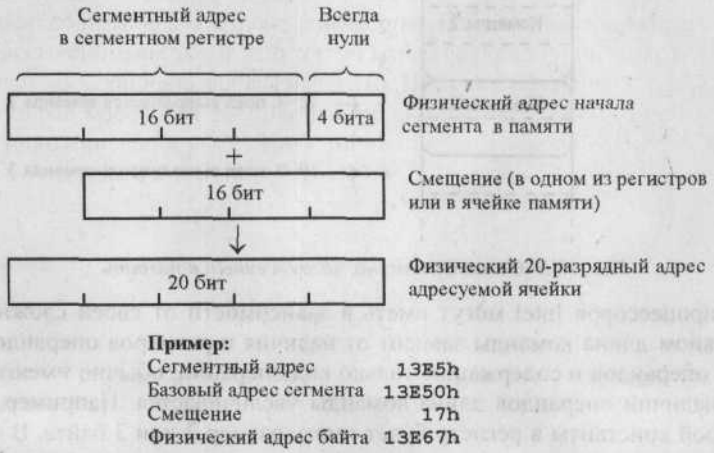


Рис. 1.11. Образование физического адреса из сегментного адреса и смещения

Итак, в МП 86 (или, что то же самое, в реальном режиме) обращение к любым участкам памяти осуществляется исключительно посредством сегментов – логических образований, накладываемых на требуемые участки физического адресного пространства. Из-за того что смещение имеет размер 16 разрядов, размер сегмента не может превышать 64 Кбайт. Еще раз подчеркнем, что сегментный адрес характеризует положение сегмента в физической памяти, а смещение – номер адресуемого байта от начала этого сегмента.

Обычно в программе имеются по крайней мере 3 сегмента – команд, данных и стека. Адресация сегмента команд осуществляется через сегментный регистр CS, адресация сегмента данных – через регистры DS или ES; регистр SS служит для обращения к стеку. Если программа сложна и не помещается в один сегмент, в ней может быть предусмотрено несколько сегментов команд. Точно так же при большом объеме данных в программе может быть несколько сегментов данных. Очевидно, что при переходе из сегмента в сегмент необходимо каким-то образом изменить содержимое соответствующего сегментного регистра.

Если программа пишется на языке ассемблера, правильную настройку сегментных регистров осуществляет программист. В программе, составляемой на языке высокого уровня, например C++, этим заниматься не надо, так как включение в программу соответствующих предложений обеспечит компилятор. Надо только понимать, что эти предложения в программе обязательно должны быть, так как, если сегментные регистры не будут содержать правильные сегментные адреса, процессор будет обращаться к каким-то неведомым участкам памяти.

Посмотрим теперь, как описанная выше процедура используется при выполнении процессором загруженной в память программы. На рис. 1.12 условно изображено начало

сегмента команд (на рисунке поместилось всего 3 команды; в реальном сегменте их, конечно, гораздо больше).

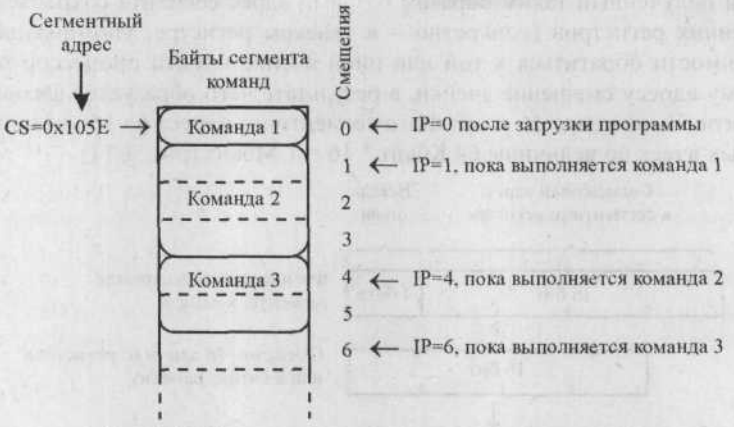


Рис. 1.12. Сегмент команд, загруженный в память

Команды процессоров Intel могут иметь в зависимости от своей сложности разную длину. В основном длина команды зависит от наличия и размеров операндов. Команды, не требующие операндов и содержащие только код операции, обычно имеют длину 1 или 2 байта. При наличии операндов длина команды увеличивается. Например, команда загрузки некоторой константы в регистр будет иметь размер 2 или 3 байта. В первом байте будет содержаться код операции и номер регистра-приемника, а еще 1 или 2 байта будет занимать загружаемая константа. Команда пересылки константы в память будет иметь размер 5 или 6 байт: 2 байта для кода операции, 2 – для адреса памяти и еще 1 или 2 – для пересылаемой константы. На рис. 1.12 изображен случай, когда первая команда занимает 1 байт, вторая – 3 байта, а третья – 2 байта.

Операционная система, загрузив программу в память, заносит в сегментный регистр CS сегментный адрес сегмента команд, а в указатель команд IP – смещение в этом сегменте первой выполнимой команды программы. Если текст программы начинается с ее главной процедуры, то начальное смещение в IP равно нулю. В других случаях (если, например, в программе сначала описаны подпрограммы) начальное смещение главной процедуры может оказаться иным.

Процессор начинает выполнение программы с того, что читает из памяти байт по адресу CS:IP (так обычно обозначается полный двухсловный адрес памяти; не следует забывать, что процессор вычисляет физический адрес байта, выполняя операцию CS * 16 + IP). Коды команд построены таким образом, что по коду первого байта команды процессор может определить, сколько байтов занимает вся команда. Прочитав первый и единственный байт первой команды, процессор увеличивает содержимое указателя команд IP на единицу и приступает к декодированию и выполнению кода команды. Все это время регистр IP указывает на первый байт очередной команды.

Завершив выполнение команды 1, процессор опять считывает из памяти байт по адресу CS:IP и тут же увеличивает IP на единицу. Декодировав первый байт команды и обнаружив, что это 3-байтовая команда, процессор считывает ее последующие байты, все время увеличивая на единицу содержимое IP. Когда все 3 байта команды будут прочитаны, процессор приступает к ее выполнению. Тем временем регистр IP указывает на пер-

вый байт следующей команды (байт с номером 4 в нашем случае). Далее весь этот процесс повторяется.

Таким образом, указатель команд IP "следит" за ходом выполнения программы, указывая в каждый момент смещение команды, следующей за исполняемой. При этом команды выполняются друг за другом; нарушение порядка выполнения команд возможно, только если процессор встретится с одной из команд переходов или вызовов подпрограмм.

В действительности все гораздо сложнее. В состав процессора включаются аппаратные средства, которые, усложняя его работу, способствуют существенному повышению скорости выполнения программ. К числу таких средств относятся конвейер команд и кеш-память.

Как уже отмечалось, выполнение каждой команды состоит из нескольких этапов: чтения из памяти кода команды, декодирования команды, чтения операндов и, наконец, собственно выполнения команды. Все эти операции можно выполнять (для последовательных команд) параллельно и одновременно с помощью отдельных узлов процессора. В этом случае, пока выполняется очередная команда, для следующей команды из памяти выбираются ее операнды, еще одна команда декодируется и, наконец, происходит чтение четвертой команды. При использовании такого конвейера скорость выполнения команд увеличивается, грубо говоря, в 4 раза.

Другое усовершенствование – кеш-память представляет собой быстродействующий блок памяти относительно небольшого объема, физически располагаемый в процессоре, а логически – между собственно процессором и основной оперативной памятью. Использование кеш-памяти основано на том уже упоминавшемся обстоятельстве, что команды программы в большинстве случаев выполняются друг за другом в том порядке, в каком они расположены в памяти. Это явление носит название локальности программы. Процессор, приступая к выполнению участка программы, считывает из основной памяти не одну команду, а целый блок байтов. Этот блок хранится в кеш-памяти, и по мере выполнения очередных команд процессор считывает их не из относительно медленной основной памяти, а из гораздо более быстрой кеш-памяти. Разумеется, во многих случаях требуемого процессору байта в кеш-памяти не оказывается (команда осуществляет переход на далекий участок программы или требуется далеко расположенный операнд). Такое событие называется кеш-промахом; оно приводит к обновлению кеш-памяти – считыванию в нее из основной памяти другого блока байтов, что уменьшает выигрыш, получаемый от использования кеш-памяти. Для повышения эффективности работы кеш-памяти предусматривается ряд мер, в частности, возможность считывания в нее нескольких смежных блоков основной памяти.

Несмотря на все эти технические сложности, можно считать, что в логическом плане процесс выполнения команд программы выглядит именно так, как изображено на рис. 1.12.

Рассмотрим теперь смысл смещения применительно к сегменту данных. Как правило, сегмент данных адресуется через сегментный регистр DS, для чего в этот регистр на начальном этапе выполнения программы (до первого обращения к ячейкам сегмента данных) должен быть помещен сегментный адрес сегмента данных. Но каким образом указывается смещение?

Пусть в самом начале сегмента данных описаны 3 байтовые переменные m0, m1 и m2 с начальными значениями 5, 10 и 15. В программе на языке ассемблера такой сегмент должен начинаться с предложений

```
m0      db 5
m1      db 10
m2      db 15
```

В программе на языке C++ то же самое будет выглядеть иначе:

```
char unsigned m0=5, m1=10, m2=15;
```

Если переменные m0, m1 и m2 находятся в самом начале сегмента данных, то их смещения относительно начала сегмента будут равны 0, 1 и 2 (рис. 1.13).

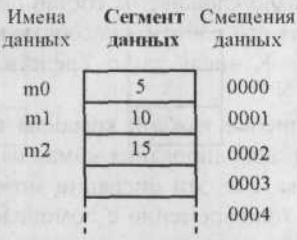


Рис. 1.13. Сегмент данных с байтовыми данными

Пусть теперь где-то в программе переменную m2 требуется увеличить на единицу. В программе на языке ассемблера это делается командой

```
inc     m2
```

а в программе на языке C++ предложением

```
m2++;
```

В любом случае в загрузочный модуль будет включена команда

```
FE 06 0002
```

где первые 2 байта, FE и 06, представляют собой код операции (инкремент байта памяти), а число 0002 является смещением адресуемой ячейки. Процессор, выполняя эту команду, прибавит содержащееся в коде команды смещение к базовому адресу сегмента данных (равному умноженному на 16 содержимому DS) и получит физический адрес адресуемого байта.

Таким образом, смещения адресуемых ячеек сегмента данных могут непосредственно входить в коды команд; в других случаях процессор использует смещение, загруженное заранее в один из регистров, например BX или SI.

Рассмотрим теперь третий вид сегмента, обязательно входящий в любую программу, а именно сегмент стека.

Стеком называют область программы для временного хранения произвольных данных. Отличительной особенностью стека является своеобразный порядок выборки содержащихся в нем данных: в любой момент времени в стеке доступен только верхний элемент, т. е. элемент, загруженный в стек последним. Выгрузка из стека верхнего элемента делает доступным следующий элемент.

Элементы стека располагаются в области памяти, отведенной под стек, начиная со дна стека (т. е. с его максимального адреса) по последовательно уменьшающимся адресам (рис. 1.14). Адрес верхнего, доступного элемента хранится в регистре – указателе стека SP. Как и любая другая область памяти программы, стек должен входить в какой-то сегмент или образовывать отдельный сегмент. В любом случае сегментный адрес этого сегмента помещается в сегментный регистр стека SS. Таким образом, пара регистров SS:SP описывает адрес доступной ячейки стека: в SS хранится сегментный адрес стека,

а в SP – смещение доступной (текущей) ячейки (рис. 1.14, *а*). Обратите внимание на то, что в исходном состоянии указатель стека SP указывает на ячейку, лежащую под дном стека и не входящую в него (на рис. 1.14 эта ячейка обозначена серым цветом).

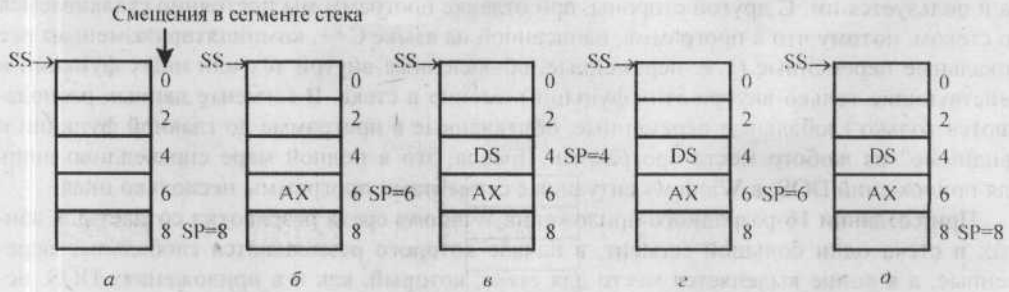


Рис. 1.14. Организация стека: а – исходное состояние; б – после загрузки первого элемента (в данном примере – содержимого регистра AX); в – после загрузки второго элемента (содержимого регистра DS); г – после выгрузки одного элемента; д – после выгрузки обоих элементов и возврата стека в исходное состояние

Загрузка в стек осуществляется специальной командой работы со стеком push (протолкнуть). Эта команда сначала уменьшает на 2 содержимое указателя стека, а затем помещает операнд по смещению, находящемуся в SP. Если, например, мы хотим временно сохранить в стеке содержимое регистра AX, следует выполнить команду

```
push    AX
```

Стек переходит в состояние, показанное на рис. 1.14, *б*. Видно, что указатель стека смещается на 2 байта вверх и по этому смещению записывается указанный в команде проталкивания операнд. Следующая команда загрузки в стек, например

```
push    DS
```

переведет стек в состояние, показанное на рис. 1.14, *в*. В стеке будут теперь храниться два элемента, причем доступным будет только верхний, на который указывает указатель стека SP. Если спустя какое-то время нам понадобится восстановить исходное содержимое сохраненных в стеке регистров, мы должны выполнить команды выгрузки из стека pop (вытолкнуть):

```
pop     DS
pop     AX
```

Состояние стека после выполнения первой команды pop показано на рис. 1.14, *г*, а после второй – на рис. 1.14, *д*. Для правильного восстановления содержимого регистров выгрузка из стека должна выполняться в порядке, строго противоположном загрузке, – сначала выгружается элемент, загруженный последним, затем предыдущий элемент и т. д.

Обратите внимание на то, что после выгрузки сохраненных в стеке данных они физически не стерлись, а остались в области стека на своих местах. Правда, при "стандартной" работе со стеком они оказываются недоступными. Действительно, поскольку указатель стека SP указывает под дно стека, стек считается пустым; очередная команда push поместит новое данное на место сохраненного ранее содержимого AX, затерев его. Однако пока стек физически не затерт, сохраненными и уже выбранными из него данными можно пользоваться, если помнить, в каком порядке они расположены в стеке. Реально, впрочем, так не делают.

Программист, составляющий программу на языке высокого уровня, непосредственно со стеком не работает. Стек автоматически создается средой программирования; компилятор, преобразуя программу в машинные коды, предполагает наличие в программе стека и пользуется им. С другой стороны, при отладке программ мы постоянно сталкиваемся со стеком, потому что в программе, написанной на языке C++, компилятор размещает все локальные переменные (т. е. переменные, объявленные внутри тех или иных функций и действующие только внутри этих функций) именно в стеке. В сегменте данных располагаются только глобальные переменные, объявленные в программе до главной функции и "видимые" из любого места программы. Правда, это в полной мере справедливо лишь для приложений DOS; в Windows ситуация с сегментами программы несколько иная.

При создании 16-разрядного приложения Windows среда разработки создает для данных и стека один большой сегмент, в начале которого размещаются глобальные переменные, а в конце выделяется место для стека, который, как и в приложениях DOS, используется, в частности, для хранения локальных переменных (рис. 1.15).

При загрузке программы в память регистры DS и SS принимают одно и то же значение. Обращение к локальным переменным осуществляется с использованием регистра SS и относительно больших смещений; при адресации глобальных переменных используется регистр DS и относительно малые смещения.

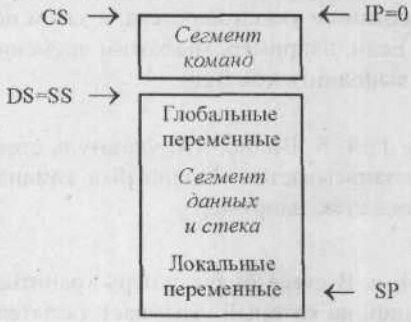


Рис. 1.15. Сегменты 16-разрядного приложения Windows

В 32-разрядных приложениях Windows сегменты программы выглядят несколько иначе, однако и там сегменты данных и стека объединяются и этот обобщенный сегмент используется для размещения как глобальных переменных (в начале сегмента), так и локальных (ближе к его концу). Этого вопроса мы еще коснемся при рассмотрении элементов архитектуры защищенного режима.

Последний важный регистр, назначение которого нам осталось выяснить, – это регистр флагов. Хотя считается, что в нем 16 разрядов, однако из них используются только 9 (рис. 1.16).

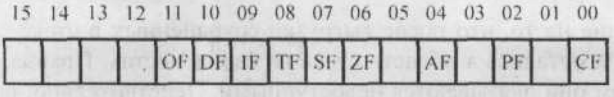


Рис. 1.16. Регистр флагов МП 86

Регистр флагов содержит информацию о текущем состоянии процессора и активно используется как процессором, так и прикладной программой. Однако при программировании на языке высокого уровня работа с регистром флагов протекает незаметно для программиста. Команды, анализирующие содержимое регистра флагов или, наоборот, изменяющие его содержимое, включаются в программу компилятором в процессе преоб-

разования предложений исходного текста в машинные коды. Программисту приходится напрямую обращаться к регистру флагов главным образом при отладке программы.

Регистр флагов включает 6 флагов состояния процессора и 3 бита управления состоянием процессора (они выделены на рисунке серым цветом), которые, впрочем, тоже обычно называются флагами. Суть использования флагов состояния заключается в том, что после выполнения каждой команды процессор устанавливает эти флаги в зависимости от результата выполненной только что команды. Например, если при выполнении команды сложения или вычитания получился нулевой результат, процессор устанавливает флаг нуля ZF; если же результат отличен от нуля, флаг нуля, наоборот, сбрасывается. Если результат команды отрицателен, устанавливается флаг знака SF; если же результат положителен, флаг SF сбрасывается и т. д. Следующая команда может проанализировать состояние требуемых флагов и в зависимости от результатов этого анализа выполнить те или иные действия. При этом в составе команд процессора имеются как "элементарные" команды анализа отдельных флагов, так и команды, анализирующие их различные комбинации. Последнее позволяет отобрать случаи, когда два данных связаны отношениями "больше", "больше или равно", "меньше", "меньше или равно" и др. Команды анализа флагов состояния включаются компилятором в программу при трансляции условных предложений, содержащих ключевое слово if.

Рассмотрим кратко назначение флагов состояния.

Флаг переноса CF (Carry Flag) индицирует перенос или заем при выполнении арифметических операций, а также служит индикатором ошибки при обращении к системным функциям.

Флаг паритета PF (Parity Flag) устанавливается в 1, если младшие 8 бит результата операции содержат четное число двоичных единиц.

Флаг вспомогательного переноса AF (Auxiliary Flag) используется в операциях над упакованными двоично-десятичными числами. Он индицирует перенос в старшую тетраду (четверку битов) или заем из старшей тетрады.

Флаг нуля ZF (Zero Flag) устанавливается в 1, если результат операции равен нулю.

Флаг знака SF (Sign Flag) показывает знак результата операции, устанавливаясь в 1, если результат является отрицательным числом.

Флаг переполнения OF (Overflow Flag) фиксирует переполнение, т. е. выход результата операции за пределы допустимого для данного процессора диапазона значений.

Управляющие флаги, в отличие от флагов состояния, устанавливаются или сбрасываются не процессором, а программой.

Управляющий флаг трассировки TF (Trace Flag) используется в отладчиках для осуществления пошагового выполнения программы. Если TF=1, то после выполнения каждой команды процессор реализует процедуру прерывания через вектор прерывания с номером 1.

Управляющий флаг разрешения прерываний IF (Interrupt Flag) разрешает (если равен единице) или запрещает (если равен нулю) процессору реагировать на прерывания от внешних устройств.

Управляющий флаг направления DF (Direction Flag) используется особой группой команд, предназначенных для обработки строк. Если DF=0, строка обрабатывается в прямом направлении, от меньших адресов к большим; если DF=1, обработка строки идет в обратном направлении.

Архитектурные особенности защищенного режима

Выше было рассмотрено влияние регистров процессора на структуру программы реального режима, а также роль регистров в организации процесса выполнения программы. Мы затронули следующие принципиальные вопросы:

- при обращении к памяти адрес любого ее байта задается двумя числами, сегментным адресом и смещением, причем сегментный адрес может находиться только в одном из сегментных регистров;
- физический адрес адресуемого байта памяти вычисляется процессором путем сложения умноженного на 16 сегментного адреса с 16-разрядным смещением;
- соответственно программа состоит из сегментов; обычно в программу входят сегменты команд, данных и стека;
- сегментный адрес сегмента команд всегда хранится в сегментном регистре CS, а смещение очередной команды находится в указателе команд IP;
- сегменты данных адресуются с помощью сегментных регистров DS и ES;
- сегмент стека служит для временного хранения данных и адресуется с помощью пары предназначенных для этого регистров SS и SP;
- регистр флагов служит для установки некоторых режимов работы процессора и для получения информации о результате выполнения очередной команды.

Рассмотрим теперь те же вопросы применительно к 32-разрядному защищенному режиму.

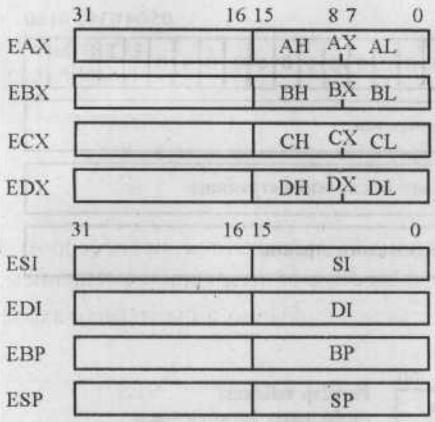
Как уже отмечалось, 16-разрядные регистры МП 86 входят в состав архитектуры современных процессоров Intel, обеспечивая возможность выполнения на 32-разрядных процессорах 16-разрядных программ. Однако реальные регистры 32-разрядных процессоров выглядят иначе (рис. 1.17).

Регистры общего назначения отличаются от аналогичных регистров МП 86 тем, что они являются 32-разрядными. Их мнемонические обозначения начинаются с буквы E (от Extended, расширенный). Иногда их так и называют – расширенные регистры процессора. Для сохранения совместимости с ранними моделями процессоров допускается обращение к младшим половинам всех регистров, которые имеют те же мнемонические обозначения, что и в МП 86 (AX, BX, CX, DX, SI, DI, BP и SP). Естественно, сохранена возможность работы с младшими (AL, BL, CL и DL) и старшими (AH, BH, CH и DH) половинами регистров МП 86. Однако старшие половины 32-разрядных регистров не имеют мнемонических обозначений и непосредственно недоступны. Если при программировании на языке ассемблера требуется, скажем, прочитать старшую половину регистра EAX (биты 31...16), придется сдвинуть все содержимое EAX на 16 бит вправо, в регистр AX, и прочитать затем содержимое регистра AX.

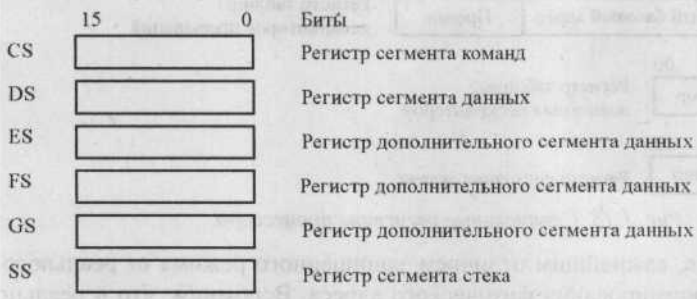
Все сегментные регистры, как и в МП 86, остались 16-разрядными. В их состав включены еще два сегментных регистра данных – FS и GS. Тем самым обеспечена возможность одновременной работы не с двумя сегментами данных, как в МП 86, а с четырьмя. Впрочем, в современных 32-разрядных приложениях нужда в таких действиях практически отпала.

Указатель команд, для которого в литературе используется условное обозначение EIP, тоже стал 32-разрядным, что имеет решающее значение для организации 32-разрядных программ. Младшая половина этого регистра соответствует регистру IP МП 86.

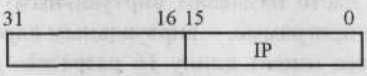
Регистры общего назначения



Сегментные регистры



Указатель команд EIP



Регистр флагов EFLAGS

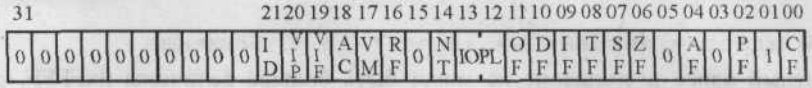
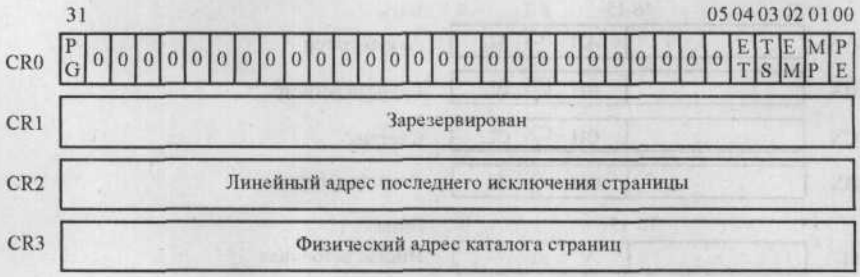


Рис. 1.17. Регистры 32-разрядных процессоров Intel

Регистр флагов с условным названием EFLAGS не только увеличил свою длину до 32 разрядов, в нем появилось много новых флагов. Все эти флаги носят весьма специальный характер и используются исключительно системными программами. Поэтому, хотя они и приведены для полноты на рис. 1.17, мы их рассматривать не будем.

Как уже упоминалось ранее, в состав процессора входит несколько специальных регистров, обслуживающих защищенный режим и используемых только в системных программах. Наиболее важными из них являются управляющие регистры процессора и регистры системных адресов (рис. 1.18). На некоторые из этих регистров нам придется ссылаться в дальнейшем при рассмотрении особенностей защищенного режима. Там же будет рассмотрено их назначение.

Управляющие регистры процессора



Регистры системных адресов

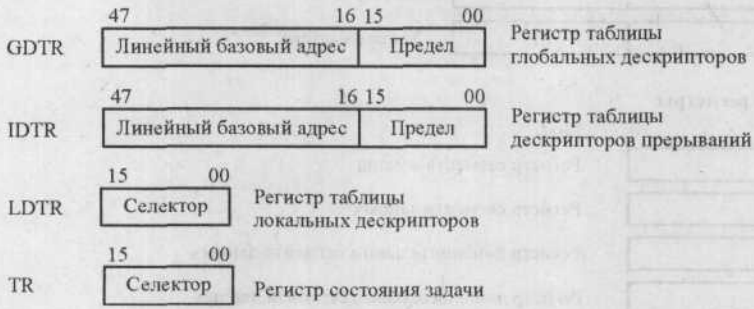


Рис. 1.18. Специальные регистры процессора

Как уже отмечалось, важнейшим отличием защищенного режима от реального является другой принцип формирования физического адреса. Вспомним, что в реальном режиме адрес любой ячейки памяти, задаваемый в программе, состоит из двух компонентов – сегментного адреса и смещения. Этот адрес часто называют виртуальным, а всю совокупность адресов, которые могут встречаться в программе, – виртуальным адресным пространством. Оба компонента виртуального адреса имеют длину 16 разрядов, и процессор, обращаясь к памяти, пользуется следующим правилом вычисления физического адреса:

```
Физический адрес = сегментный адрес * 16 + смещение
```

И сегментный адрес и смещение не могут быть больше величины 0xFFFF, откуда следуют два важнейших ограничения реального режима: объем адресного пространства составляет всего 1 Мбайт, а сегменты не могут иметь размер, превышающий 64 Кбайт.

В защищенном режиме программа по-прежнему состоит из сегментов, адресуемых с помощью 16-разрядных сегментных регистров, и местоположение адресуемого байта определяется, как и раньше, заданием смещения в сегменте, однако преобразование виртуального адреса в физический осуществляется не по приведенной выше формуле, а путем значительно более сложной цепочки преобразований (рис. 1.19).

Сначала процессор преобразует виртуальный адрес в линейный. В процессе этого преобразования процессор обращается к таблицам дескрипторов, которые заранее строятся операционной системой защищенного режима. На втором этапе по линейному адресу определяется физический адрес памяти. В этом преобразовании участвует другой набор системных таблиц – таблицы страничной трансляции, которые также составляются операционной системой. Оба набора системных таблиц могут динамически изменяться

операционной системой в процессе работы программ, обеспечивая максимально эффективное использование наличной оперативной памяти.

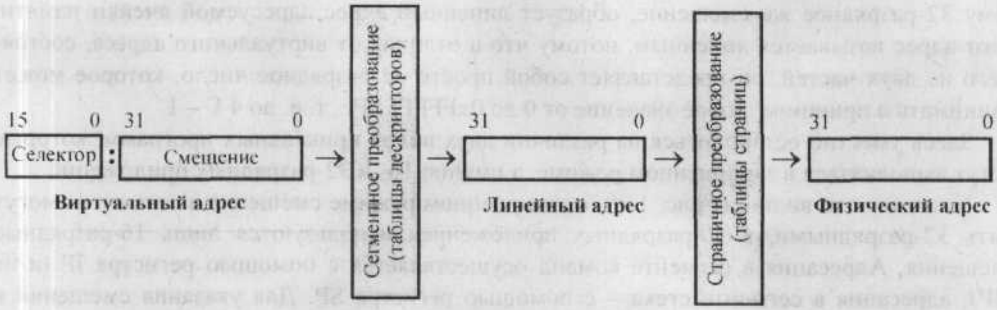


Рис. 1.19. Цепочка преобразования виртуального адреса в физический

Следует подчеркнуть, что, хотя таблицы преобразования строятся операционной системой, само преобразование виртуального адреса в физический осуществляет процессор на аппаратном уровне. Разумеется, таблицы находятся в физической памяти, и в процессе преобразования процессор вынужден дополнительно обращаться к памяти за недостающей информацией. Однако с учетом кеширования адресуемых участков физической памяти (см. предыдущий раздел) этот процесс осуществляется достаточно быстро.

Рассмотрим более детально цепочку преобразования виртуального адреса в физический, начав с первого этапа этого механизма – определения линейного адреса.

В сегментные регистры в защищенном режиме записываются не сегментные адреса, а так называемые селекторы, в состав которых входят номера (индексы) ячеек специальной таблицы, содержащей дескрипторы сегментов программы. Каждый дескриптор таблицы дескрипторов имеет размер 8 байт, и в нем хранятся все характеристики, необходимые процессору для обслуживания этого сегмента: базовый линейный адрес сегмента, его граница, которая представляет собой номер последнего байта сегмента, а также атрибуты сегмента, определяющие его свойства (рис. 1.20). Таким образом, селекторы в конечном счете характеризуют сегменты памяти.

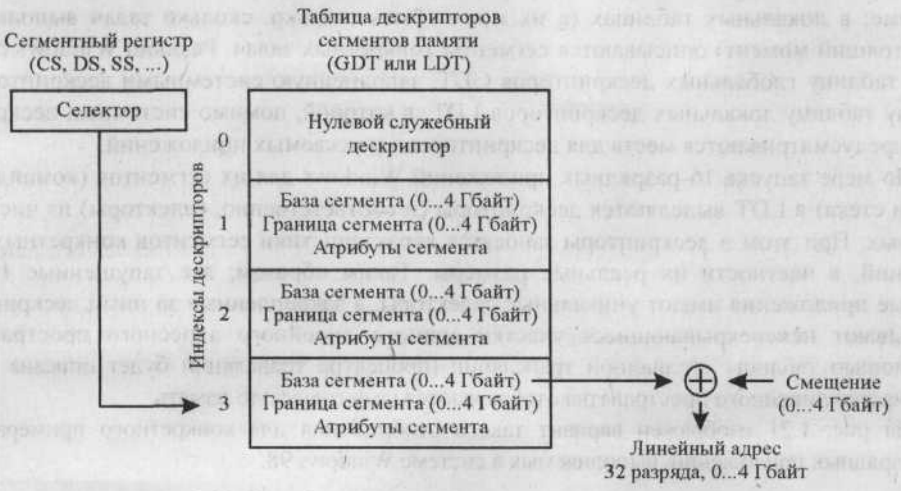


Рис. 1.20. Преобразование виртуального адреса в линейный

Процессор с помощью селектора определяет индекс дескриптора адресуемого сегмента, извлекает из него базовый линейный 32-разрядный адрес сегмента и, прибавив к нему 32-разрядное же смещение, образует линейный адрес адресуемой ячейки памяти. Этот адрес называется линейным, потому что в отличие от виртуального адреса, состоящего из двух частей, он представляет собой просто 32-разрядное число, которое может принимать в принципе любое значение от 0 до 0xFFFFFFFF, т. е. до 4 Г – 1.

Здесь уместно остановиться на различии двух видов прикладных программ, которые могут выполняться в защищенном режиме, а именно 16- и 32-разрядных приложений.

Хотя, как это видно из рис. 1.20, в защищенном режиме смещения в сегментах могут быть 32-разрядными, в 16-разрядных приложениях используются лишь 16-разрядные смещения. Адресация в сегменте команд осуществляется с помощью регистра IP (а не EIP); адресация в сегменте стека – с помощью регистра SP. Для указания смещений в сегментах данных используются младшие половины расширенных регистров процессора или 2-байтовые поля в составе команды. В результате размеры сегментов не могут превышать 64 Кбайт и для построения большой программы приходится использовать несколько сегментов команд или несколько сегментов данных.

Иначе функционируют 32-разрядные приложения защищенного режима. В них для задания смещений используются расширенные регистры процессора, в частности EIP для адресации в сегменте команд и ESP для работы со стеком. Ограничение 64 Кбайт уже не действует, сегменты могут быть любого размера, и необходимость в построении многосегментных программ отпадает. Как уже отмечалось, в настоящее время практически все новые приложения делаются 32-разрядными.

Вернемся к рис. 1.20. Архитектурой процессора предусмотрены таблицы дескрипторов двух типов – таблица глобальных дескрипторов GDT (от Global Descriptor Table) и таблица локальных дескрипторов LDT (от Local Descriptor Table). Различие между ними заключается в том, что сегменты, описанные в GDT, могут быть доступны всем выполняемым задачам (как прикладным, так и системным); сегменты, описанные в LDT, доступны только той задаче, к которой относится данная локальная таблица. В глобальной таблице описываются сегменты операционной системы, а также сегменты всех LDT в системе; в локальных таблицах (а их может быть столько, сколько задач выполняется в настоящий момент) описываются сегменты конкретных задач. Реально Windows строит одну таблицу глобальных дескрипторов GDT, заполненную системными дескрипторами, и одну таблицу локальных дескрипторов LDT, в которой, помимо системных дескрипторов, предусматриваются места для дескрипторов запускаемых приложений.

По мере запуска 16-разрядных приложений Windows для их сегментов (команд, данных и стека) в LDT выделяются дескрипторы (и соответственно, селекторы) из числа резервных. При этом в дескрипторы заносятся характеристики сегментов конкретных приложений, в частности их реальные размеры. Таким образом, все запущенные 16-разрядные приложения имеют уникальные селекторы, а закрепленные за ними дескрипторы описывают неперекрывающиеся участки единого линейного адресного пространства. С помощью таблицы страничной трансляции (процедура трансляции будет описана ниже) эти участки линейного пространства отображаются на физическую память.

На рис. 1.21 изображен вариант такого отображения для конкретного примера двух 16-разрядных приложений, выполняемых в системе Windows 98.

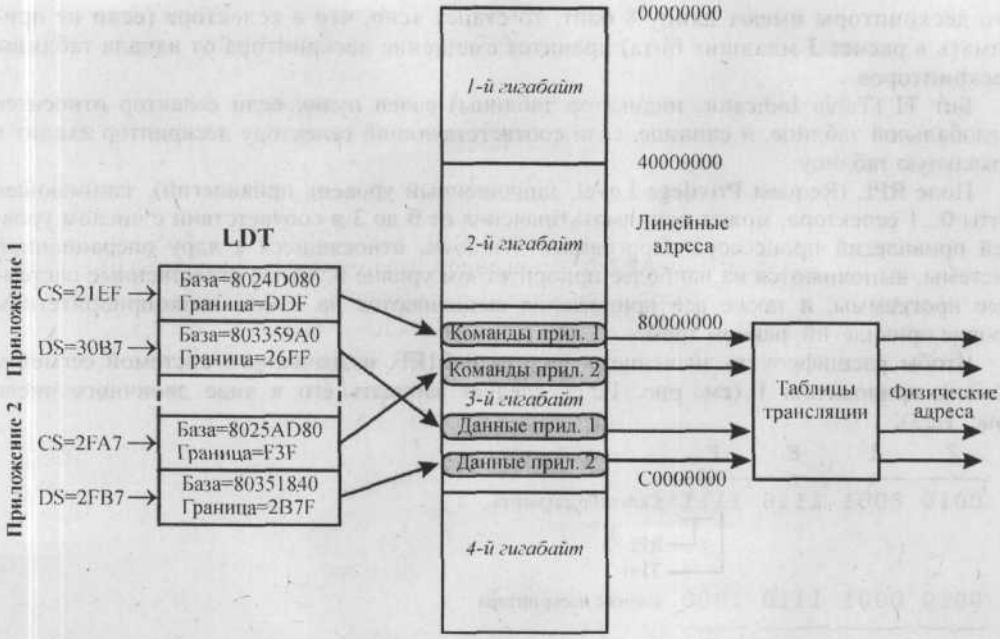


Рис. 1.21. Пример преобразования адресов для 16-разрядных приложений в Windows 98 (все числа шестнадцатеричные)

При запуске приложений система назначила их сегментам произвольные дескрипторы из числа свободных в LDT (с селекторами 0x21EF, 0x30B7 и т. д.). В дескрипторы были помещены характеристики сегментов: границы, отвечающие реальным размерам сегментов приложений, и базы, соответствующие свободным участкам линейного пространства. Обратите внимание на то, что все сегменты размещены в 3-м гигабайте (в области линейных адресов от 0x80000000 до 0xC0000000).

Линейные адреса не отвечают каким-либо физическим объектам. Это, можно сказать, фиктивные адреса, возникающие на полдороге от виртуальных адресов, с которыми работает приложение, к физическим адресам реальной оперативной памяти компьютера. Тем не менее они имеют большое значение и используются вполне определенным образом. Так, 16-разрядные приложения Windows 98 всегда работают в 3-м Гбайте линейного адресного пространства.

Получив линейный адрес адресуемого байта, процессор с помощью таблиц трансляции преобразует его в 32-разрядный физический адрес того байта памяти, где находится конкретная команда или конкретное данное. Этот адрес зависит от объема памяти, установленной на компьютере, и может располагаться в любом месте физической памяти (кроме 1-го мегабайта, где размещаются программы MS-DOS, видеопамять и ПЗУ BIOS).

Почему селекторы на рис. 1.21 имеют такие значения? Как селектор связан с номером дескриптора дескрипторной таблицы?

Рассмотрим формат селектора, приведенный на рис. 1.22.

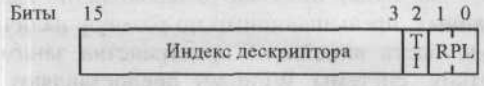


Рис. 1.22. Селектор дескриптора

Порядковый номер (индекс) дескриптора в таблице дескрипторов записывается в селектор начиная с бита 3, что эквивалентно умножению его на 8. Если вспомнить,

что дескрипторы имеют длину 8 байт, то станет ясно, что в селекторе (если не принимать в расчет 3 младших бита) хранится смещение дескриптора от начала таблицы дескрипторов.

Бит TI (Table Indicator, индикатор таблицы) равен нулю, если селектор относится к глобальной таблице, и единице, если соответствующий селектору дескриптор входит в локальную таблицу.

Поле RPL (Request Privilege Level, запрошенный уровень привилегий), занимающее биты 0...1 селектора, может принимать значения от 0 до 3 в соответствии с числом уровней привилегий процессора. Программы Windows, относящиеся к ядру операционной системы, выполняются на наиболее приоритетном уровне 0; менее ответственные системные программы, а также все приложения выполняются на самом низкоприоритетном уровне привилегий, равном трем.

Чтобы расшифровать значение селектора 0x21EF, назначенного системой сегменту команд приложения 1 (см. рис. 1.21), следует записать его в виде двоичного числа (рис. 1.23).

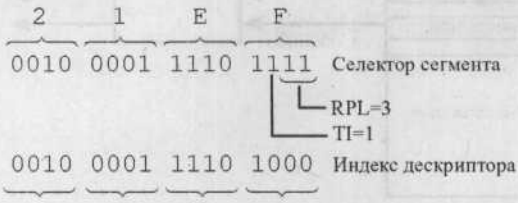


Рис. 1.23. Расшифровка значения селектора

Из рисунка видно, что уровень привилегий RPL=3, TI=1 (дескриптор принадлежит таблице LDT), а смещение дескриптора в LDT составляет 0x21E8, что соответствует номеру дескриптора 0x21E8/8 = 0x43D = 1085.

Любопытно отметить, что в системах Windows NT/2000 преобразование адресов 16-разрядных приложений осуществляется так же, однако в линейном адресном пространстве для них выделен не 3-й, а 1-й гигабайт.

Совершенно иначе обстоит дело с 32-разрядными приложениями Windows. Удивительным образом система предоставляет всем таким приложениям, сколько бы их ни было, один и тот же селектор для сегмента команд и один и тот же селектор для сегмента данных (рис. 1.24). Windows 98 назначает всем сегментам команд селектор 0x167, а сегментам данных – селектор 0x16F. В Windows NT/2000 выделяются соответственно селекторы 0x1B и 0x23. Далее, базы обоих сегментов равны нулю, а границы – 4 Гбайт–1. Другими словами, каждому запущенному приложению как бы предоставляется все линейное адресное пространство. Принято говорить, что такие приложения работают в модели плоской памяти (иногда для обозначения плоской памяти используют англоязычный термин "FLAT", являющийся, кстати, ключевым словом языка ассемблера).

Разумеется, никакое приложение не может использовать всю плоскую память. Во-первых, линейное адресное пространство – это, как уже отмечалось, фикция, а приложение, чтобы оно могло выполняться, должно находиться в оперативной памяти, объем которой обычно гораздо меньше 4 Гбайт (в действительности это, казалось бы, вполне очевидное утверждение не совсем правильно – как будет показано в дальнейших главах, приложение может работать с массивами данных, превышающими по размеру наличную оперативную память). Во-вторых, заметную часть линейного пространства занимают программы операционной системы. В-третьих, системы Windows предоставляют для прикладных 32-разрядных программ только половину адресного пространства, а именно 1-й и 2-й гигабайты, резервируя 3-й и 4-й гигабайты для системных программ.

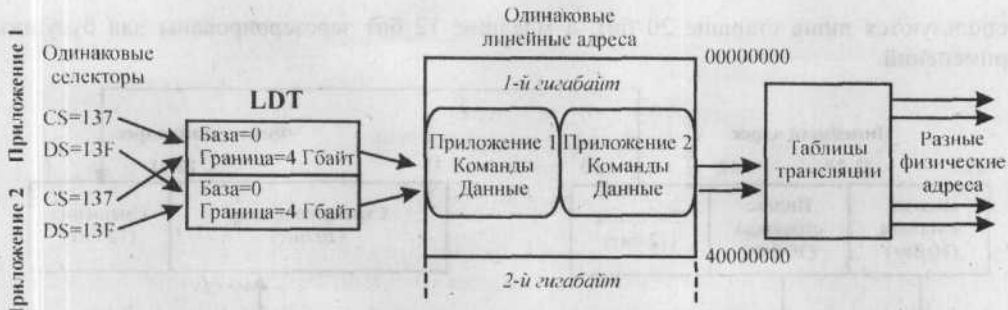


Рис. 1.24. Схема преобразования адресов для 32-разрядных приложений в Windows 98 (все числа шестнадцатеричные)

Поскольку базовые линейные адреса всех (впрочем, фактически всего двух) сегментов программы равны нулю, виртуальные смещения, с которыми работает программа, совпадают с линейными адресами. Другими словами, плоское *виртуальное* адресное пространство программы совпадает с плоским *линейным* адресным пространством. При этом все приложения используют одни и те же диапазоны линейных адресов. Для того чтобы при одинаковых линейных адресах различные приложения занимали отдельные участки физической памяти, не затирая друг друга, Windows при смене текущего приложения изменяет таблицы страничной трансляции, с помощью которых как раз и происходит преобразование линейных адресов в физические адреса оперативной памяти.

Страничная трансляция представляет собой довольно сложный механизм, в котором принимают участие аппаратные средства процессора и находящиеся в памяти таблицы страничной трансляции. Назначение и взаимодействие элементов системы страничной трансляции схематически изображены на рис. 1.25.

Система страничных таблиц состоит из двух уровней. На первом уровне находится каталог таблиц страниц (или просто каталог страниц) – резидентная в памяти таблица, содержащая 1024 4-байтовых поля с адресами таблиц страниц. На втором уровне находятся таблицы страниц, каждая из которых содержит также 1024 4-байтовых поля с адресами физических страниц памяти. Поскольку размер физической страницы составляет 4 Кбайт, 1024 таблицы по 1024 страницы перекрывают все адресное пространство (4 Гбайт).

Не все 1024 таблицы страниц должны обязательно иметься в наличии (кстати, они заняли бы в памяти довольно много места – 4 Мбайт). Если программа реально использует лишь часть возможного линейного адресного пространства, а так всегда и бывает, то неиспользуемые поля в каталоге страниц помечаются как отсутствующие. Для таких полей система, экономя память, не выделяет страничные таблицы. С другой стороны, каталог страниц (или страница каталога, как его иногда называют, поскольку весь он имеет размер 4 Кбайт) должен всегда находиться в физической памяти.

При включенной страничной трансляции линейный адрес рассматривается как совокупность трех полей: 10-битового индекса в каталоге страниц, 10-битового индекса в выбранной таблице страниц и 12-битового смещения в выбранной странице.

Старшие 10 бит линейного адреса образуют номер элемента в каталоге страниц. Базовый физический адрес каталога хранится в одном из управляющих регистров процессора, конкретно – в регистре CR3. Из-за того, что каталог сам представляет собой страницу и выровнен в памяти на границу 4 Кбайт, в регистре CR3 для адресации к каталогу

используются лишь старшие 20 бит, а младшие 12 бит зарезервированы для будущих применений.

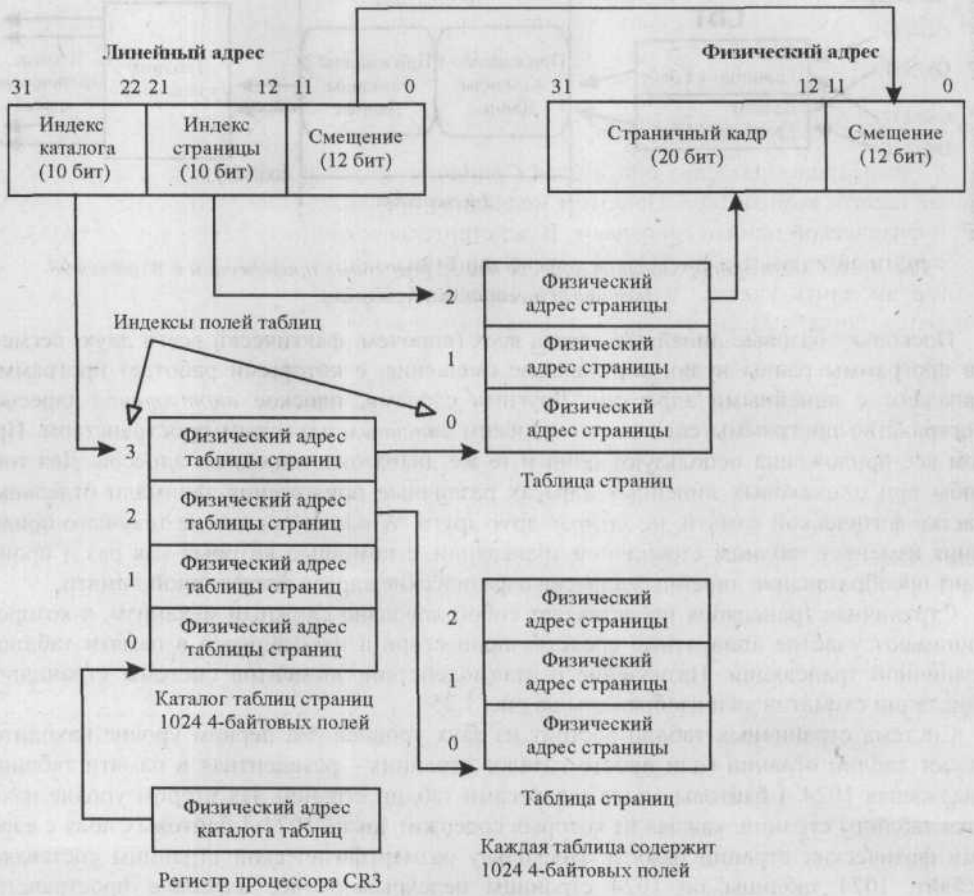


Рис. 1.25. Страничная трансляция адресов

Элементы каталога имеют размер 4 байта, поэтому индекс, извлеченный из линейного адреса, сдвигается влево на 2 бита (т. е. умножается на 4) и полученная величина складывается с базовым адресом каталога, образуя адрес конкретного элемента каталога. Каждый элемент каталога содержит физический базовый адрес одной из таблиц страниц, причем, поскольку таблицы страниц сами представляют собой страницы и выровнены в памяти на границу 4 Кбайт, и в этом адресе значащими являются только старшие 20 бит.

Далее из линейного адреса извлекается средняя часть (биты 12...21), сдвигается влево на 2 бита и складывается с базовым адресом, хранящимся в выбранном поле каталога. В результате образуется физический адрес страницы в памяти, в котором опять же используются только старшие 20 бит. Этот адрес, рассматриваемый как старшие 20 бит физического адреса адресуемой ячейки, носит название страничного кадра. Страничный кадр дополняется с правой стороны младшими 12 битами линейного адреса, которые проходят через страничный механизм без изменения и играют роль смещения внутри выбранной физической страницы. В результате каждые 4 Кбайт линейных адресов ото-

бражаются на 4 Кбайт физических, причем операционная система, изменяя содержимое таблиц трансляции адресов, может легко изменять отображение общего для всех программ линейного адресного пространства на требуемые участки физической оперативной памяти, содержащих элементы той или иной активной программы.

Система ввода-вывода

В предыдущих разделах обсуждались вопросы взаимодействия процессора и оперативной памяти компьютера. При этом молчаливо предполагалось, что понятия оперативной и физической памяти совпадают. В действительности это не так: в компьютере, кроме оперативной памяти, имеются и другие запоминающие устройства, для которых необходимо выделить участки физического адресного пространства. К таким устройствам относится, во-первых, видеопамять и, во-вторых, постоянные запоминающие устройства (ПЗУ).

Первые 640 Кбайт адресного пространства с адресами от 0x00000 до 0x9FFFF отводятся под основную оперативную память, которою еще называют стандартной или обычной. В реальном режиме в начало этого участка загружается MS-DOS, а все свободное пространство используется для загрузки программ. В защищенном режиме часть этого участка по-прежнему резервируется для DOS, что позволяет в рамках системы Windows организовывать множество одновременно работающих сеансов DOS.

Оставшиеся 384 Кбайт адресного пространства, иногда называемые верхней памятью, первоначально предназначались для размещения ПЗУ. Практически под ПЗУ занята только часть адресов. В самом конце адресного пространства, в области 0xF0000...0xFFFFF, располагается ПЗУ, в котором записана важная часть операционной системы – так называемая BIOS (от Basic In-Out System, базовая система ввода-вывода). В функции BIOS входит тестирование узлов компьютера при каждом его включении, а также управление штатной аппаратурой компьютера – клавиатурой, экраном, таймерами и т. д. Программы BIOS также инициируют начальную загрузку компьютера. ПЗУ BIOS располагается на системной плате компьютера и, таким образом, всегда входит в его состав.

Два диапазона физических адресов, 0xA0000...0xAFFFF и 0xB8000...0xBFFFF, отводится для адресации видеопамяти графической системы компьютера. Первый диапазон (64 Кбайт) используется для реализации графических режимов, второй (32 Кбайт) – текстовых. Физически видеопамять обычно находится на видеоплате, подключаемой к компьютеру через разъем расширения, хотя может быть встроена (вместе с контроллером терминала) в состав системной платы. Любопытно отметить, что, хотя современные графические режимы (до 16 млн. цветов и до 1280x1024 точек на экране) требуют большого объема видеопамяти, не менее нескольких мегабайт, для адресации этой памяти в адресном пространстве компьютера выделено всего 64 Кбайт. Обращение ко всей видеопамяти через это узкое адресное окно обеспечивает контроллер терминала, который обычно называют видеоадаптером.

Наконец, еще одна небольшая область адресов, обычно в диапазоне 0xC0000...0xCFFFF, отводится для адресации ПЗУ – расширений BIOS, предназначенных для обслуживания видеосистемы и дисков.

Реально на современных персональных компьютерах устанавливают оперативную память объемом 256 мегабайт и более. Вся память за пределами первого мегабайта называется расширенной и может использоваться только в защищенном режиме работы процессора. В реальном режиме, под управлением MS-DOS, эта память недоступна (за ис-

ключением небольшого участка объемом чуть меньше 64 Кбайт в самом начале расширенной памяти; этот участок носит специальное название области старшей памяти). На рис. 1.26 изображено типичное распределение адресного пространства компьютера.

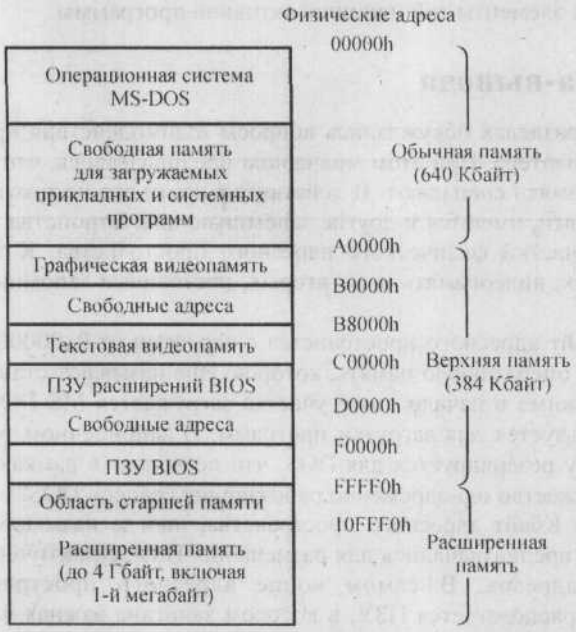


Рис. 1.26. Типичное распределение адресного пространства (серым цветом выделена оперативная память)

Области адресов, закрепленные за ПЗУ или видеопамятью, входят в общее с оперативной памятью адресное пространство, и к ним, следовательно, применимы все правила работы с оперативной памятью. В частности, для записи в видеопамять можно использовать весь набор команд процессора – пересылки, арифметических действий, побитовых операций и т. д. Можно также и читать содержимое видеопамяти или ПЗУ. При этом идентификация адресуемого устройства (оперативная память, видеопамять или ПЗУ) осуществляется по закрепленным за устройствами диапазонам адресов. Существенно, что эти диапазоны не перекрываются. Обращение по адресу, например, 0x00060 приведет нас к оперативной памяти, по адресу 0xA0060 – к видеопамяти, а по адресу 0xF0060 – к ПЗУ BIOS.

Как видно из рис. 1.26, в адресном пространстве компьютера не предусмотрены адреса для таких периферийных устройств, всегда входящих в состав компьютера, как клавиатура, жесткие и гибкие диски, последовательные и параллельные интерфейсы, и ряда других. В то же время очевидно, что для программного обращения к этим устройствам им должны быть присвоены некоторые адреса. Почему их нет на рис. 1.26? Ответ заключается в том, что эти устройства программируются особым образом, отличным от программирования памяти.

Все устройства компьютера можно по способу программного обращения к ним разбить на две категории. В первую входят устройства, упомянутые выше: оперативная и видеопамять, а также ПЗУ. Все они отличаются большим объемом хранимой информации и соответственно большим диапазоном используемых адресов. К другой категории

относятся все периферийные и многие внутренние устройства компьютера: многочисленные контроллеры подключаемой аппаратуры, КМОП-память, системные таймеры и т. д. Эти устройства не содержат в себе массовой памяти и требуют для работы лишь небольшого числа адресов. Обращение к таким устройствам осуществляется не через пространство памяти, а через так называемое пространство ввода-вывода.

Проблема идентификации устройств той и другой категории имеет два аспекта: аппаратный и программный. Рассмотрим чуть подробнее аппаратные особенности подключения устройств компьютера к системной шине (рис. 1.27).

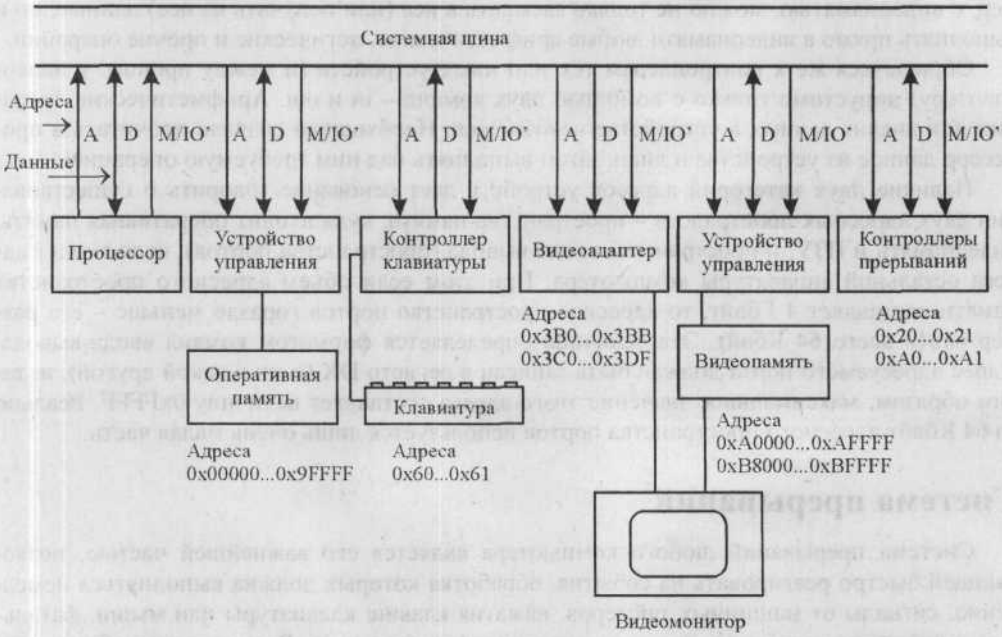


Рис. 1.27. Подключение устройств к пространствам памяти и ввода-вывода

Среди линий управления, входящих в системную шину, нас сейчас будет интересовать сигнал, носящий название M/IO' (M – "IO с отрицанием"). Этот сигнал, строго говоря, имеется только среди выходных сигналов процессора, а на системную шину приходят производные от этого сигнала, образованные как комбинации сигнала M/IO' с управляющими сигналами записи и чтения. Однако суть дела от этого не изменяется, и для простоты мы опустили эти подробности.

Сигнал M/IO' генерируется процессором в любой операции чтения или записи и, поступая во все устройства, подключенные к системной шине, разрешает или запрещает их работу. При этом для устройств, подключаемых через пространство памяти, разрешающим является значение сигнала M/IO'=1; для устройств, подключаемых через пространство ввода-вывода, – значение M/IO'=0. Таким образом осуществляется аппаратное разделение устройств типа памяти и устройств ввода-вывода.

Программное разделение устройств реализуется с помощью двух наборов команд процессора – для памяти и для устройств ввода-вывода. В первую группу команд входят практически все команды процессора, с помощью которых можно обратиться по тому или иному адресу памяти, – команды пересылки mov и movs, арифметических действий add, mul и div, сдвигов rol, ror, sal и sar, анализа содержимого test и многие другие. Вто-

рую группу команд образуют специфические команды ввода-вывода. В МП 86 их всего две – команда вывода out и команда ввода in; в 32-разрядных процессорах имеются модификации этих команд outs и ins для передачи в устройство или из него последовательности данных. При выполнении команд первой группы (обращения к памяти) процессор автоматически генерирует M/IO' = 1; при выполнении команд ввода-вывода процессор устанавливает сигнал M/IO' = 0.

Таким образом, при обращении к оперативной и видеопамяти программист может использовать все подходящие по смыслу команды процессора, при этом, работая, например, с видеопамятью, можно не только засылать в нее (или получать из нее) данные, но и выполнять прямо в видеопамяти любые арифметические, логические и прочие операции.

Обращаться же к контроллерам тех или иных устройств (и между прочим, к видеоадаптеру) допустимо только с помощью двух команд – in и out. Арифметические операции или анализ данных в устройстве невозможен. Необходимо сначала прочитать в процессор данное из устройства и лишь затем выполнять над ним требуемую операцию.

Наличие двух категорий адресов устройств дает основание говорить о существовании двух адресных пространств – пространства памяти, куда входит оперативная память, видеопамять и ПЗУ, и пространства ввода-вывода (пространства портов), куда входят адреса остальной аппаратуры компьютера. При этом если объем адресного пространства памяти составляет 4 Гбайт, то адресное пространство портов гораздо меньше – его размер равен всего 64 Кбайт. Эта величина определяется форматом команд ввода-вывода. Адрес адресуемого порта должен быть записан в регистр DX (и ни в какой другой), и, таким образом, максимальное значение этого адреса составляет величину 0xFFFF. Реально из 64 Кбайт адресного пространства портов используется лишь очень малая часть.

Система прерываний

Система прерываний любого компьютера является его важнейшей частью, позволяющей быстро реагировать на события, обработка которых должна выполняться немедленно: сигналы от машинных таймеров, нажатия клавиш клавиатуры или мыши, фатальные ошибки, возникающие при выполнении программы и пр. Рассмотрим в общих чертах компоненты этой системы.

Сигналы аппаратных прерываний, возникающие в устройствах, входящих в состав компьютера или подключенных к нему, поступают в процессор не непосредственно, а через два контроллера прерываний, один из которых называется ведущим, а второй – ведомым (рис. 1.28). В прежних моделях машин контроллеры представляли собой отдельные микросхемы; в современных компьютерах они входят в состав многофункциональной микросхемы периферийного контроллера.

Два контроллера используются для увеличения допустимого числа внешних устройств. Дело в том, что каждый контроллер прерываний может обслуживать сигналы лишь от восьми устройств. Для обслуживания большего количества устройств контроллеры можно объединять, образуя из них веерообразную структуру. В современных машинах устанавливают два контроллера, увеличивая тем самым возможное число входных устройств до 15 (7 у ведущего и 8 у ведомого контроллера).

К входным выводам IRQ1...IRQ7 и IRQ8...IRQ15 (IRQ – это сокращение от Interrupt Request, запрос прерывания) подключаются выводы устройств, на которых возникают сигналы прерываний. Выход ведущего контроллера подключается к входу INT микропроцессора, а выход ведомого – к входу IRQ2 ведущего. Основной функцие контроллеров является передача сигналов запросов прерываний от внешних устройств на единст-

венный вход прерываний процессора. При этом, кроме сигнала INT, контроллеры передают в процессор по линиям данных номер вектора, который образуется в контроллере путем сложения базового номера, записанного в одном из его регистров, с номером входной линии, по которой поступил запрос прерывания. Номера базовых векторов заносятся в контроллеры автоматически в процессе начальной загрузки компьютера. Как видно из рис. 1.28, базовые векторы различаются в разных операционных системах. Очевидно, что номера векторов аппаратных прерываний однозначно связаны с номерами линий, или уровнями IRQ, а через них – с конкретными устройствами компьютера. На рис. 1.28 обозначены основные устройства компьютера, работающие в режиме прерываний.

Рис. 1.28. Аппаратная организация прерываний

Процессор, получив по линии INT сигнал прерывания, выполняет последовательность стандартных действий, которую можно назвать процедурой прерывания. Подчеркнем, что здесь идет речь лишь о реакции самого процессора на сигналы прерываний, а не об алгоритмах обработки прерываний, предусматриваемых пользователем в обработчиках прерываний. Смысл процедуры прерывания заключается в том, чтобы сохранить состояние текущей (прерываемой) программы и передать управление обработчику, соответствующему возникшему прерыванию.

Процедура прерывания реального режима

Процедуры прерываний реального и защищенного режимов схожи в принципе, но различаются в деталях. Рассмотрим сначала процедуру прерывания реального режима (рис. 1.29).

В реальном режиме, т. е. при работе компьютера под управлением MS-DOS, самое начало физической оперативной памяти от адреса 0 до 0x3FF отводится под векторы прерываний – 4-байтовые области, в которых хранятся адреса обработчиков прерываний (ОбрПр на рис. 1.29). В 2 старших байта каждого вектора записывается сегментный адрес обработчика, в 2 младших – смещение точки входа в обработчик. Векторы, как и соответствующие им прерывания, имеют номера, причем вектор с номером 0 располагается начиная с адреса 0, вектор 1 – с адреса 4, вектор 2 – с адреса 8 и т. д. Вектор с номером n занимает, таким образом, байты памяти от n*4 до n*4+3. Всего в выделенной под векторы области памяти помещается 256 векторов.

Получив сигнал INT на выполнение процедуры прерывания, а также номер вектора, процессор сохраняет в стеке выполняемой программы текущее содержимое трех регистров процессора: регистра флагов, CS и IP. Два последних числа образуют полный адрес

возврата в прерванную программу, а всю сохраненную в стеке информацию иногда называют вектором прерванного процесса. Далее процессор загружает CS и IP из соответствующего вектора прерывания, осуществляя тем самым переход на обработчик прерывания, связанный с этим вектором.

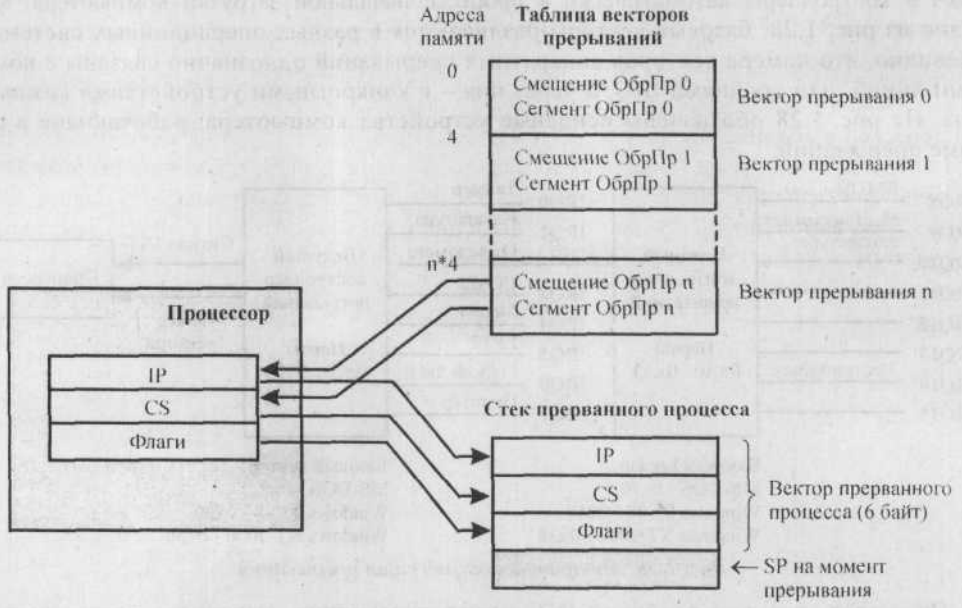


Рис. 1.29. Обслуживание прерывания в реальном режиме

Любой обработчик прерывания всегда заканчивается командой iret (interrupt return, возврат из прерывания), выполняющей обратные действия – извлечение из стека сохраненных там трех слов и помещение их назад в регистры IP, CS и флагов. Это приводит к возврату в основную программу в ту самую точку, где она была прервана.

В действительности запросы на обработку прерываний могут иметь различную природу. Помимо описанных выше аппаратных прерываний от периферийных устройств, называемых часто внешними, имеются еще два типа прерываний: внутренние и программные.

Внутренние прерывания возбуждаются цепями самого процессора при возникновении одной из специально оговоренных ситуаций, например при выполнении операции деления на 0 или при попытке выполнить несуществующую команду. За каждым из таких прерываний в самом процессоре закреплен определенный вектор. Например, за делением на 0 закреплен вектор 0, а за неправильной командой – вектор 6. Если процессор сталкивается с одной из таких ситуаций, он выполняет описанную выше процедуру прерывания, используя закрепленный за этой ситуацией вектор прерывания. При программировании в DOS с внутренними прерываниями приходится сталкиваться редко; иногда их действие проявляется в выводе на экран предупредительных сообщений. Так, если в приложение DOS включить операцию деления содержимого некоторой переменной на 0, то программа аварийно завершится, а на экран будет выведено сообщение вроде

```
integer divide by 0 [целочисленное деление на 0]
```

причем конкретный текст сообщения будет зависеть от использованного языка программирования. Очевидно, что вывод этого сообщения инициирует обработчик внутреннего прерывания с вектором 0.

Наконец, еще одним чрезвычайно важным типом прерываний являются программные прерывания. Они вызываются командой int с числовым аргументом, который рассматривается процессором как номер вектора прерывания. Если в программе на языке ассемблера встречается, например, команда

```
int    21h
```

то процессор выполняет ту же процедуру прерывания, используя в качестве номера вектора операнд команды int. Программные прерывания применяются в приложениях DOS в первую очередь для вызова системных обслуживающих программ – функций DOS и BIOS. Так, вызов любой функции DOS – чтения с клавиатуры, вывода на экран, записи на диск и пр. – осуществляется с помощью команды int 21h. Эта команда передает управление на обработчик прерывания 0x21, который является, по существу, входной программой операционной системы. Назначение этой программы (ее можно назвать диспетчером DOS) – выяснить, какую именно функцию DOS запросила прикладная программа, и осуществить вызов этой конкретной функции. Команда int 10h вызывает программу BIOS, управляющую терминалом, в функции которой входит вывод на экран символов и точек, задание видеорежима, загрузка в знакогенератор таблиц начертаний символов и вообще все действия, связанные с экраном. Команда int 13h также передает управление в BIOS, вызывая программу управления дисками – запись и чтение секторов, форматирование дорожек и т. д.

Важно подчеркнуть, что описанные действия процессора выполняются совершенно одинаково для всех видов прерываний – внутренних, аппаратных и программных, хотя причины, возбуждающие процедуру прерывания, имеют принципиально разную природу.

Прерывания и исключения защищенного режима

Как было показано в предыдущем разделе, в МП 86 (и соответственно, в реальном режиме) предусмотрены прерывания трех видов: внутренние, возникающие в самом микропроцессоре, внешние, поступающие в процессор от внешних устройств компьютера через контроллеры прерываний, и программные, инициируемые командой int. В защищенном режиме также возможны прерывания всех трех типов, при этом число внешних прерываний, определяемое количеством контроллеров прерываний в компьютере, осталось, естественно, тем же, однако функции внутренних прерываний и их количество существенно расширены. Внутренние прерывания, называемые здесь исключениями, исключительными ситуациями или особыми случаями (exceptions), являются важнейшим элементом организации защищенного режима.

Так же как и в реальном режиме, все прерывания защищенного режима имеют свои номера, причем их общее количество не должно превышать 256. Под исключения отданы первые 32 номера (0...31), хотя реально возникающие исключения имеют номера 0...17, а номера с 18-го по 31-й зарезервированы для будущих моделей процессоров.

В реальном режиме процессор при регистрации прерывания обращается к таблице векторов прерываний, находящейся всегда в самом начале памяти и содержащей двухсловные адреса обработчиков прерываний. В защищенном режиме аналогом таблицы векторов прерываний является таблица дескрипторов прерываний (IDT от Interrupt

Descriptor Table), которую создает операционная система защищенного режима. Таблица IDT содержит дескрипторы обработчиков прерываний, в которые, в частности, входят их адреса.

Если в таблицах дескрипторов памяти (LDT и GDT) находятся дескрипторы сегментов программы, то таблица дескрипторов прерываний IDT состоит из дескрипторов другого вида, которые называются шлюзами (вентилями). Через шлюзы осуществляется доступ к обработчикам прерываний и исключений. Шлюз, как и дескриптор памяти, занимает 8 байт и в него входит полный адрес (16-разрядный селектор и 32-разрядное смещение) обработчика данного прерывания.

Процессор, зарегистрировав то или иное исключение, сохраняет в стеке содержимое расширенного регистра флагов, селектор сегмента команд, смещение точки возврата, а также (в некоторых случаях) 32-битовый код ошибки. Таким образом, в отличие от реального режима, где в результате прерывания стек смещается на 3 слова, в защищенном режиме в стек заносятся 3 или даже 4 *двойных* слова. Код ошибки, если он есть, должен быть снят со стека обработчиком соответствующего исключения.

Сохранив вектор прерванного процесса, процессор по номеру вектора извлекает из IDT шлюз, определяет адрес обработчика и передает ему управление (рис. 1.30). Обработчик заканчивается командой iret, возвращающей управление в прерванную программу. Поскольку стек в результате прерывания сместился на 3 двойных слова, команда iret, которая должна вернуть стек в исходное (перед прерыванием) состояние, в 32-разрядной программе снимает со стека не 6 байт, как в реальном режиме, а 12.

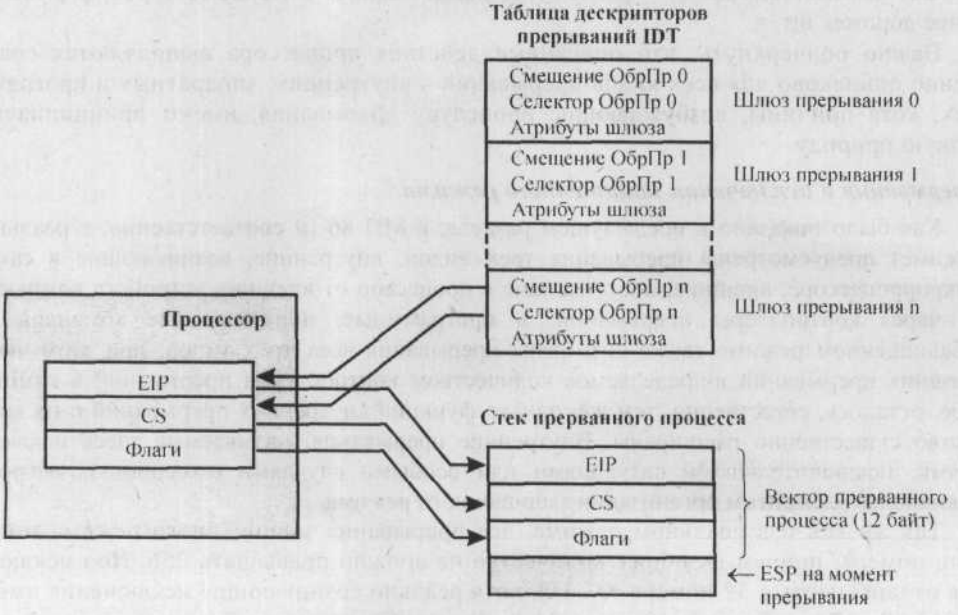


Рис. 1.30. Обслуживание прерывания в защищенном режиме

Как уже говорилось выше, различные устройства компьютера (таймер, клавиатура и др.) сигнализируют о необходимости программного вмешательства в их работу с помощью сигнала прерывания, который, пройдя через контроллер прерываний, поступает на вход INT микропроцессора и инициирует в нем выполнение процедуры прерывания.

В состав этой процедуры входит, в частности, чтение номера вектора, устанавливаемого контроллером прерываний на шине данных компьютера. При этом контроллер прерываний формирует передаваемый в процессор номер вектора путем сложения базового номера, хранящегося в контроллере, с номером линии, по которой поступил запрос от устройства. Номер базового вектора (в принципе в диапазоне 0...255) устанавливается в процессе программной инициализации контроллера. Поскольку в защищенном режиме векторы 0...31 зарезервированы за исключениями, базовые векторы контроллеров прерываний должны быть расположены в диапазоне 32...255. Таким образом, для обработки внешних аппаратных прерываний в защищенном режиме необходимо перепрограммировать контроллеры прерываний компьютера.

Нарушения в работе программы, приводящие к исключениям, могут иметь разную природу и разные возможности исправления в процессе выполнения программы. В соответствии с этим исключения подразделяются на 3 класса: нарушения, ловушки и аварии.

Нарушение, или отказ, (fault) – это исключение, фиксируемое еще до выполнения команды или в процессе ее выполнения. Типичными примерами нарушений являются адресация за установленной границей сегмента или обращение к отсутствующему дескриптору. При обработке нарушения процессор сохраняет в стеке адрес той команды, выполнение которой привело к исключению. При этом предполагается, что в обработчике нарушения его причина будет ликвидирована, после чего команда iret вернет управление на ту же, еще не выполненную команду. Таким образом, сам механизм обработки нарушений предполагает восстановление этого программного сбоя.

Ловушка (trap) обрабатывается процессором после выполнения команды, вызвавшей это исключение, и в стеке сохраняется адрес не этой, а следующей команды. Таким образом, после возврата из обработчика ловушки выполняется не команда, инициировавшая исключение, а следующая за ней команда программы. К ловушкам относятся все команды программных прерываний int.

Авария, или выход из процесса, (abort) является следствием серьезных невосстановимых ошибок, например обнаружение в системных таблицах неразрешенных или несовместимых значений. Адрес, сохраняемый в стеке, не позволяет локализовать вызвавшую исключение команду, и восстановление программы не предполагается. Обычно аварии требуют перезагрузки системы.

В табл. 1.1 приведен выборочный перечень исключений с краткими характеристиками.

Таблица 1.1. Исключения процессора

Вектор	*Название исключения*	*Класс исключения*	*Код ошибки*	*Команды, вызывающие исключение*
0	Ошибка деления	Нарушение	Нет	div, idiv
3	int 3	Ловушка	Нет	int 3
4	Переполнение	Ловушка	Нет	into
5	Нарушение границы массива	Нарушение	Нет	bound
6	Недопустимый код команды	Нарушение	Нет	Любая команда
8	Двойное нарушение	Авария	Да	Любая команда

Окончание табл. 1.1

Вектор	*Название исключения*	*Класс исключения*	*Код ошибки*	*Команды, вызывающие исключение*
10	Недопустимый сегмент состояния задачи TSS	Нарушение	Да	jmp, call, iret, прерывание
11	Отсутствие сегмента	Нарушение	Да	Команда загрузки сегментного регистра
12	Ошибка обращения к стеку	Нарушение	Да	Команда обращения к стеку
13	Общая защита	Нарушение	Да	Команда обращения к памяти
14	Страничное нарушение	Нарушение	Да	Команда обращения к памяти

В 16-разрядных приложениях Windows ошибки программирования чаще других приводят к двум типам исключений: ошибке обращения к стеку (исключение 12) и ошибке общей защиты (исключение 13). Исключение 12 возникает при попытке записать в заполненный стек или прочитать из пустого. Исключение 13 носит довольно универсальный характер и генерируется во многих случаях: загрузка в сегментный регистр несуществующего селектора, обращение к ячейке памяти за пределами сегмента данных, попытка записи в сегмент команд (в системах Windows сегмент команд объявляется с доступом для исполнения и чтения, но не записи) и в других случаях. Поскольку в дескрипторах сегментов указаны их границы, процессор при выполнении каждой команды чтения или записи памяти проверяет, не выходит ли смещение адресуемой ячейки за указанную в соответствующем дескрипторе границу. Попытка обращения к ячейке памяти за пределами сегмента приводит к исключению.

По-другому обстоит дело в 32-разрядных приложениях. Как мы видели, в таких приложениях используется модель плоской памяти, в которой все сегменты имеют размер 4 Гбайт. В этом случае нельзя выйти за пределы сегмента; однако программе реально выделяется в оперативной памяти (в процессе отображения линейных адресов на физические) лишь столько места, сколько она фактически занимает. Значительный выход за пределы объявленного в программе массива приведет к попытке обращения к ячейке оперативной памяти, находящейся за пределами адресов, отображенных на линейное пространство программы. В этом случае будет сформировано страничное нарушение, наиболее распространенный тип исключения при выполнении 32-разрядных программ.

Глава 2
Интегрированные среды разработки

В первые годы существования системы Windows разработка прикладных программ для этой системы выполнялась приблизительно так же, как и для системы MS-DOS: в каком-либо текстовом редакторе создавались исходные тексты приложения, после чего в сеансе DOS последовательным вызовом с командной строки программ компиляции и компоновки исходные файлы объединялись и преобразовывались в выполнимый файл с расширением .EXE, который уже можно было запустить в среде Windows. В случае обнаружения ошибок всю процедуру, начиная с редактирования исходных текстов, приходилось повторять заново. При этом команды вызова программ компиляции и компоновки оказывались весьма сложными, так как должны были содержать большое количество ключей и параметров.

В дальнейшем этот процесс несколько упростился введением так называемых MAKE-файлов – текстовых файлов с расширением .MAK, в которых в специальном формате указывалась вся информация, необходимая для создания исполнимого файла приложения: имена рабочих файлов, пути к каталогам с используемыми библиотеками, некоторые характеристики создаваемого приложения и др. При использовании этой методики процедура подготовки исполнимого файла приложения сводится к вызову программы MAKE.EXE (или NMAKE.EXE) с указанием в качестве параметра имени созданного заранее MAKE-файла. Как и ранее, подготовка исполнимого файла приложения осуществляется в сеансе DOS, а его запуск и отладка – в среде Windows.

В настоящее время приложения Windows готовятся, как правило, с помощью интегрированной среды разработки (Integrated Development Environment, IDE), которая обеспечивает весь цикл разработки приложения (подготовку исходного текста, компиляцию, отладку и пробное исполнение) в полноэкранном интерактивном режиме в среде Windows. При этом сама IDE широко использует стандартные средства графического интерфейса Windows – меню, диалоговые окна, инструментальные кнопки и пр.

Существует большое количество программных пакетов интегрированных сред разработки приложений, ориентирующихся на различные языки программирования: Бейсик, Паскаль, C++ и др. Эти пакеты постоянно модернизируются и совершенствуются, что отражается в номерах их версий. Как уже отмечалось, в процессе подготовки книги мы использовали две среды разработки: Borland C++ 5.02 и Microsoft Visual C++ 6.0, хотя, разумеется, читатель может воспользоваться и другими версиями этих IDE или вообще другими программными продуктами того же назначения.

Интегрированная среда разработки Borland C++

IDE Borland C++ 5.02 содержит большое количество настраиваемых параметров, доступ к которым осуществляется в основном через команду Options главного меню (рис. 2.1). К счастью, о большей части этих параметров можно не заботиться, так как их значения по умолчанию устанавливаются разумным образом. Однако кое-что все же придется настроить и проверить.

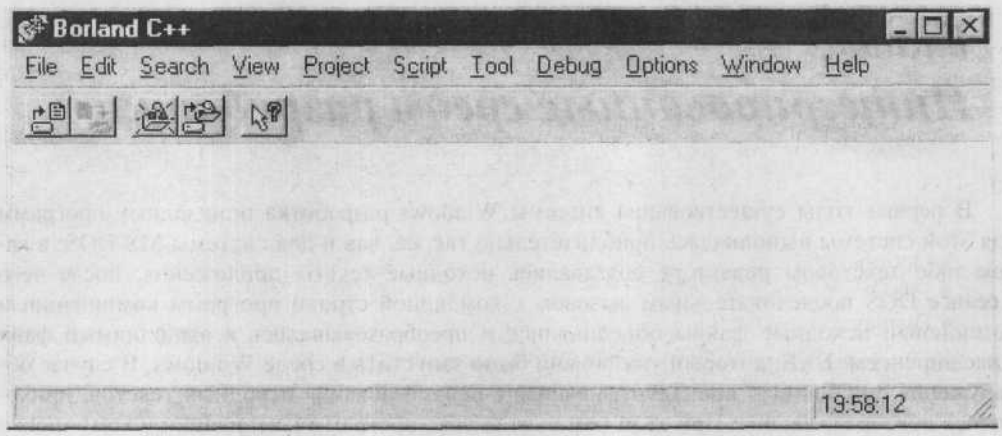


Рис. 2.1. Начальный кадр IDE Borland C++ 5.0

Прежде всего следует установить имена каталогов пользователя, в которых будут храниться исходные (Source), промежуточные (Intermediate) и конечные (Final) файлы, возникающие при разработке программного проекта. Доступ к кадру задания каталогов осуществляется выбором команды Options➢Project➢Directories. На рис. 2.2 для всех файлов программного проекта задан один каталог F:\EXAMPLES.

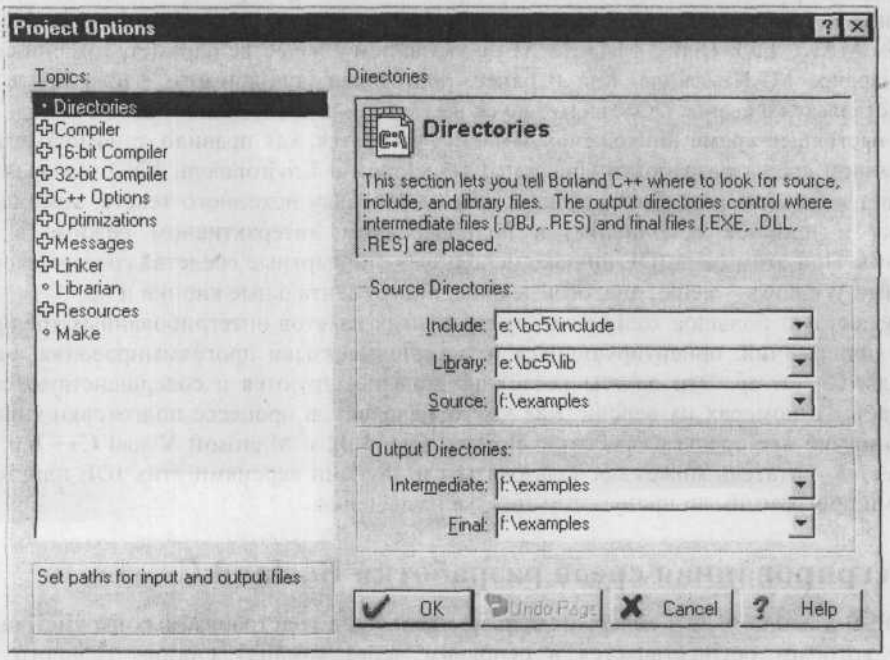


Рис. 2.2. Настройка каталогов пользователя

В том же кадре стоит проверить правильность задания каталогов для включаемых (Include) и библиотечных (Library) файлов. Если IDE установлена стандартным образом, то для этих файлов должны быть указаны каталоги \BC5\INCLUDE и \BC5\LIB.

Далее полезно выбрать пункт Options➢Environment➢Editor➢Display и удостовериться, что установлен шрифт с русскими буквами (например, Courier New Cyr), и заодно задать удобный для пользователя размер шрифта. Если после выбора пункта Options➢Environment в перечне пунктов следующего уровня пункт Editor окажется свернутым (о чем будет свидетельствовать знак + перед ним), его следует развернуть, щелкнув по нему дважды левой клавишей мыши или щелкнув один раз по знаку +. Строка +Editor развернется в список

—Editor
- Options
- File
- Display

из которого уже можно выбрать пункт Display для настройки шрифта.

Для подготовки исходного текста программы выберите команду File➢New➢Text Edit. Поскольку IDE дает новому файлу не очень благозвучное имя NONAME00.CPP и к тому же пытается расположить его в каталоге \BC5\BIN, лучше всего сразу же сохранить пустой пока файл под удобным для пользователя именем в его рабочем каталоге, что выполняется, как обычно, с помощью пункта меню File➢Save As.

Как только на экране IDE появляется окно для исходного текста программы, кадр IDE дополняется целым рядом новых управляющих кнопок, служащих для редактирования, запуска и отладки программы. Назначение наиболее важных кнопок указано на рис. 2.3.

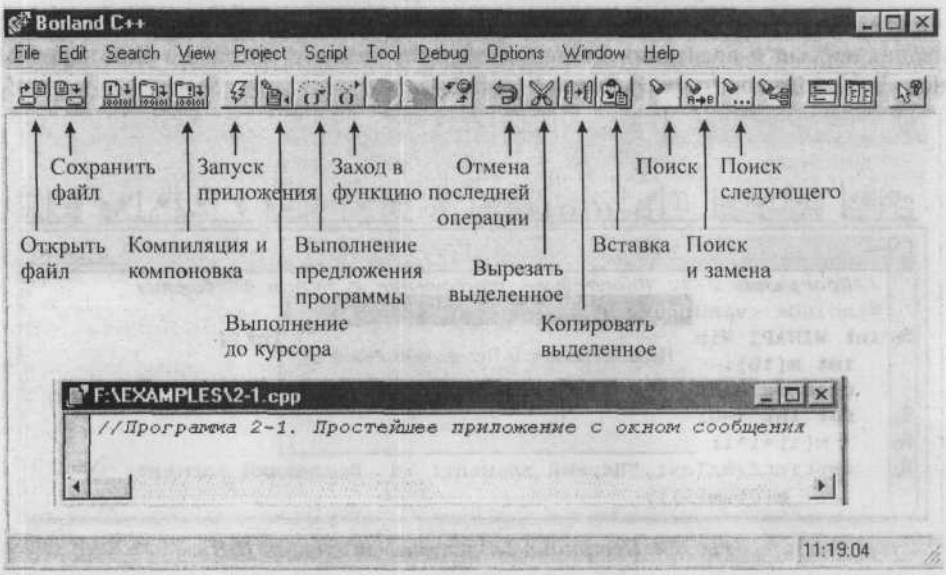


Рис. 2.3. Кадр IDE с окном редактирования

Введите текст программы 2-1 и сохраните его на диске под именем 2-1.cpp. Приводимый ниже текст снабжен комментариями, которые, разумеется, при вводе программы можно опустить. Комментарий в программах на языке C++ либо предваряется двумя знаками деления //, если он располагается в конце строки:

```
#include <windows.h>// Директива препроцессора
```

либо заключается в пары знаков /* и */, например:

```
/*Главная функция WinMain*/
```

Для удобства дальнейших ссылок мы перенумеровали содержательные строки программы, включив номера в состав комментариев. В большинстве случаев каждое предложение языка пишется на отдельной строке, хотя если предложение слишком длинное, его можно разместить на нескольких строках, при этом перенос на следующую строку допустим практически в любом месте предожения. Так, предложение с вызовом функции wsprintf() размещено нами на трех строках текста, с (7) по (9).

Каждое предложение языка должно заканчиваться знаком точки с запятой ";" (это не относится к директивам препроцессора, см. строку (1)).

```
/*Программа 2-1. Простейшая программа с окном сообщения*/
#include <windows.h>//(1) Директива препроцессора
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){//(2) Заголовок функции
  int m[10];//(3) Объявление массива из 10 целых чисел
  char szText[100];//(4) Объявление символьного массива
  for(int i=0;i<=9;i++)//(5) Цикл из 10 шагов
    m[i]=i*i;//(6) Тело цикла - занесение в массив квадратов чисел
  wsprintf(szText,//(7) Переменная, в которой формируется выводимая строка
     "Первый элемент: %d  Последний элемент: %d",//(8) Форматы преобразования
      m[0],m[9]);//(9) Перечень выводимых переменных
  MessageBox(NULL,szText,"Info",MB_OK);//(10) Вывод окна сообщения
  return 0;//(11) Возврат из главной функции в Windows
  }//(12) Скобка, завершающая текст главной функции
```

Приведенная программа формирует массив квадратов десяти целых чисел от 0 до 9 и выводит первый и последний элементы сформированного массива в окно, предназначенное для вывода коротких сообщений (рис. 2.4).

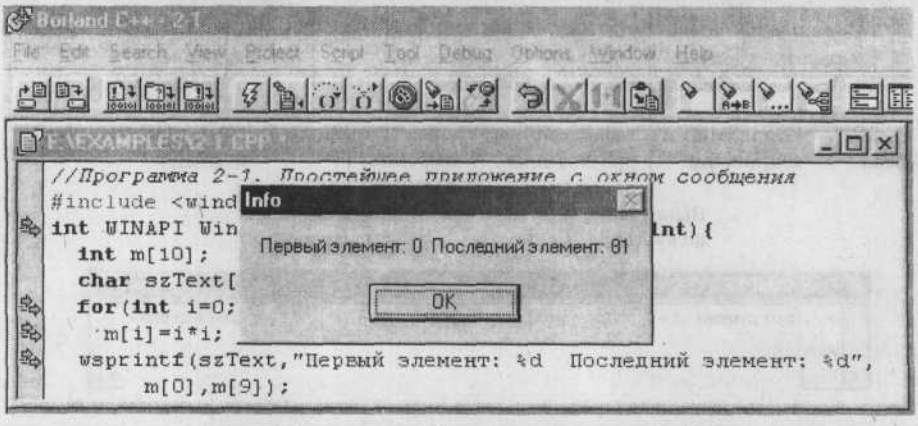


Рис. 2.4. Программа 2-1, запущенная из среды IDE

Детали программы (объявления переменных, предложения цикла и т. д.) будут проясняться по мере изучения разделов книги; сейчас мы ограничимся лишь беглым обзором ее структуры и функционирования.

В качестве первого предложения программы выступает директива препроцессора #include, с помощью которого к тексту программы подключается файл windows.h. Этот входящий в систему программирования файл содержит целый ряд определений, необходимых для правильной работы компилятора. В частности, в него входят определения

специфических для Windows и отсутствующих в языке C++ типов переменных (в нашем примере это HINSTANCE и LPSTR), констант (у нас это NULL и MB_OK) и функций Windows (wsprintf() и MessageBox()). Предложение

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

должно включаться в любую программу, предназначенную для выполнения в системе Windows.

Вслед за предложением #include идет собственно текст программы; в нашем примере весь он входит в главную функцию с именем WinMain().

При запуске приложения Windows управление всегда передается функции WinMain(), которая, таким образом, должна присутствовать в любой программе. В нашей простой программе фунцкция WinMain() является единственной; если же в программу входят какие-то другие функции (подпрограммы), написанные программистом, все они должны так или иначе вызываться из главной функции WinMain().

Главная функция начинается с заголовка, имеющего стандартный вид, за которым следует тело функции – программные предложения, которые мы желаем выполнить. В принципе в главную функцию можно включать любые действия – математические вычисления, чтение и обработку файлов с данными, команды управления автоматизированной установкой и пр. Все тело главной (как, впрочем и любой другой) функции заключается в фигурные скобки, причем последним предложением главной функции должно быть предложение

```
return 0;
```

которое завершает выполнение программы и передает управление в Windows.

В начале главной функции объявляются два массива – массив m из десяти целых (типа int) чисел и символьный (тип char) массив размером в 100 символов. Компилятор, встретившись с объявлениями переменных, выделяет под них участки памяти требуемого размера, однако в этих переменных пока нет ничего полезного – они заполнены тем, что случайно осталось в выделенных для них участках памяти от предыдущих программ. Такое случайное содержимое не инициализированных еще переменных иногда называют "мусором".

Предложения (5) и (6) образуют цикл, в котором последовательно выполняются 10 шагов. Шаги отсчитываются с помощью переменной i, служащей счетчиком цикла. В каждом шаге цикла в элемент массива m с номером, совпадающим со значением i (этот элемент обозначается m[i]), засылается произведение i на i, т. е. i^2. После этого значение i увеличивается на 1 и шаги цикла повторяются до тех пор, пока i не станет больше 9. На этом выполнение цикла прекращается и программа переходит к выполнению следующего предложения.

Числа, образующие массив m, не могут быть выведены на экран непосредственно – сначала их надо преобразовать в символьную форму, т. е. для каждого числа сформировать последовательность кодов, которая приведет к появлению на экране изображений цифр, образующих отображаемое число. Например, чтобы вывести на экран число 81, следует послать на экран коды 0x38 (символьный код цифры 8) и 0x31 (код цифры 1). Для вывода же дробного, да еще отрицательного числа – 8.1, символьная последовательность должна иметь вид 0x2D, 0x38, 0x2E, 0x31. Здесь код 0x2D соответствует изображению знака минус, а код 0x2E – знаку точки.

Для преобразования чисел в символьную форму удобно использовать функцию Windows wsprintf(). При вызове этой функции (строки (7)...(9)) в качестве ее первого аргумента следует указать имя достаточно длинного символьного массива (у нас это szText),

в который функция wsprintf() и зашлет результат преобразования. Вторым аргументом функции служит символьная строка, в которой перечисляются *форматы* вывода интересующих нас переменных. Число форматов должно соответствовать числу выводимых переменных. В рассматриваемом примере используются два формата %d, которые определяют вывод значений двух переменных (у нас это m[0] и m[9]) в форме целых десятичных чисел. Строка с перечнем форматов может содержать любые поясняющие тексты, которые также появятся на экране. Вся строка с форматами заключается в кавычки (строка (8) нашей программы).

Последнее действие программы – вывод на экран сформированной функцией wsprintf() символьной строки. В системе Windows нельзя (или, лучше сказать, нецелесообразно) выводить какую бы то ни было информацию непосредственно на экран, на котором всегда должен быть отображен Рабочий стол Windows. Любая информация должна выводиться в отдельное окно, закрепляемое за выполняемой программой. В дальнейшем мы рассмотрим методику создания "полноценного" окна с рамкой, заголовком, меню и стандартными кнопками управления (см., например, кадр IDE на рис. 2.3). Сейчас мы ограничимся выводом интересующей нас информации в так называемое окно сообщения, которое формируется функцией Windows MessageBox(). Среди аргументов этой функции указывается имя символьного массива, который мы хотим вывести (у нас это szText), а также произвольный заголовок окна сообщения (у нас – "Info").

Приложение Windows с окном сообщения в качестве главного и единственного окна приложения имеет весьма ограниченные возможности, но чрезвычайно удобно для изучения языка или алгоритмов программирования. Мы будем использовать такую форму приложения Windows на протяжении всей главы книги, посвященной основам языка C++.

Вернемся к подготовке приложения. Введя текст программы и сохранив его в файле 2-1.CPP, следует приступить к созданию проекта. Вообще проект служит для объединения в единый комплекс всех файлов, входящих в приложение. Обычно исходные тексты даже простого приложения образуют не один, а по меньшей мере 3 файла: файл .CPP с исходным текстом программы, файл ресурсов .RC, в котором, в частности, описывается конфигурация меню и диалоговых окон, и файл определения модуля .DEF с информацией о типе загрузочного модуля, атрибутах программных сегментов и некоторой другой информацией. Более сложные программы могут содержать несколько файлов типа .CPP, которые должны транслироваться и компоноваться совместно. Файл проекта (в системе Borland C++ он имеет расширение .IDE) как раз и служит для объединения всех файлов, относящихся к данному приложению.

Для нашей программы, в которой нет ни меню, ни диалогов, не нужен файл ресурсов; что же касается файла определений, то создавать его тоже нет необходимости, так как вполне можно воспользоваться стандартным файлом DEFAULT.DEF, имеющимся в системе. Однако создать проект все-таки необходимо, так как именно в проекте указываются некоторые важные характеристики приложения, без знания которых IDE не сможет создать загрузочный модуль приложения.

Выберите команду File➢New➢Project и в верхней части открывшейся панели с именем New Target компонента IDE Target Expert укажите имя файла проекта в рамке Project Path and Name. Это имя может быть любым, но удобнее для всех составляющих проекта иметь одно имя (при разных расширениях). Поэтому укажите имя 2-1.IDE, предварив его полным путем к вашему рабочему каталогу (рис. 2.5).

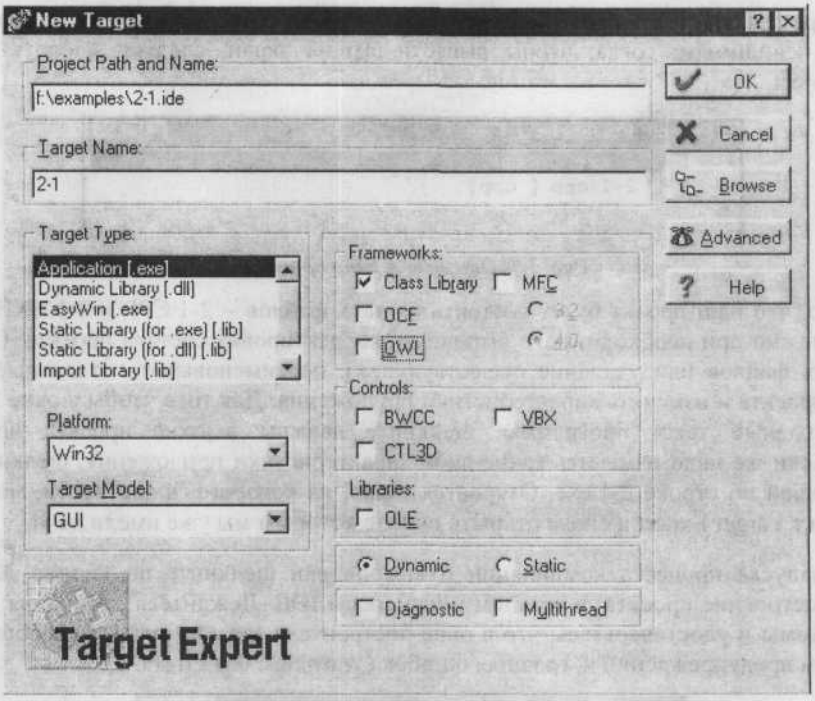


Рис. 2.5. Панель Target Expert

Удостоверьтесь, что в рамке Platform установлен режим Win32 (32-разрядное приложение Windows), а в рамке Target Model – GUI (Graphical User Interface, графический интерфейс пользователя). С помощью рамки Platform можно также при желании создать 16-разрядное приложение Windows или приложение DOS, а рамка Target Model позволяет задать особый тип создаваемого приложения – Console, т. е. консольное приложение, которое представляет собой программу, предназначенную для выполнения в сеансе DOS. Заметим, что, хотя консольное приложение пишется по правилам обычных программ DOS, выполнить его можно только в среде Windows. При запуске такого приложения на рабочем столе Windows создается окно сеанса DOS, в котором и работает запущенная программа. Нас консольные приложения интересовать не будут.

Нажмите кнопку Advanced. В открывшейся панели Advanced Options снимите флажки (щелчком левой клавиши мыши) с кнопок выбора файлов .rc и .def, так как этих файлов у нас нет (рис. 2.6). Подтвердите настройки, выбрав OK в панели Advanced Options и еще раз OK в панели Target Expert.

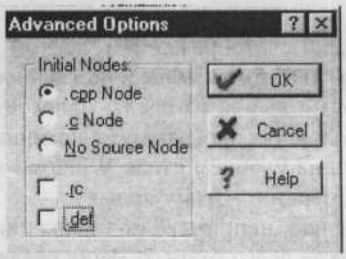


Рис. 2.6. Окно задания состава проекта

В кадре IDE появится окно описания проекта Project (рис. 2.7). Иногда это окно оказывается невидимым; тогда, чтобы вывести его на экран, следует выбрать команду View➢Project.

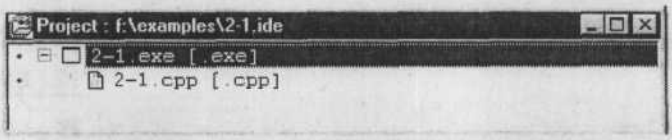


Рис. 2.7. Открытое окно проекта

Видно, что наш проект будет состоять из двух файлов – 2-1.CPP и 2-1.EXE. В окне проекта можно при необходимости выполнять корректировку состава проекта (добавление новых файлов или удаление существующих), переименовывать отдельные составляющие проекта и изменять характеристики приложения. Для того чтобы в окне IDE появился исходный текст программы, щелкните дважды в окне проекта по строке 2-1.cpp. Если же надо изменить какие-либо характеристики приложения, щелкните *правой* клавишей по строке 2-1.exe. Откроется меню, из которого, в частности, можно выбрать пункт Target Expert и снова открыть окно, с которым мы уже имели дело.

Для запуска процесса компиляции и компоновки щелкните по кнопке  Build Project (построение проекта) в главном меню среды IDE. Дождитесь окончания обработки программы и удостоверьтесь, что в окне построителя задачи (рис. 2.8) сообщается об отсутствии предупреждений и, главное, ошибок (Warnings: 0 и Errors: 0).

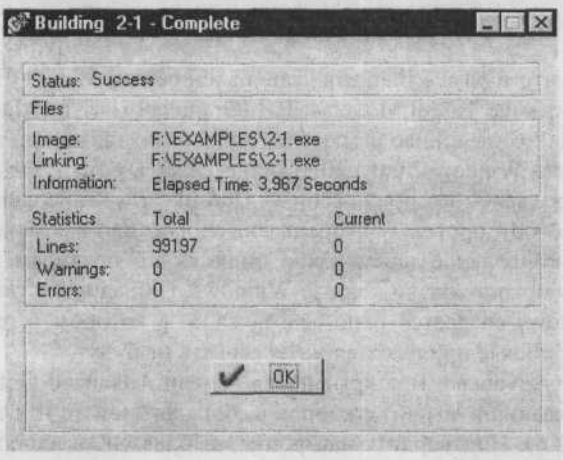


Рис. 2.8. Построитель задач сообщает об отсутствии ошибок

Запустите приложение, щелкнув по кнопке  с изображением желтой молнии в главном меню среды IDE. Поверх экранного кадра IDE будет выведено главное окно запущенного приложения (см. рис. 2.4). Теперь можно выйти из среды проектирования и при желании запускать приложение, как и любую другую программу, с помощью пункта пускового меню Выполнить. Можно также создать на Рабочем столе Windows ярлык для этой программы и запускать ее двойным щелчком мыши по этому ярлыку.

При повторной загрузке IDE Borland C++ с целью, например, модификации и исследования созданного приложения, необходимо прежде всего открыть файл созданного ра-

нее проекта. Это делается с помощью команды Project➢Open project... (рис. 2.9). Если просто открыть исходный файл программы и, не открывая файла проекта, попробовать выполнить компиляцию и запуск, вы получите сообщение об ошибках. Чтобы исправить дело, надо будет открыть файл проекта и повторить компиляцию.

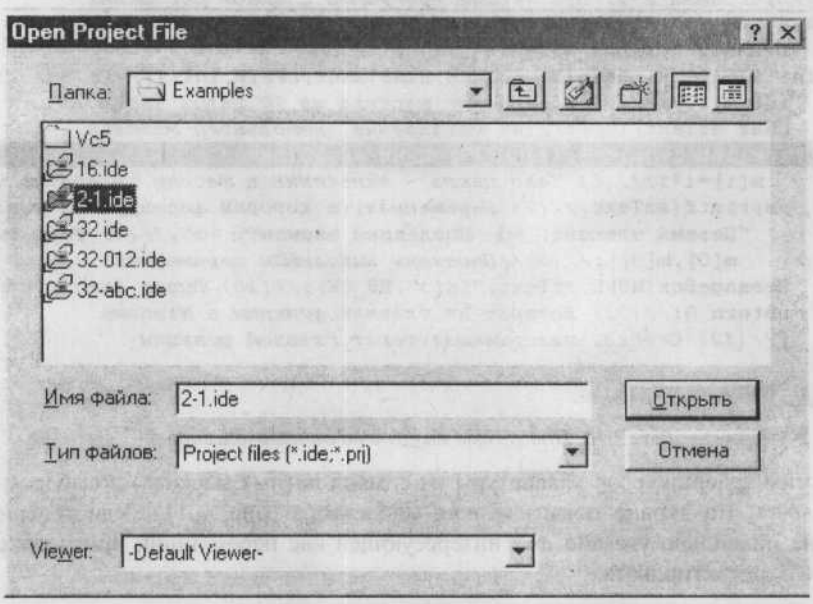


Рис. 2.9. Окно для выбора открываемого проекта

Отладка программ в IDE Borland C++

Как уже отмечалось, интегрированные среды разработки имеют встроенные средства отладки разрабатываемых программ. В IDE Borland C++ 5.02 предусмотрено два режима отладки: пошаговый контроль выполнения программы непосредственно в кадре IDE и вызов полноэкранного отладчика TD32.EXE (для отладки приложений в состав пакета включена отдельная программа TD.EXE). Рассмотрим сначала некоторые возможности первого режима.

Отладка программы и изучение ее работы заключается главным образом в ее пошаговом выполнении с контролем текущих значений включенных в состав программы переменных, а также значений, возвращаемых вызываемыми в программе функциями Windows. Покажем, как это делается, с помощью программы 2-1.

Выше уже отмечалось, что переменные, не инициализированные какими-либо конкретными значениями, могут содержать что угодно. Убедимся в этом с помощью отладчика, для чего остановим выполнение программы после объявления интересующих нас переменных, но до их программного заполнения.

Указание точки в программе, до которой ее надо выполнить, осуществляется помещением в начало соответствующей строки курсора (не курсора мыши в виде вертикальной черты с "полочками" I, а курсора клавиатуры, иногда называемого "каре" (знак ▌). Поместим курсор перед строкой 5 и нажмем кнопку  (можно также воспользоваться

клавишей F4). Программа выполнится до указанного места и остановится (рис. 2.10). При этом выполнятся лишь два предложения – объявление переменных m и szText.

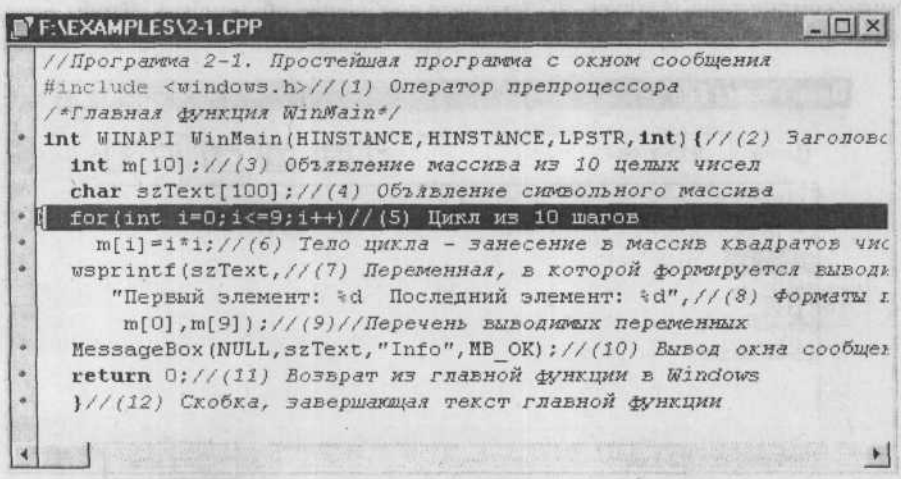


Рис. 2.10. Программа выполнена до предложения 5

Поместим теперь курсор клавиатуры на символ m (имя массива) и выберем команду Debug➢Inspect. На экране появится окно инспектора (рис. 2.11). Удостоверимся, что в этом окне правильно указано имя интересующей нас переменной; при необходимости имя можно отредактировать.

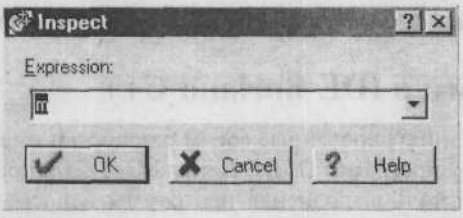


Рис. 2.11. Окно инспектора для задания имени переменной

Щелкнем по кнопке OK. На экран будет выведено окно инспектора с содержимым интересующей нас переменной. Поскольку переменная m представляет собой массив, в окне появятся текущие значения всех ее элементов. Повторим ту же операцию с массивом szText; на экран выведется второе окно инспектора (рис. 2.12).

Видно, что оба массива содержат в части своих элементов нули, а в части – произвольные бессмысленные значения. Проведенный эксперимент наглядно подтверждает одно из важнейших правил программирования: никогда не пользуйтесь неинициализированными переменными. Не надейтесь, что по умолчанию в них будут нулевые значения – мы видим, что это совсем не так.

Следующим этапом отладки будет проверка заполнения массива m квадратами чисел. Следует заметить,что при работе с массивами легко допустить ошибку, в результате которой либо массив будет заполнен не целиком, либо, наоборот, в память будет записано лишнее число, что, скорее всего, фатально отразится на работоспособности программы, поскольку это лишнее число с большой вероятностью затрет какое-то другое данное нашей программы.

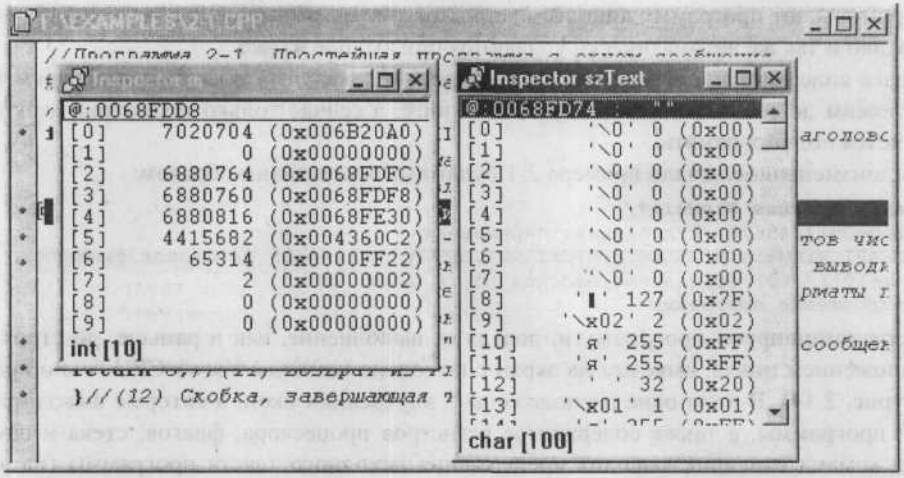


Рис. 2.12. Начальные значения элементов массивов m и szText

Остановим программу на строке (7) и посмотрим, как изменилось содержимое окна инспектора с массивом m (если вы удалили окна инспектора, выведите их на экран снова командой Debug➢Inspect) (рис. 2.13). Мы видим, что все 10 элементов массива m правильно заполнились квадратами последовательных целых чисел. В то же время массив szText по-прежнему заполнен "мусором", так как мы еще не дошли до выполнения функции MessageBox().

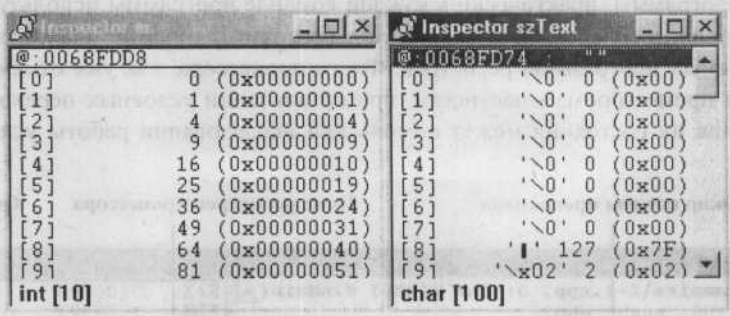


Рис. 2.13. Произошло присвоение значений элементам массива m

В некоторых случаях при отладке или исследовании программы приходится переходить на более низкий уровень отладки, выводя на экран "окно процессора". Выполним такое исследование, введя для этого в программу новую переменную с именем, например, glob, объявив ее *до* функции WinMain(). Переменные, объявленные до функции WinMain(), называются глобальными, и их основное качество состоит в том, что к ним можно обращаться из любых функций, включенных в программу. Глобальными переменными удобно пользоваться для того, чтобы передавать данные из одной функции в другую. Переменные, объявленные внутри какой-либо функции (в нашем случае – внутри единственной функции WinMain()) называются локальными, и ими можно пользоваться только внутри той функции, в которой они объявлены. Вне этой функции они не только невидимы, а по-просту не существуют. Локальные переменные удобны тем, что

не загромождают программу лишними данными; они автоматически создаются при входе в функцию и так же автоматически удаляются при выходе из нее.

Более содержательное рассмотрение вопроса о локальных и глобальных переменных мы отложим до соответствующего места в книге, а сейчас только посмотрим, как и где выделяется под них память.

Видоизмененное начало примера 2-1 будет выглядить таким образом:

```
/*Главная функция WinMain*/
int glob=0x12345678;//Глобальная переменная
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){//(2) Заголовок функции
  int m[10];//(3) Объявление массива из 10 целых чисел
... Продолжение программы
```

Перекомпилируем программу и, доведя ее выполнение, как и раньше, до строки (5) с предложением цикла, выведем на экран с помощью команды View➢CPU "окно процессора" (рис. 2.14). В этом окне расположены 5 внутренних окон, в которые выводятся команды программы, а также содержимое регистров процессора, флагов, стека и памяти. В окно команд отладчик выводит предложения исходного текста программы (на языке C++), а также команды процессора (отображаемые на языке ассемблера), которые образуются в результате компиляции этих предложений и составляют, в сущности, выполнимую программу. Перед каждой командой процессора указан ее адрес. В окне памяти можно изучать содержимое любых участков памяти, выделенных программе; в окне стека наблюдать, как выделяется и освобождается память под локальные переменные (которые всегда размещаются в стеке). Окно регистров может понадобиться при детальном изучении хода выполнения программы. Действительно, как можно видеть из ассемблерного текста программы, практически в каждой команде программы используется тот или иной регистр процессора, и, прослеживая ход программы, полезно наблюдать содержимое используемых программой регистров. Флаги процессора, как уже отмечалось ранее, используются процессором, в частности, при организации условных переходов, и динамика изменения их состояния может помочь при исследовании работы условных предложений if.

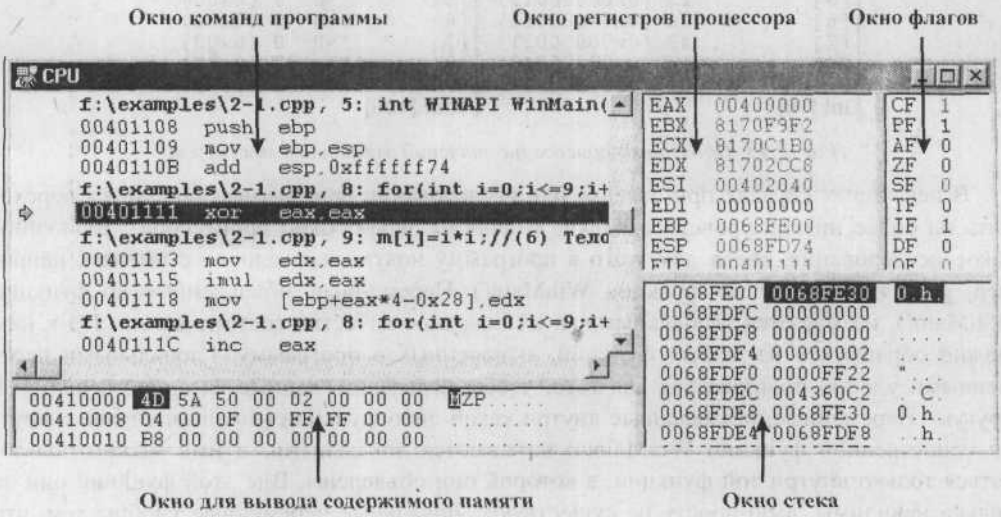


Рис. 2.14. Окно процессора с внутренними окнами

С помощью команд Debug➢Inspect определим адрес глобальной переменной glob и локальной переменной (массива) m. Получим для переменной glob адрес 0x00402088, а для переменной m – 0x0068FDD8. В окне памяти найдем глобальную переменную glob. Для этого, щелкнув мышью по этому окну, введем команду Alt+F10, которая в любом внутреннем окне процессора позволяет открыть меню, специфичное для данного окна. Тот же результат можно получить, щелкнув по окну правой клавишей мыши. Внутреннее меню окна памяти показано на рис. 2.15.

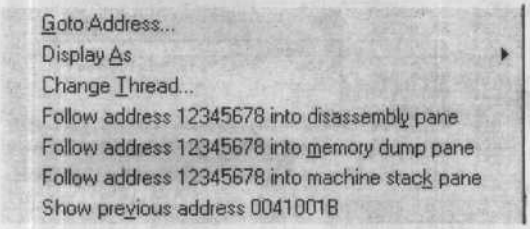


Рис. 2.15. Внутреннее меню окна памяти

Для перехода по определенному адресу следует выбрать пункт Goto Address... и в появившемся окне (рис. 2.16) ввести требуемый адрес. При этом надо иметь в виду, что, хотя отладчик в большинстве случаев *при выводе* числовых данных использует шестнадцатеричную систему счисления (и полученный нами адрес 00402088 является шестнадцатеричным), *вводимое* число отладчик воспринимает как десятичное. Поэтому в окне задания адреса следует в явной форме указать систему счисления по правилам языка C++: 0x402088.

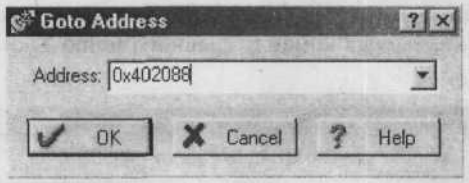


Рис. 2.16. Окно для задания адреса

Аналогично перейдем в окне стека по адресу 0x0068FDD8, с которого начинается массив m. На рис. 2.17 показан кадр отладчика стояние окна процессора после выполненных настроек; на рисунке видны также окна, содержащие значения исследуемых переменных.

Посмотрим, как компилятор выделяет память под различные элементы программы. Главная функция WinMain() начинается с адреса 0x401108. Поскольку наше приложение является 32-разрядным, этот виртуальный адрес совпадает с линейным и обозначает просто номер байта линейного пространства адресов (общим размером 4 Гбайт), с которого начинаются коды функции WinMain(). Кстати, этот адрес соответствует началу пятого мегабайта линейного адресного пространства. Глобальная переменная glob размещается ниже, по адресу 0x402088 и занимает там четыре байта. Обратите внимание на то, как байты переменной glob хранятся в памяти. Начиная с адреса 0x402088 располагаются байты 78 56 34 12, т. е. переменная в памяти выглядит как бы перевернутой (ср. с рис. 1.5).

Локальная переменная m, представляющая собой массив из десяти целых чисел, размещается, как и все локальные переменные, в стеке – области памяти для временного хранения данных, которая адресуется с помощью регистра ESP. Действительно, в регист-

ре ESP на момент остановки нашей программы содержится адрес 0x68FD74, что соответствует адресу одной из ячеек стека. Сравнив содержимое массива m (пока еще не инициализированного, т. е. содержащего "мусор") в окне инспектора с содержимым стека, можно обнаружить в стеке весь наш массив. Обратите внимание на то, что адреса в окне стека располагаются не так, как окнах команд или памяти – они возрастают снизу вверх, т. е. память стека изображается как бы перевернутой.

Рис. 2.17. Поиск глобальных и локальных переменных в окнах процессора

Вторая возможность отладки программ с помощью средств пакета Borland C++ заключается в использовании полноэкранного отладчика TD32.EXE, который можно вызвать непосредственно из среды разработки, выбрав команду главного меню Tool➢Turbo Debugger (рис. 2.18).

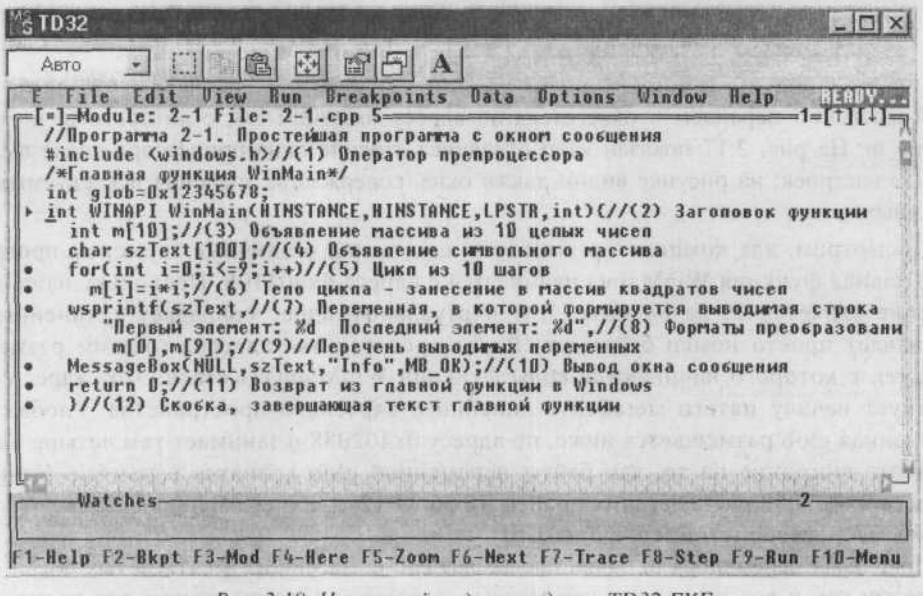


Рис. 2.18. Начальный кадр отладчика TD32.EXE

Дальнейшие процедуры отладки с помощью полноэкранного отладчика в принципе не отличаются от описанных выше. Так, с помощью команды Data главного меню отладчика можно выводить содержимое переменных программы, команда View➢CPU отображает на экране окно процессора, нажатие клавиши F4 позволяет выполнить программу до курсора и т. д. Фактически при работе в среде IDE нет необходимости использовать отладчик TD32.

Однако запуская этот отладчик непосредственно с Рабочего стола Windows, можно отлаживать или изучать программы в автономном от среды IDE режиме, в том числе и программы, для которых отсутствуют исходные тексты.

Существенным элементом отладки программ является получение справки по тем или иным средствам Windows. Как уже отмечалось, в среду IDE встроен интерактивный справочник, который можно активизировать выбором команды меню Help➢Windows API, а также просто нажатием клавиши F1, если курсор клавиатуры установлен на интересующем нас термине.

Интегрированная среда разработки Microsoft Visual C++

Общие принципы разработки и отладки приложения одинаковы во всех средах разработки, однако разные пакеты программирования могут заметно различаться в деталях. Не является здесь исключением и среда Visual C++, в которой процедура подготовки программы (да и получаемый результат) выглядит совсем не так, как в IDE Borland C++. При запуске Visual C++ (файл MSDEV.EXE) на Рабочий стол выводится начальный кадр (рис. 2.19), с помощью которого можно определить дальнейший ход разработки.

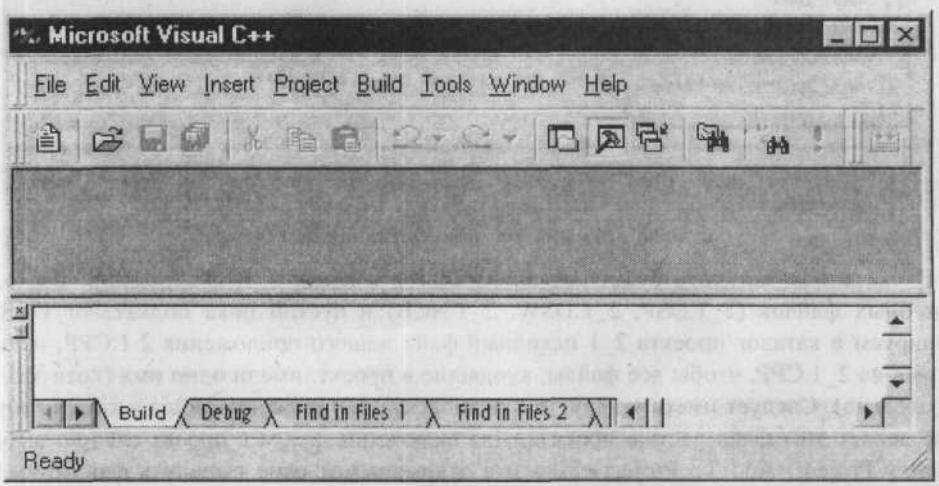


Рис. 2.19. Начальный кадр IDE Visual C++ 6.0

Введя команду File➢New➢Projects, мы получим возможность выбрать тип создаваемого объекта и указать его имя (рис. 2.25).

Мы хотим создать приложение Windows (тип Win32 Application); в качестве имени проекта придется использовать, например, имя 2_1, так как имя 2-1 в этой среде оказывается недопустимым. В следующем кадре (рис. 2.21) можно задать пустой пока проект для самостоятельной подготовки исходных текстов приложения (An empty project), готовый шаблон для простого приложения (A simple Win32 application), исходный текст которого включает лишь пустую функцию WinMain(), или полностью сформированное приложение, выводящее в главное окно текст "Hello, World!" (A typical "Hello World!" application). Выберем первую возможность.

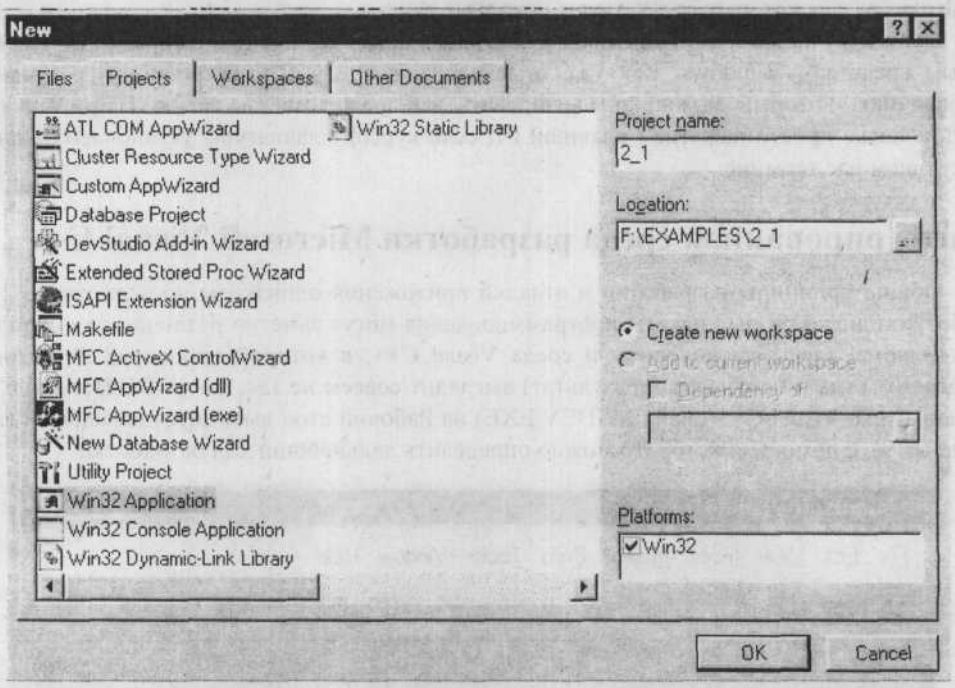


Рис. 2.20. Кадр выбора типа создаваемого объекта

В указанном каталоге будет создан каталог с именем 2_1, содержащий несколько служебных файлов (2_1.DSP, 2_1.DSW, 2_1.NCB) и пустой пока подкаталог DEBUG. Скопируем в каталог проекта 2_1 исходный файл нашего приложения 2-1.CPP, изменив его имя на 2_1.CPP, чтобы все файлы, входящие в проект, имели одно имя (хотя это и не обязательно). Следует иметь в виду, что физическое копирование файла в каталог проекта не делает этот файл частью проекта. Для включения файла в проект следует выбрать команду Project➢Add To Project➢Files и в открывшемся окне выделить файл с именем 2_1.CPP. На экран будет выведен рабочий кадр Visual C++, состоящий из трех окон – состава проекта, исходных текстов и сообщений.

В левом окне кадра (окне состава проекта) перейдем на закладку FileView, далее, последовательно щелкая по значкам ⊞, откроем список имеющихся файлов и, наконец, щелкнув дважды по имени файла 2_1.CPP, выведем в правое окно исходный текст нашей программы (рис. 2.22).

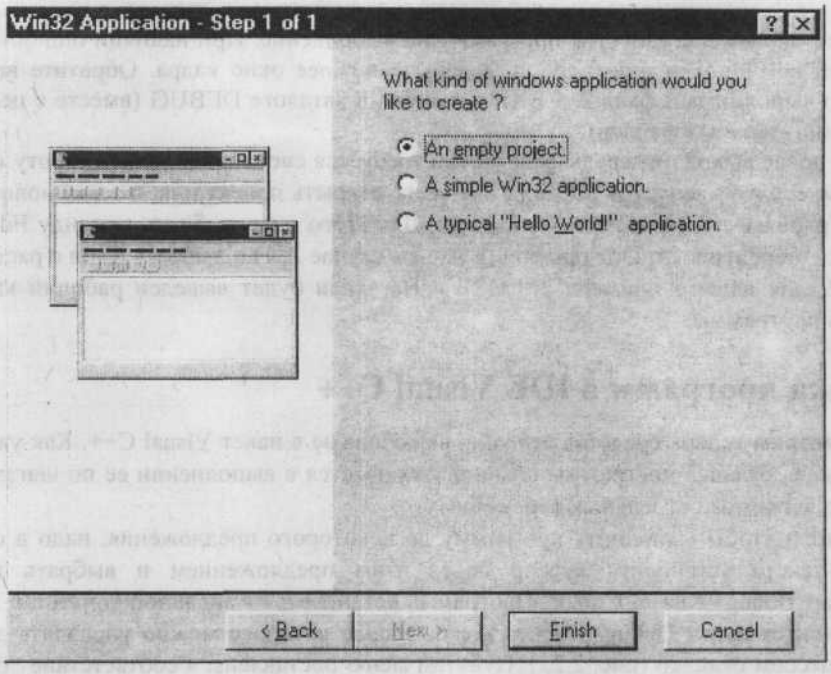


Рис. 2.21. Выбор типа создаваемого приложения

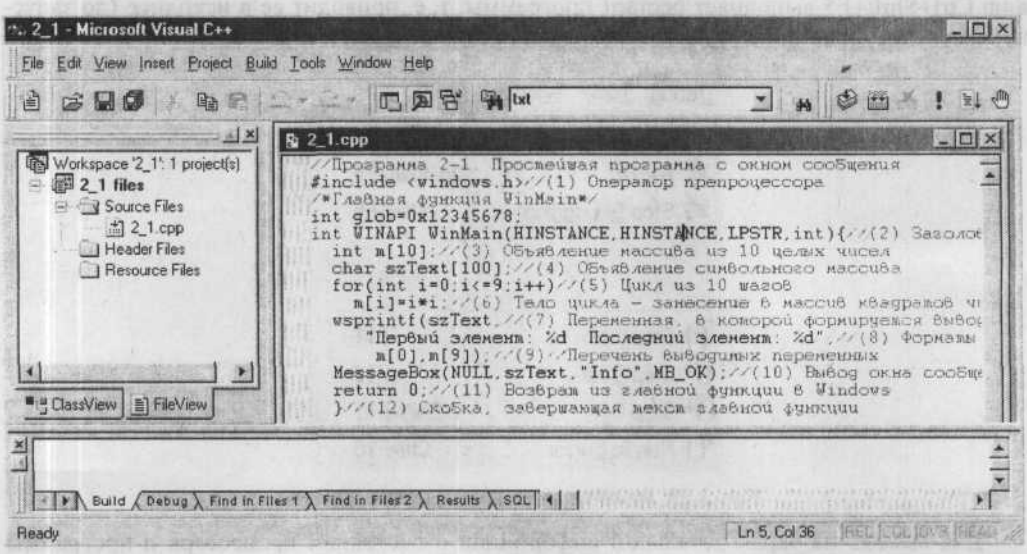


Рис. 2.22. Рабочий кадр среды Visual C++ с открытым исходным текстом программы

Для получения выполнимого файла приложения следует выбрать (с помощью главного меню) команду Build➢Build 2_1.exe или просто нажать клавишу F7. После этого

щелчок по кнопке ! запустит программу на выполнение. При наличии ошибок компиляции или компоновки они будут выведены в нижнее окно кадра. Обратите внимание на то, что выполнимый файл 2_1.EXE создается в каталоге DEBUG (вместе с целым рядом вспомогательных файлов).

Если после выхода из среды разработки требуется снова продолжить работу над приложением, следует, запустив IDE, прежде всего открыть проект или, в терминологии Visual C++, рабочее пространство (workspace). Для этого надо выбрать команду File➢Open Workspace, перейти в каталог проекта (в нашем случае 2_1) и выбрать файл с расширением .DSW (для нашего примера 2_1.DSW). На экран будет выведен рабочий кадр IDE с текстом программы.

Отладка программ в IDE Visual C++

Рассмотрим теперь средства отладки, включенные в пакет Visual C++. Как уже отмечалось выше, отладка программы обычно заключается в выполнении ее по шагам с контролем содержимого отдельных переменных.

Для того чтобы выполнить программу до некоторого предложения, надо в окне исходного текста установить курсор перед этим предложением и выбрать команду Build➢Start Debug➢Run to Cursor. Программа остановится в заданной точке, а в главном меню появится пункт Debug (отладка), с помощью которого можно управлять дальнейшим процессом отладки (рис. 2.23). Пунктам меню поставлены в соответствие "горячие" клавиши. Так, клавиша F5 заставляет программу выполниться до конца, сочетание клавиш Ctrl+Shift+F5 выполняет рестарт программы, т. е. приводит ее в исходное (до запуска) состояние, сочетание Shift+F5 прекращает сеанс отладки и т. д.

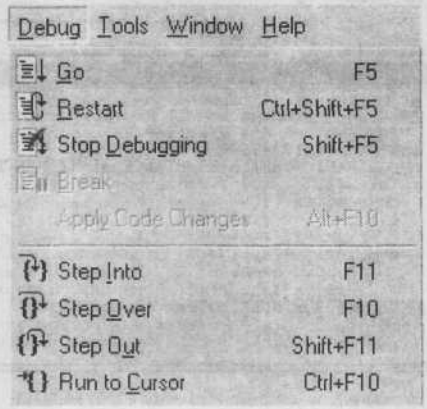


Рис. 2.23. Меню управления процессом отладки

Для вывода на экран детальной информации о состоянии процессора и программы следует выбрать команду View➢Debug Windows. Появившееся меню (рис. 2.29) позволяет управлять содержимым окон кадра IDE. Так, пункт Memory создает окно памяти, пункт Variables – окно переменных программы (именно это окно изображено в нижней части рис. 2.29), пункт Registers – окно регистров.

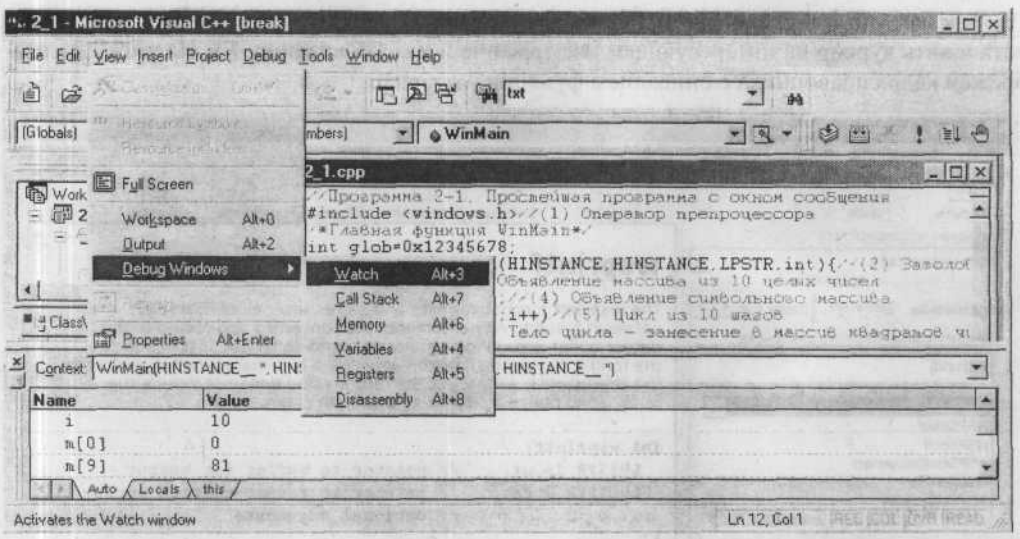


Рис. 2.24. Меню выбора окон для отладки

С помощью пункта Disassembly можно получить окно команд программы, в которое выводятся как предложения исходного текста, так и полученные в результате компиляции команды процессора. На рис. 2.25 такое окно изображено на момент, когда выполнение программы дошло до вызова функции wsprintf().

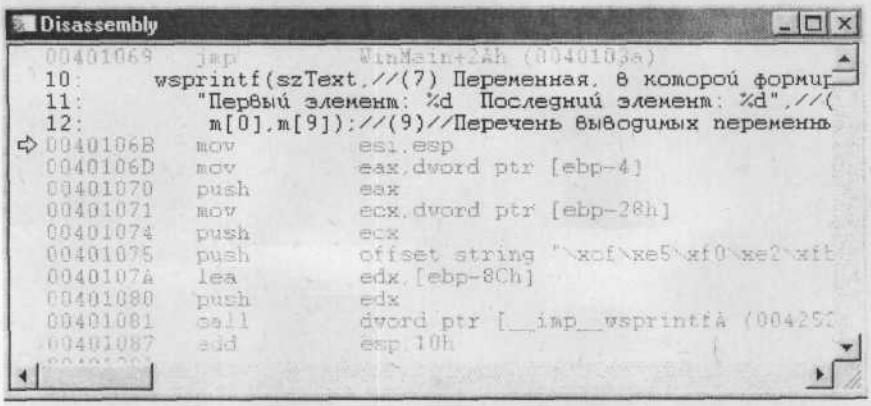


Рис. 2.25. Окно команд программы

В состав комплексного пакета Visual Studio, включающего, в частности, интерактивную среду разработки IDE Visual C++, входит объемный интерактивный справочник, занимающий несколько компакт-дисков. Этот справочник можно вызывать непосредственно с компакт-диска, а можно (при наличии достаточного дискового пространства) скопировать на жесткий диск (точнее, установить на жестком диске), что существенно повысит удобство работы с ним.

Так же, как и в среде Borland C++, для получения необходимой справки достаточно установить курсор на интересующем нас термине и нажать клавишу F1. На рис. 2.26 изображен кадр справочника с описанием функции wsprintf().

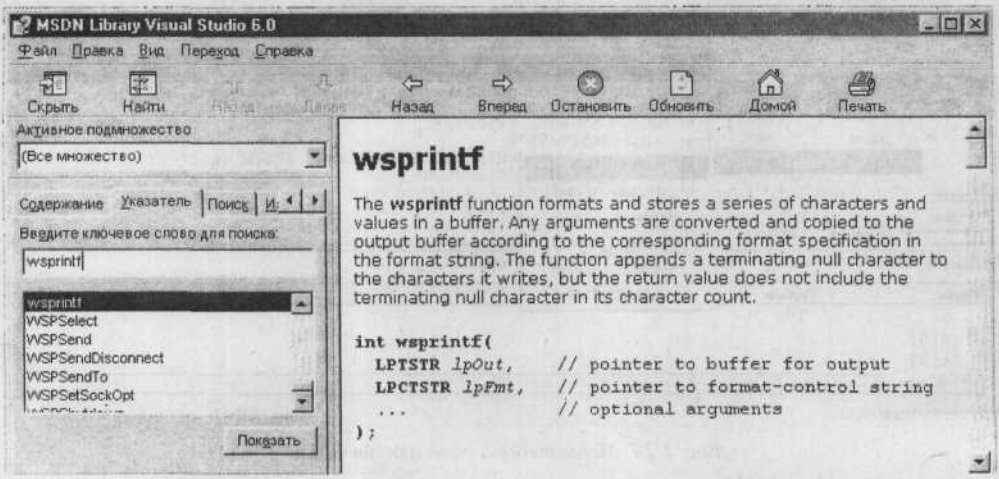


Рис. 2.26. Окно интерактивного отладчика MSDN Library Visual Studio 6.0

Глава 3
Основы программирования на языке C++

Перед тем как приступить к изучению специфических вопросов программирования в системе Windows, нам следует познакомиться с некоторыми базовыми понятиями языка C++. Настоящую главу не следует рассматривать как систематический учебник по языку; в ней будут выборочно описаны лишь те средства языка, без которых не обходится ни одна Windows-программа и которые, в то же время, часто вызывают трудности у начинающих программистов.

Работа над программными примерами

Рассмотрение конструкций языка в нашей книге сопровождается фрагментарными примерами, не являющимися законченными программами и часто состоящими всего из нескольких строк. Читатель, желающий закрепить этот материал практической работой на компьютере, может воспользоваться программой 2-1 из предыдущей главы в качестве заготовки, помещая в нее те или иные из приводимых ниже примеров. При составлении очередного примера в программу следует включить требуемые в примере переменные, а также изучаемые предложения языка. Для удобства отладки переменные лучше объявлять до функции WinMain(), т. е. делать их глобальными. В тех случаях, когда программа компилируется без ошибок, но работает неправильно, следует воспользоваться встроенным отладчиком и, выполняя программу по шагам и анализируя результаты каждого шага, попытаться найти место, где частичный результат не соответствует ожидаемому. Отладчиком также можно пользоваться для вывода содержимого образуемых в программе массивов (см., например, рис. 2.12 и 2.13). В тех же случаях, когда результаты работы программы выражаются в одном или нескольких данных, их удобно вывести в окно сообщения с помощью пары функций wsprintf() – MessageBox(),. первая из которых позволяет преобразовать числовые значения в символьную строку, а вторая создает и выводит на экран окно сообщения с сформированным текстом. Рассмотрим кратко особенности использования этих функций.

Функция MessageBox()

Общий формат вызова функции MessageBox() при использовании ее в примерах настоящей главы выглядит следующим образом:

```
MessageBox(NULL, адрес_выводимой_строки, заголовок_окна, MB_OK);
```

Заголовок окна может быть каким угодно (например, фразы “Info” или “Результаты программы”. Обратите внимание на необходимость помещения этих фраз в кавычки “”.

Что касается самой выводимой строки, то здесь могут встретиться разные варианты. В некоторых примерах желательно вывести в окно сообщения некоторый фиксированный текст. В таком случае этот текст можно непосредственно поместить в качестве второго аргумента функции MessageBox(), заключив его в кавычки:

```
MessageBox(NULL, "Все нормально!", "Результат", MB_OK);
```

В других примерах назначением программы является то или иное преобразование символьной (текстовой) строки. Например, программа может анализировать фразу и заменять все строчные буквы на прописные или наоборот. В этом случае в окно сообщения необходимо вывести преобразованную символьную строку, для чего в качестве второго аргумента функции MessageBox() используется имя этой строки:

```
char txt[]="Myfile.aaa"//Объявляется символьная строка с именем txt
/*Алгоритм преобразования строчных букв в прописные. В результате в той же
  строке txt будет такой текст: "MYFILE.AAA" */
MessageBox(NULL, txt, "Результат", MB_OK);//Вывод преобразованной строки
```

Наконец, в большинстве случаев вывести надо не текст, а одно или несколько чисел, предварительно преобразовав их в символьную форму. Для этого необходимо воспользоваться функцией wsprintf().

Функция wsprintf

Функция Windows wsprintf() преобразует заданный список переменных в символьную форму в соответствии с указанным списком форматов преобразования и помещает образованный ею текст в предусмотренный в программе символьный массив-приемник. Вызов функции wsprintf() в общем случае выглядит следующим образом:

```
wsprintf(адрес_приемника,"строка_со_списком_форматов", список_переменных);
```

Строка со списком форматов включает в себя как произвольные слова или фразы, которые мы хотим вывести на экран, так и список форматов, указывающих, в какой форме должны быть выведены соответствующие переменные (например, в форме десятичного или шестнадцатеричного числа), причем количество форматов должно точно соответствовать количеству указанных переменных.

В строку со списком форматов могут быть включены любые фрагменты текста, например, имена выводимых переменных вместе с поясняющими словами, либо просто пробелы для разделения выводимых данных. Обозначение \n, включенное в эту строку, реализует переход на следующую строку окна сообщения, обеспечивая тем самым вывод значений переменных столбиком. Другая возможность форматирования текста – включение в строку форматов обозначения \n, которое смещает следующий за ним текст к позиции табуляции.

Формат преобразования переменной начинается со знака процента (%), за которым следует один или несколько символов. Наиболее распространенные обозначения форматов приведены в табл. 3.1.

Таблица 3.1. Наиболее распространенные форматы преобразования, используемые в функции wsprintf()

Обозначение формата	*Назначение*
%x, %X	Значение переменной выводится в форме шестнадцатеричного числа строчными (%x) или прописными (%X) символами
%#x, %#X	То же, но перед числом выводится обозначение 0x, символизирующее шестнадцатеричное число (для наглядности)
%d, %i	Значение переменной выводится в форме десятичного числа со знаком
%u	Значение переменной выводится в форме десятичного числа без знака

Обозначение формата	*Назначение*
%0*кол*d	То же, но выводятся *кол* десятичных разрядов, выровненных по правому краю. Недостающие разряды слева заполняются нулями
%c	Выводится один символ, соответствующий коду ASCII переменной
%s	Выводится символьная строка; переменная должна представлять собой имя этой строки или саму строку

Ниже приведено несколько примеров использования функции wsprintf(); в комментариях показано содержимое формируемой строки:

```
char szText[100];//Массив-приемник для формирования выводимой строки
int value=0x12ab;
int number=100;
char sym='U';
int m[3]={10,20,30};
wsprintf(szText,"Значение value=%X",value);//Вывод: "Значение value=12AB"
wsprintf(szText,"value=%#x...number=%d",value,number);//Вывод:
                                                "value=0x12ab...number=100"
wsprintf(szText,"<<<%c>>>",sym);//Вывод: "<<<U>>>"
wsprintf(szText,"%d\n%d\n%d",m[0],m[1],m[2]);//Вывод столбиком: "10
                                                                 20
                                                                 30"
```

Функция sprintf

При программировании научных задач может потребоваться вывести в окно сообщения значения чисел с плавающей точкой. Функция wsprintf() не предоставляет такой возможности. Однако в приложении Windows можно воспользоваться функцией sprintf(), предназначенной, строго говоря, для приложений DOS. Формат вызова этой функции в точности такой же, как и у функции wsprintf(), однако в списке форматов преобразования, кроме описанных в табл. 3.1, имеются обозначения, служащие для вывода чисел с плавающей точкой (табл. 3.2).

Таблица 3.2. Дополнительные форматы преобразования, используемые в функции sprintf()

Обозначение формата	*Назначение*
%f	Значение переменной выводится в форме числа с десятичной точкой
%e, %E	Значение переменной выводится в форме числа с десятичной точкой и множителем, обозначающим десять в указанной степени. Множитель обозначается символом e (для формата %e) или E (для формата %E)
%.*n*f, %.*n*e, %.*n*E	Значение переменной выводится в одной из указанных выше форм, при этом число знаков после десятичной точки ограничивается значением *n*

Примеры использования функции sprintf():

```
char szText[100];//Массив-приемник для формирования выводимой строки
```

```
float v=1.2459;
sprintf(szText,"Значение v=%f",v);//Вывод: "Значение v=1.245900"
sprintf(szText,"Значение v=%.4f",v);//Вывод: "Значение v=1.2459"
sprintf(szText,"Значение v=%.2f",v);//Вывод: "Значение v=1.24"
sprintf(szText,"Значение v=%e",v);//Вывод: "Значение v=1.245900e+00"
sprintf(szText,"Значение v=%.2e",v);//Вывод: "Значение v=1.24e+00"
```

Объявление и использование данных

В языке C++ действует жесткое правило: все переменные перед тем, как их использовать, должны быть объявлены. Объявление переменной состоит из ее типа, произвольного имени и завершающего символа ";". При объявлении нескольких переменных одного типа их можно перечислить в одном предложении объявления, разделяя запятыми. Например, предложение

```
int Counter;
```

описывает целочисленную переменную (типа int) с именем Counter, предложение

```
char symbol;
```

вводит переменную symbol, занимающую 1 байт и предназначенную для хранения кода того или иного символа (буквы, цифры, знака препинания и др.), а предложение

```
float x,y,z;
```

объявляет 3 переменные x, y и z с плавающей точкой.

Чем объясняется необходимость объявления переменных? Объявление позволяет компилятору выделить память, необходимую для размещения данной переменной, и заодно присвоить этой памяти имя, по которому в дальнейшем к этому участку памяти можно обращаться. Переменные разных типов имеют различную длину, например тип char занимает всего один байт, а тип int – целых четыре. Соответственно, массив из, скажем, 1024 данных типа char потребует для своего размещения точно 1 Кбайт, а такой же по числу элементов массив данных типа int – в четыре раза больше.

Необходимо иметь в виду, что компилятор языка C++ различает строчные и прописные буквы. Так, использование в тексте программы имен counter или COUNTER будет обозначать совсем другие переменные, отличные от переменной Counter. Точно так же недопустимо вместо int использовать INT, а вместо char – CHAR. С другой стороны, во вспомогательных файлах Windows, обслуживающих 32-разрядные программы, введены дополнительные типы INT и CHAR, вполне эквивалентные стандартным типам C++ int и char. Однако в 16-разрядных программах ими пользоваться нельзя.

Многие программисты предпочитают давать своим переменным относительно длинные, но зато исчерпывающе информативные имена, состоящие из нескольких слов. В этом случае разумно каждое слово, входящее в имя переменной, начинать с прописной буквы: GreenBrush, OldValue, MainWindowTitle и т. д. По тому же принципу строятся имена большинства функций Windows: CreateWindow(), GetMessage(), SendDlgItemMessage() и др.

Следует заметить, что в системе Windows (и в Win16, и в Win32) определено большое количество специфических типов данных. Во многих случаях они просто являются другими обозначениями (часто более короткими и удобными) стандартных типов C++. Так, например, указатель на целочисленную переменную, обозначаемый в языке C++ как int *, в Win32 имеет имя PINT (от Pointer to Int); указатель на строку символов в языке C++ выглядит как char * или char[ ], а в Win32 – PSTR (от Pointer to String) или PSZ

(от Pointer to String with Zero). (Назначение указателей будет рассмотрено в последующих разделах этой главы.)

В других случаях типы Win32 характеризуют специфические для Windows объекты, и им нет прямых аналогов в C++. Например, для переменных, задающих цвет некоторого объекта, используется тип COLOREF (от Color Reference); для переменных, в которые поступают результаты обработки сообщений Windows, предусмотрен тип LRESULT (от Long Result) и т. д. Эти типы данных используются главным образом при вызове функций Windows в качестве входных или выходных параметров; их выбор не вызывает особых затруднений, так как в интерактивных справочниках по функциям Windows содержится исчерпывающая информация о том, с какими типами данных работает та или иная функция.

Необходимо иметь в виду, что один и тот же тип данных в разных операционных средах может иметь различные размеры и возможности. Так, булева переменная типа bool в языке C++ занимает 1 байт и может принимать только значения true (истина) или false (ложь), которые фактически записываются в память как 1 или 0. Однако булева переменная, объявленная с помощью типа Windows BOOL (прописными буквами) занимает четыре байта и хотя ее, так сказать, официальные значения должны быть только TRUE или FALSE, фактически в нее можно записать любое целое число, которое будет храниться в памяти и может быть использовано в нужном месте. Такое "неправомерное" использование булевых переменных в некоторых случаях оказывается принципиально важным.

На возможности переменных влияет также тип приложения. Так, в Win16 (и в C++ для DOS) тип int обозначает переменную размером в одно слово, т. е. 16 бит; в Win32 тот же тип int (или INT) относится к переменным размером в двойное слово, т. е. 32 бита.

Нас будут интересовать только 32-разрядные приложения Windows. Поэтому мы будем просто использовать в программах типы переменных, допустимые для среды Win32, не вдаваясь в историю вопроса и в рассуждения о возможной несовместимости с 16-разрядными приложениями. Во многих случаях мы не будем также прибегать к специальным типам Windows, когда вместо них можно использовать привычные стандартные эквиваленты языка C++ вроде int или char*.

Данные, с которыми работают программы, можно подразделять по разным принципам. Например, данные могут быть целочисленными или с плавающей точкой; со знаком или без знака; с малым или большим диапазоном значений. Более принципиальным является подразделение данных на скалярные, массивные и структурные.

Скалярные данные

К скалярным относятся такие переменные, каждая из которых описывает одно данное, в отличие от массивов и структур, которые представляют собой коллекции данных и иногда называются агрегатными переменными. Если переменная по смыслу должна иметь начальное значение, его можно присвоить на этапе объявления этой переменной:

```
int Index;//Целочисленная переменная без начального значения
int Dimension=1000;//Целочисленная переменная с начальным значением
char FirstLetter='A';//Символьная переменная с начальным значением
```

В отличие от языка ассемблера, где данные обычно описываются в отдельном сегменте данных, а сегмент команд содержит только команды процессора, выполняющие те или иные операции над данными, в языке C++ объявлять новые переменные можно где угодно по ходу программы. Объявление всех переменных в начале программы придает программе более структурированный характер, однако "приближение" объявления пере-

менной к тому месту в программе, где она действительно нужна, упрощает отладку программы.

Как уже упоминалось, в Windows предусмотрено много специализированных типов данных, относящихся к определенным объектам. Почти любой объект Windows (кисть, шрифт, окно, файл и др.) имеет характеризующее его число, называемое дескриптором (handle). Иногда дескриптор представляет собой просто порядковый номер объекта. В других случаях дескриптор является адресом области данных, содержащей подробные характеристики объекта. Так или иначе при любых действиях с конкретным объектом мы должны указывать его дескриптор, чтобы система знала, какой именно объект мы имеем в виду. Дескрипторы каждого вида объектов принадлежат к определенному типу данных. Так, тип HWND описывает дескрипторы окон, тип HBRUSH – дескрипторы кистей, тип HFILE – дескрипторы файлов и т. д. Имеется и обобщенный тип дескриптора объекта HANDLE. Со многими видами дескрипторов мы столкнемся при рассмотрении примеров приложений Windows.

Помимо специализированных типов данных, в Win32 имеется целый ряд типов общего назначения для описания любых данных пользователя. Некоторые из них приведены в табл. 3.3.

Таблица 3.3. Наиболее распространенные типы данных общего назначения в Win32

Тип данного	*Аналоги в C++*	*Размер, бит*	*Описание*	*Диапазон чисел*
BOOL	bool	32	Булева переменная	TRUE (1), FALSE (0)
BOOLEAN	bool	32	Булева переменная	TRUE (1), FALSE (0)
BYTE	unsigned char	8	Байт без знака. Может обозначать число или код символа	0…255
CCHAR	char	8	Символ Windows	-128…+127
CHAR	char	8	Символ Windows	-128…+127
CONST	const	–	Константа	–
DWORD	unsigned long	32	Двойное слово без знака	0… 42944967295
DWORDLONG	double	64	Число с плавающей точкой со знаком	–1.7E-308… 1.7E+308
FLOAT	float	32	Число с плавающей точкой со знаком	–3.4E-38… 3.4E+38
INT	int, long	32	Целое число со знаком	–2147483648… +2147483647
LONG	long, int	32	Целое число со знаком	–2147483648… +2147483647
LONGLONG	double	64	Число с плавающей точкой со знаком	–1.7E-308… 1.7E+308
SHORT	short	16	Короткое целое число со знаком	–32768… +32767
TBYTE	unsigned char	8	Байт без знака. Может обозначать число или код символа	0…255

Тип данного	*Аналоги в C++*	*Размер, бит*	*Описание*	*Диапазон чисел*
TCHAR	char	8	Символ Windows или Unicode	–128...+127
UCHAR	unsigned char	8	Символ Windows без знака	0...255
UINT	unsigned int	32	Целое число без знака	0... 4294967295
ULONG	unsigned long	32	Целое число без знака	0... 4294967295
USHORT	unsigned short	16	Короткое целое число без знака	0...65535
VOID	void	–	Любой тип	–
WCHAR	wchar_t	16	Символ Unicode	0...65535
WORD	–	16	Короткое целое число без знака	0...65535

Ключевое слово CONST (или const), включенное в таблицу, характеризует особый тип константных переменных – переменных, значения которых можно только читать, но нельзя изменять. Такой переменной должно быть присвоено начальное (и единственное) значение при ее объявлении, попытка изменить ее значение по ходу программы приведет к ошибке компиляции. Символическое обозначение CONST представляет собой не самостоятельный тип, а модификатор, так как используется совместно с другими типами данных:

```
const int i=125;//Целочисленная константа i, равная 125
const char star='*';//Символьная константа star, равная коду символа '*'
```

Максимально широкое использование константных переменных приветствуется в C++, поскольку повышает надежность разрабатываемой программы.

Особо следует отметить тип VOID (или void). Ключевое слово void (вакуум, пустота) используется в языке C++ в двух ситуациях. Если это слово указывается в качестве типа данного, возвращаемого некоторой функцией, это означает, что данная функция не возвращает никакого результата. В таком контексте слово void означает "отсутствие". Однако слово void может использоваться также и в тех случаях, когда тип данного пока неизвестен, т. е. тип void может заменять собой любой тип данного. Более подробно об этом будет рассказано в разделе, посвященном указателям.

Рассмотрим немного подробнее некоторые распространенные типы скалярных данных.

Числа со знаком и без знака

В 16-битовое машинное слово или 16-разрядный регистр можно записать 64 К различных чисел в диапазоне от 0x0000 до 0xFFFF, или от 00000 до 65535. Это справедливо в том случае, если все возможные числа рассматриваются как положительные (без знака). Если, однако, мы хотим в какой-то программе работать как с положительными, так и с отрицательными числами, т. е. с числами со знаком, нам придется часть чисел из их полного диапазона (наиболее естественно – половину) считать отрицательными. В вычислительной технике принято отрицательными считать все числа, у которых установлен старший бит, т. е. числа в диапазоне 0x8000...0xFFFF. Положительными же считаются числа со сброшенным старшим битом, т. е. числа в диапазоне 0x0000...0x7FFF. При этом отрицательные числа записываются в дополнительном коде, который образуется из прямого путем замены всех двоичных нулей единицами и наоборот (обратный код) и прибавления к полученному числу двоичной единицы (рис. 3.1).

	Для байта	Для слова
Прямой код числа 5:	0000 0101	0000 0000 0000 0101
Обратный код числа 5:	1111 1010	1111 1111 1111 1010
	+1	+1
Дополнительный код числа 5:	1111 1011	1111 1111 1111 1011

	Для двойного слова
Прямой код числа 5:	0000 0000 0000 0000 0000 0000 0000 0101
Обратный код числа 5:	1111 1111 1111 1111 1111 1111 1111 1010
	+1
Дополнительный код числа 5:	1111 1111 1111 1111 1111 1111 1111 1011

Рис. 3.1. Образование отрицательного числа

Следует подчеркнуть, что знак числа условен. Одно и то же число 0xFFFB, изображенное в правой верхней части рис. 3.1, можно в одном контексте рассматривать, как положительное (+65531), а в другом – как отрицательное (–5). Таким образом, знак числа является характеристикой не самого числа, а способа его обработки. Как видно из рис. 3.1, все сказанное применимо как к совсем коротким числам, занимающим 1 байт, так и к двухсловным числам, имеющим размер 32 бита (4 байта).

Приведем программный пример исследования свойств чисел со знаком и без знака.

```
/*Программа 3-1. Знаковые и беззнаковые переменные?/
#include <windows.h>
unsigned char C1=0xFF;//Объявим байт без знака
char C2=0xFF;//Объявим байт со знаком с тем же значением
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
  if(C1>10)
    MessageBox(NULL,"C1 > 10","Info",MB_OK);
  else
    MessageBox(NULL,"C1 < 10","Info",MB_OK);
  if(C2>10)
    MessageBox(NULL,"C2 > 10","Info",MB_OK);
  else
    MessageBox(NULL,"C2 < 10","Info",MB_OK);
  return 0;
  }
```

В приведенном примере объявляются две глобальные переменные, одна без знака, а другая со знаком, и обеим присваиваются *одинаковые* значения 0xFF. Значение каждой переменной сравнивается с произвольным числом 10. Если анализируемая переменная больше 10, выводится сообщение "C*i* > 10"; в противном случае выводится сообщение "C*i* < 10" (под C*i* здесь понимается как C1, так и C2). Вывод программы (рис. 3.2) подтверждает, что C1 рассматривается как положительное число равное $0xFF = 255_{10}$; оно, естественно, больше 10. Что же касается числа со знаком C2, то поскольку его старший бит установлен ($0xFF = 11111111_2$), оно считается отрицательным и равным конкретно –1. Очевидно, что –1 < 10, о чем и сообщается во втором окне вывода программы.

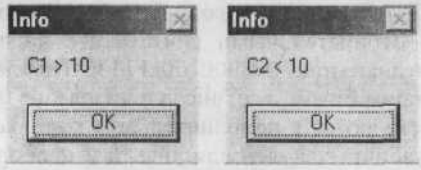


Рис. 3.2. Анализ чисел без знака и со знаком

Символьные переменные

Переменные типа char (от character, символ) часто называют символьными, хотя, по существу, это просто целые числа со знаком, каждое из которых занимает 1 байт. Однако ряд функций C++ рассматривают данное, хранящееся в переменной типа char, не как число, а как код ASCII символа в соответствии с загруженной на данном компьютере кодовой таблицей (рис. 3.3). В частности, так действуют функции, вводящие данные с клавиатуры и выводящие их на экран. При вводе с клавиатуры данного в некоторую символьную (байтовую) переменную в нее поступает код ASCII введенного символа; при выводе на экран символьной переменной, хранящей какой-то код, на экране появляется изображение символа, соответствующего этому коду.

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
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	Ъ	Ѓ	‚	ѓ	„	…	†	‡	□	‰	Љ	‹	Њ	Ќ	Ћ	Џ
90	ђ	‘	’	“	”	•	–	—	□	™	љ	›	њ	ќ	ћ	џ
A0		Ў	ў	Ј	¤	Ґ	¦	§	Ё	©	Є	«	¬	-	®	Ї
B0	°	±	І	і	ґ	µ	¶	·	ё	№	є	»	ј	S	s	ї
C0	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
D0	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
E0	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
F0	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я

а

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
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	□	□	,	ƒ	„	…	†	‡	^	‰	Š	‹	Œ	□	□	□
90	□	‘	’	“	”	•	–	—	~	™	š	›	œ	□	□	Ÿ
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

б

Рис. 3.3. Кодировка символов в Windows:
а – для русифицированного шрифта Times New Roman Cyr;
б – для нерусифицированного шрифта Times New Roman

К сожалению, Microsoft приняла для своих русифицированных продуктов собственную кодировку русских символов, расходящуюся со стандартной, используемой многие годы в приложениях DOS. Это вызывает определенные сложности при составлении и отладке программ, так как исходные тексты приложений Windows, содержащие фразы на русском языке, можно просматривать только в текстовых редакторах Windows, что не всегда удобно.

Следует иметь в виду, что в систему Windows, как правило, встраивается большое количество различных шрифтов (Times New Roman, Courier New, Arial, Wingdings, Symbol и др.), отличающихся конфигурацией символов. Не все эти шрифты могут оказаться русифицированными; более того, многие шрифты включают в себя кроме (или вместо) обычных символов изображения разного рода стрелок, скобок, специальных математических знаков или даже небольших игровых картинок. Поэтому при выборе шрифта для вывода на экран текста надо сначала посмотреть, из чего он состоит. С точки зрения кодировки это означает, что один и тот же код при выводе на экран может отобразиться в символ самой неожиданной формы. Например, код 237, соответствующий для шрифтов Times New Roman Cyr или Courier New Cyr русской букве "н", в шрифте Symbol даст

изображение элемента вертикальной фигурной скобки ⎨, а в шрифте Wingdings – наклонной стрелки ⬋.

Как уже отмечалось, символьные переменные представляют собой собой просто однобайтовые числа со знаком. Разумеется, в тех случаях, когда в символьной переменной по смыслу ее использования хранится код символа (который, например, будет выводиться на экран), то и инициализировать такую переменную следует кодом символа:

```
char sym='!';//В переменную sym записывается код ASCII символа '!' (число 33)
```

Если же байтовая переменная используется для хранения числа (например, возраста человека), то и инициализировать ее следует числом:

```
char age=33;//В переменную age записывается возраст (число 33)
```

Заметьте, что обе переменные, и sym, и age, полностью эквивалентны. Более того, даже их содержимое в данном случае совпадает. Различие между ними может заключаться только в способе их использования и трактовке смысла содержащегося в них кода.

Указатели

Указатели представляют собой специфические переменные, значениями которых могут являться только адреса, по которым располагаются в памяти другие переменные. Указатели важны, разумеется, не сами по себе, а лишь потому, что с их помощью можно "добраться" до переменных, на которые они указывают. Во многих случаях такое опосредованное (косвенное) обращение к переменной оказывается более удобным, чем прямое. Указатели относятся к важнейшим типам данных и применяются в программах на языке C++ чрезвычайно широко. Однако нужда в них возникает главным образом при работе с массивами, структурами и функциями, поэтому содержательные примеры использования указателей нам придется отложить до рассмотрения этих вопросов; здесь будут даны только начальные сведения.

В программах для DOS, а также в 16-разрядных программах для Windows указатели могут быть ближними или дальними. Дальний указатель содержит полный адрес переменной – сегмент и смещение (или, в защищенном режиме, селектор и смещение) и позволяет обращаться к переменной, находящейся в любом сегменте. Ближний указатель состоит из одного смещения; с его помощью можно обратиться только к переменной в текущем сегменте. Соответственно дальний указатель занимает в памяти два слова, а ближний – одно.

В 32-разрядных программах, работающих в плоском адресном пространстве объемом 4 Гбайт, отпадает необходимость (и возможность) составлять программу из нескольких сегментов. Поэтому для таких программ все указатели имеют размер 32 бита (4 байта) и являются ближними.

Очевидно, что указатели на данные любого типа (целые, символьные и др.) физически ничем не отличаются друг от друга – все они содержат адрес некоторого данного, т. е. четырехбайтовое целое число. Однако объявляя указатель в программе на языке C++, мы должны обязательно определить, к переменным какого типа будет относиться данный указатель. Таким образом, для языка C++ указатель на целое – это один тип данных, а указатель на символ – другой тип.

Объявление указателя выполняется следующим образом:

```
int* pi;//Указатель pi на целое число
char* pc;//Указатель pc на символ
float* pf;//Указатель pf на число с плавающей точкой
```

Знак звездочки (*), стоящий после типа переменной, говорит о том, что объявляется не просто переменная, а указатель на переменную данного типа. Обратите внимание, что самих переменных еще нет и указателям пока не на что указывать. Компилятор только выделил место в памяти для хранения указателей; в них пока находятся произвольные случайные значения ("мусор").

Пусть в программе объявлены переменные i, c и f разных типов и для определенности им сразу присвоены некоторые значения:

```
int i=428;
char c='Q';
float f=1.25;
```

Для того чтобы инициализировать указатели адресами этих переменных, можно воспользоваться оператором &, который, встретившись в правой части выражения, обозначает адрес переменной:

```
pi=&i;
pc=&c;
pf=&f;
```

Как обратиться к переменной через ее указатель? Для этого надо воспользоваться тем же знаком *. Если он стоит в правой части выражения перед переменной-указателем, он уже обозначает не адрес, как при объявлении указателя, а данное по этому адресу. Такую операцию (получение данного по его адресу) называют снятием ссылки:

```
int j=*pi;//Переменная j принимает то же значение, что и i (т. е. 428)
(*pc)++;//Символ Q инкрементируется и превращается в R
*pf=*pf*2;//Переменная f умножается на 2 и принимает значение 2.50
```

Еще раз подчеркнем, что все наши указатели несовместимы друг с другом. Будучи указателями на данные разных типов, они сами тоже принадлежат к разным типам. Например, предложение

```
pc=pi;
```

приведет (при использовании компилятора Borland C++ 5.01) к сообщению об ошибке:

```
Cannot convert 'int*' to 'char*'
```

о невозможности преобразования указателя на int (int*) в указатель на char (char*).

Если нам нужно объявить несколько указателей одного типа, например указателей на символы, это можно сделать следующим образом:

```
char *p1, *p2, *p3;
```

При объявлении нескольких указателей в одном предложении перед каждым необходимо ставить знак звездочки, так как предложение

```
char* p1, p2, p3;
```

объявит переменную-указатель p1 и две *символьные* переменные (не указатели!) p2 и p3.

К чему же относится знак звездочки: к типу, делая его типом-указателем, или к имени переменной, делая ее переменной-указателем? Исходя из двух последних примеров, объявление указателя лучше трактовать таким образом: если *перед* именем переменной стоит знак *, то эта переменная является переменной-указателем. С другой стороны, в прототипах функций и иногда в их определениях используются обозначения вида int*, char* и т. д., где имя переменной отсутствует и звездочка явно относится к обозначению типа, говоря о том, что имеется в виду тип-указатель. Примирить это противоречие можно с формальной позиции: в предложениях языка C++ пробелы игнорируются компилятором (за исключением, разумеется, пробелов между стоящими рядом идентификатора-

ми) и следующие два объявления формально полностью эквивалентны, а объяснять их смысл каждый волен по-своему:

```
char* c;
char *c;
```

Довольно часто возникает необходимость использовать одну и ту же переменную в качестве указателя на *разные* типы данных в разных местах программы. Для этого достаточно объявить переменную как указатель на тип void:

```
void* AnyPtr;
```

После этого указателю AnyPtr можно присваивать адреса переменных любого типа: целочисленных, символьных, массивных и пр.

Рассмотрим на примере, как размещается и чем инициализируется указатель.

```
/*Программа 3-2. Указатели*/
#include <windows.h>
int var=0x12345678;//Объявим переменную типа int
int* pvar;//Объявим указатель на int
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
  char szText[100];
  pVar=&var;//Присваиваем указателю значение адреса переменной var
/*Выведем: значение var, адрес var, значение pvar, адрес pvar*/
  wsprintf(szText,"var: %#x\t@var: %#x\npvar: %#x\t@pvar: %3x",
                  var, &var, pvar, &pvar);
  MessageBox(NULL,szText,"Info",MB_OK);
  return 0;
  }
```

Вывод программы (рис. 3.4) показывает, что переменная var, как этого и следовало ожидать, имеет значение 0x12345678 и располагается по адресу 0x402074. Для указателя pVar компилятор выделил такую же четырехбайтовую ячейку на некотором удалении, по адресу 0x402220. Содержимое этой ячейки, т. е. значение указателя (после его инициализации) действительно оказалось равным адресу переменной var, составляющему 0x402074. Поскольку указатель представляет собой ячейку памяти, в него можно записать адрес любой другой переменной (но только типа int), после чего этот указатель будет указывать уже на другую переменную.

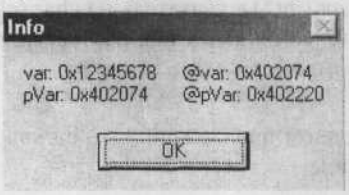


Рис. 3.4. Исследование указателя

Приведенные выше примеры носят несколько схоластический характер, так как при наличии одиночной целочисленной или символьной переменной нет особой необходимости вводить еще и указатель на нее. Указатели приобретают особую ценность прежде всего в двух случаях: при работе с коллекциями данных (массивами или структурами), а также для передачи параметров в вызываемые функции и получения от них результата. Позже мы приведем примеры такого использования указателей.

В Win32 предусмотрено большое количество специальных обозначений для указателей на переменные разного рода. Эти обозначения обычно начинаются с символов P (от Pointer, указатель) или LP (от Long Pointer, дальний указатель). Последнее обозначение перешло по наследству из Win16: в Win32, как известно, дальних указателей не существует и указатель, например LPINT, на целое в действительности является ближним.

Некоторые указатели начинаются с символов PC или LPC. Так обозначаются указатели на константные переменные. Заметьте, что константными здесь предполагаются не сами указатели, а те переменные или, лучше сказать, константы, адресами которых они являются. Впрочем, компилятор не проверяет "константность" переменной, с которой связан указатель типа, например LPCCH или LPCSTR. Указателю такого типа можно присвоить адрес и обычной, неконстантной переменной.

Многие указатели, объявленные в Win32, относятся к специальным переменным Windows. Например, тип FARPROC является указателем на специальный класс функций обратного вызова; тип PCRITICAL_SECTION описывает указатель на объект "критическая секция". Однако большое количество указателей относится к переменным общего назначения – целым, символьным и пр. В табл. 3.4 приведены обозначения некоторых указателей такого рода с указанием адресуемых ими типов данных.

Таблица 3.4. Наиболее распространенные обозначения указателей на типы данных общего назначения в Win32

Обозначение указателей	*Тип адресуемого данного*	*Примечание к типу адресуемого данного*
LPBOOL, PBOOL	BOOL	Булева переменная
LPBYTE, PBYTE	BYTE	Байт
LPCCH, PCCH	CONST CHAR	Константный символ Windows
LPCH, PCH	CHAR	Символ Windows
LPCSTR, PCSTR	CONST CHAR	Константная строка символов Windows с завершающим нулем
LPCTSTR	CONST TCHAR	Константная строка символов Windows или Unicode с завершающим нулем
LPCWCH, PCWCH	CONST WCHAR	Константный символ Unicode
LPCWSTR, PCWSTR	CONST WCHAR	Константная строка символов Unicode с завершающим нулем
LPDWORD, PDWORD	DWORD	Двойное слово
LPINT, PINT	INT	Целочисленная переменная
LPLONG, PLONG	LONG	В Win32 – то же самое
LPSTR, PSTR	CHAR	Строка символов Windows с завершающим нулем
LPTCH, PTCH	TCHAR	Символ Windows или Unicode
LPTSTR, PTSTR	TCHAR	Строка символов Windows или Unicode с завершающим нулем
LPVOID, PVOID	VOID	Любой тип
LPWCH, PWCH	WCHAR	Символ Unicode
LPWORD, PWORD	WORD	Слово
LPWSTR, PWSTR	WCHAR	Строка символов Unicode с завершающим нулем
NPSTR	CHAR	Строка символов Windows с завершающим нулем

Окончание табл. 3.4.

Обозначение указателей	*Тип адресуемого данного*	*Примечание к типу адресуемого данного*
NWPSTR	WCHAR	Строка символов Unicode с завершающим нулем
PBOOLEAN	BOOL	Булева переменная
PCHAR	CHAR	Символ Windows
PFLOAT	FLOAT	Переменная с плавающей точкой
PSHORT	SHORT	Короткое целое
PSZ	CHAR	Строка символов Windows с завершающим нулем
PTBYTE	TBYTE	Символ Windows или Unicode
PTCHAR	TCHAR	Символ Windows или Unicode
PUCHAR	UCHAR	Символ Windows без знака
PUINT	UINT	Целое без знака
PULONG	ULONG	Длинное целое без знака
PUSHORT	USHORT	Короткое целое без знака
PWCHAR	WCHAR	Символ Unicode

Исходя из данных, приведенных в этой таблице, для объявления, например, обобщенного указателя в приложении Windows можно с равным успехом использовать любое из следующих предложений:

```
void* AnyPtr;
PVOID AnyPtr;
LPVOID AnyPtr;
```

Ссылки

В языке C++ существуют два способа косвенного обращения к переменной. Один из них – указатели – был рассмотрен выше. Указатель достаточно нагляден в том отношении, что он является явным адресом переменной. Любые операции над указателями являются операциями над адресами и служат для получения доступа к тем или иным переменным.

Второй способ заключается в использовании специального типа данных – ссылочных переменных, или просто ссылок. По существу ссылочная переменная, как и указатель, представляет собой адрес некоторой переменной. Однако операции над ссылочными переменными воспринимаются компилятором как операции над самими переменными, а не над их адресами. Это упрощает обращение к переменным, так как избавляет от необходимости использовать операцию снятия ссылки (знак *). Для объявления в программе ссылочной переменной перед ней необходимо поставить значок амперсанда (&). Приведем предложения языка, демонстрирующие объявление и использование ссылочной переменной.

```
int &i=5;//Объявляется ссылочная переменная i со значением 5
i++;//Переменная принимает значение 6
i=i+10;//Переменная принимает значение 16
```

Несмотря на то что i представляет собой *адрес* некоторой (безымянной!) переменной, т. е. адрес ячейки памяти, в операциях с *содержимым* этой ячейки указывается просто обозначение i без знака снятия ссылки.

Между прочим, приведенный формальный пример неубедителен в том отношении, что в нем не доказывается "адресный" характер переменной-ссылки. Может быть, i – это просто значение переменной? Убедиться в том, что в действительности i представляет собой именно адрес, а не значение переменной, можно с помощью отладчика. Следует заметить, что переменные-ссылки используются главным образом в качестве аргументов при вызове функций. Именно в этом качестве проявляются все их достоинства и особенности. Обсуждение этого вопроса мы отложим до раздела, посвященного функциям.

Массивы

Массив представляет собой последовательность некоторого количества данных *одного типа*. Соответственно в программе могут быть массивы целых чисел или чисел с плавающей точкой, символьные массивы и т. д. Весь массив в целом характеризуется именем, а обращение к элементам массива выполняется по их индексам, которые всегда начинаются от нуля. Если, например, объявить в программе массив по имени data из 1000 целых чисел

```
int data[1000];
```

то самый первый элемент массива получает обозначение data[0], следующий – data[1] и т. д. до последнего элемента data[999].

Точно так же можно объявить массивы данных любых других типов:

```
char city[20];//Массив из 20 символов
float weight[16]//Массив из 16 чисел с плавающей точкой
```

Если в массиве не слишком много элементов и их значения известны заранее, массив можно инициализировать вместе с его объявлением, заключив перечень значений в фигурные скобки:

```
int tens [10]={10,20,30,40,50,60,70,80,90,100};
```

Если же массив большой, заполнить его придется программно в цикле. Пусть мы хотим образовать тестовый массив из 100 элементов, заполненных натуральным рядом чисел. Это делается следующим образом:

```
int test [100];
for(int i=0;i<100;i++)
  test[i]=i;
```

Использованное в этом фрагменте предложение for служит для выполнения следующего за ним предложения

```
test[i]=i;
```

100 раз при значениях i от 0 до 99, в результате чего элемент массива test[0] приравнивается нулю, элемент test[1] – единице и т. д.

Разумеется, кроме инициализации всего массива в цикле к элементам массива можно обращаться и выборочно, например

```
test[50]=0xFFFF;//Присвоим 51-му элементу массива test значение 0xFFFF
int x=test[2];//Объявим переменную x и присвоим ей значение
              //3-го элемента массива test
```

Символьные массивы

Символьные массивы имеют некоторые особенности инициализации. Следует заметить, что символьный массив можно использовать двояко. Во-первых, он может представлять коллекцию отдельных символов, например последовательность букв алфавита или цифр. По ходу выполнения программа обращается, как и в случае числового массива, то к одному, то к другому элементу массива по его индексу. Во-вторых, символьный массив может представлять связную символьную строку (фамилия человека, имя какого-то объекта Windows, вообще текст любой длины). В этом случае разумно обращаться только ко всему массиву целиком по его имени, а обращения к отдельным элементам-символам, хотя и вполне возможны, но не имеют смысла.

В первом случае массив можно инициализировать так же, как и числовой, перечислением всех его элементов:

```
char digits[10]={'0','1','2','3','4','5','6','7','8','9'};
```

Такой массив занимает в памяти точно 10 байт (рис. 3.5).

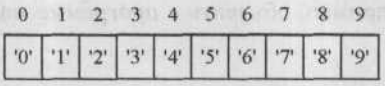


Рис. 3.5. Символьный массив, инициализированный 10 символами

Во втором случае массив инициализируется символьной строкой:

```
char FamilyName[15]="Григорович";
```

При такой инициализации компилятор записывает в память в конце строки двоичный нуль, который служит символом-ограничителем (рис. 3.6). Многие функции для работы со строками (например, копирования строки или вывода строки на экран) по этому символу определяют, где кончается строка. Это избавляет нас от необходимости определять и указывать при вызове функции фактическую длину строки. Таким образом, символьная строка всегда занимает в памяти на 1 байт больше, чем в ней есть значащих символов. Это обстоятельство необходимо учитывать при задании длины символьного массива в его объявлении, которая должна выбираться на единицу больше максимально возможной длины помещаемой в него строки.

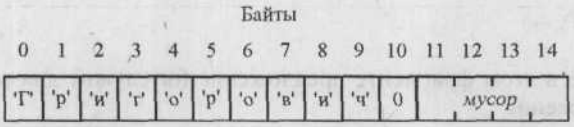


Рис. 3.6. Символьный массив, инициализированный символьной строкой

Объявление длины символьного массива с запасом целесообразно лишь в тех случаях, когда он будет заполняться программно строками разной длины. Если же данная текстовая строка изменяться не будет, удобнее объявить ее без указания длины, оставив в объявлении пару квадратных скобок, которые скажут компилятору, что переменная FamilyName является не одиночным символом, а символьным массивом:

```
char FamilyName[]="Григорович";
```

Компилятор, выделяя память под этот массив, сам определит его длину, которая в данном случае будет равна 11 байтам (10 байт под собственно строку и 1 байт под завершающий нуль, рис. 3.7).

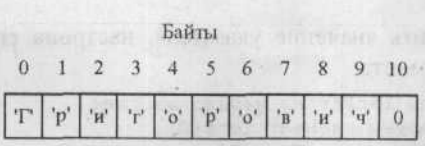


Рис. 3.7. Символьный массив неопределенной длины, инициализированный символьной строкой

Массивы и указатели

Как уже отмечалось, в программах на языке C++ наряду со значениями переменных можно использовать их адреса, для работы с которыми введен специальный тип данных – указатели. Рассмотрим вопрос о значении переменной, адресе и указателе применительно к массиву.

В отличие от скалярных переменных, для которых *имя переменной* отождествляется с ее *значением*, *имя массива* отождествляется с его *адресом*, при этом адрес всего массива совпадает с адресом его начального элемента. Однако обозначения *элементов массива* отождествляются с их *значениями*.

Пусть в программе объявлен массив Tens из 10 целых чисел, инициализируемый при его объявлении:

```
int Tens[10]={0,10,20,30,40,50,60,70,80,90};
```

Несмотря на то, что массив является переменной в том смысле, что любому элементу массива можно присвоить любое допустимое значение, *имя* Tens переменной не является. Имя массива является символическим обозначением его адреса и, таким образом, представляет собой по существу константу (ее можно назвать указателем-константой), так как после того, как компилятор выделил в памяти место под этот массив, его адрес становится фиксированной величиной и изменен быть не может.

Элементы массива в данном случае именуются tens[0], tens[1], tens[2] и т. д., причем значение Tens определяет как адрес всего массива, так и адрес его первого элемента Tens[0].

Если ввести в программу новую переменную типа int*, т. е. типа указателя на целое

```
int* intPtr;
```

и присвоить ей значение адреса массива Tens

```
intPtr=Tens;
```

то после этого к нашему массиву можно с равным успехом (и одинаково синтаксически!) обращаться как с помощью его имени, так и через новый указатель:

```
Tens[1]=110;//Изменяем значение второго элемента массива
intPtr[2]=120;//Изменяем значение третьего элемента массива
```

Другими словами, указатель, так же, как и имя массива, можно индексировать.

В соответствии с табл. 3.2 вместо обозначения int* можно использовать типы PINT или LPINT. Результат будет тот же самый:

```
PINT intPtr;
```

Выше был приведен пример применения к указателю операции присваивания, когда переменная intPtr была инициализирована адресом массива. С таким же успехом можно было присвоить переменной intPtr значение адреса любого конкретного элемента массива и затем работать с этим элементом:

```
intPtr=&Tens[3];//intPtr равен адресу элемента массива с индексом 3
*intPtr=300;//Теперь Tens[3]=300 (операция снятия ссылки!)
```

Другой способ изменить значение указателя, настроив его на требуемый элемент массива, – операция инкремента.

```
intPtr=Tens;//Настраиваем intPtr на начало массива
*intPtr=200;//Инициализируем элемент Tens[0]
intPtr++;//Теперь intPtr=адрес Tens[1]
*intPtr=210;// Инициализируем элемент Tens[1]
intPtr++;// Теперь intPtr=адрес Tens[2]
*intPtr=220;// Инициализируем элемент Tens[2]
```

Важно отметить, что операция инкремента над указателем изменяет его значение не на 1, а на величину, равную длине переменной, адресуемой через указатель, причем длину переменной автоматически определяет компилятор. В нашем примере элементами массива являются 4-байтовые числа и операция инкремента увеличивает значение указателя на 4. Если бы массив Tens состоял из 2-байтовых чисел, каждая операция

```
intPtr++
```

увеличивала бы значение intPtr на 2.

То же относится и к другим операциям над указателями. Например, операция декремента

```
intPtr--
```

уменьшит значение указателя не на 1, а на длину данного (4 для целых чисел в Win32), а, казалось бы, очевидная операция

```
intPtr=intPtr+3
```

изменит значение intPtr совсем не на 3, а на 3x4=12 (в Win32) или на 3x2=6 (в Win16). Продемонстрируем вышеизложенное на примере.

```
/*Программа 3-3. Арифметические операции над указателями*/
#include <windows.h>
int m[5]={100,200,300,400,500};//Объявим и инициализируем массив целых (int)
int* pm;//Объявим указатель на int
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
  char szText[100];
  pm=m;//Инициализируем указатель адресом массива m
/*Выведем значения указателя и адресуемого через него
  элемента массива*/
  wsprintf(szText,"pm: %#x  Элемент массива: %#d", pm, *pm);
  MessageBox(NULL,szText,"Info",MB_OK);
  pm++;//Выполним инкремент указателя
/*Выведем значения указателя и адресуемого через него
  элемента массива*/
  wsprintf(szText,"pm: %#x\  Элемент массива: %#d", pm, *pm);
  MessageBox(NULL,szText,"Info",MB_OK);
  return 0;
  }
```

Из рис. 3.8 видно, что после инкремента указателя его значение увеличилось не на 1, а на 4 (длина переменной типа int), и он стал указывать на следующий элемент массива m.

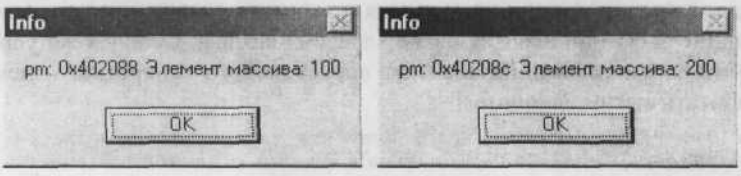


Рис. 3.8. Демонстрация инкремента указателя

Выше уже отмечалось, что указатели на данные разных типов несовместимы друг с другом. Поэтому нельзя, например, инициализировать указатель на целое адресом символьного массива:

```
char message[80];//Объявление символьного массива
int* addr;//Объявление указателя на целое
addr=message;//Неверно. Указатель addr следовало объявить как char* или PCHAR
```

Многомерные массивы

Массивы, о которых до сих пор шла речь, были одномерными, потому что для обращения к какому-либо элементу массива требовался один-единственный индекс. Такие массивы часто называют векторами или строками.

Во многих случаях данные удобно представлять не в виде линейной последовательности, а в форме таблицы, или матрицы, в которой данные занимают несколько строк. Тогда для обращения к конкретному элементу надо задать два индекса: номер строки, в которой находится элемент, и номер элемента в строке (номер столбца). Такие массивы данных называют двумерными.

Матрица может иметь одинаковые размеры по горизонтали и вертикали, и тогда она называется квадратной; при разных размерах мы получим прямоугольную матрицу. Элементами матрицы могут быть любые данные: целые, с плавающей точкой, символы и др.

Кстати, представителем двумерных матриц можно считать экран терминала. В текстовом режиме на нем размещается 25 строк по 80 символов, что представляет собой матрицу символов размером 80x25. В графическом режиме экран можно рассматривать как матрицу пикселов размером, например, 800x600.

Другой пример двумерной матрицы – записанный в цифровой форме рельеф местности, где значение каждого элемента массива соответствует высоте данной точки местности (с координатами x и y) над уровнем моря.

Массивы могут иметь число измерений (т. е. число индексов, определяющих местоположение в массиве отдельного элемента) и больше двух. Например, распределение температуры в некотором объеме описывается трехмерным массивом, в котором 3 индекса каждого элемента отвечают x-, y- и z-координатам соответствующей точки объема. В общем случае массивы с числом измерений больше единицы называются многомерными.

Рассмотрим правила объявления и инициализации многомерных массивов.

Предположим, что для решения некоторой задачи нам потребовалась таблица чисел, изображенная на рис. 3.9.

1	2	3	4
10	20	30	40
100	200	300	400

Рис. 3.9. Пример числовой таблицы

Эту таблицу можно представить в программе в виде массива с размерностью 3x4:

```
int twod[3][4];
```

В квадратных скобках указываются размеры массива; первое число относится к строкам, второе – к столбцам. Для обращения к отдельным элементам массива надо указывать оба индекса, причем индексы будут соответствовать элементам так, как это показано на рис. 3.10.

Строка 0	twod[0][0]	twod[0][1]	twod[0][2]	twod[0][3]
Строка 1	twod[1][0]	twod[1][1]	twod[1][2]	twod[1][3]
Строка 2	twod[2][0]	twod[2][1]	twod[2][2]	twod[2][3]
	Столбец 0	Столбец 1	Столбец 2	Столбец 3

Рис. 3.10. Индексация элементов двумерного массива

Если массив желательно инициализировать при его объявлении, это надо сделать следующим образом:

```
int twod[3][4]={{1,2,3,4},{10,20,30,40},{100,200,300,40}};
```

Сначала надо перебрать все значения элементов строки 0, затем элементов строки 1 и т. д. Обратите внимание на то, что в фигурные скобки заключаются как наборы данных каждой строки, так и (еще раз) все данные вместе.

Как и для одномерных массивов, имя двумерного массива отождествляется с его адресом (а также с адресом его первого элемента twod[0][0]). Точно так же для получения адреса произвольного элемента можно воспользоваться оператором &, например &twod[2][3]. Однако для двумерных массивов, помимо адресов элементов, разумно выделить еще адреса строк, которые могут понадобиться, если требуется обработать не весь массив целиком, а его отдельные строки. Такая возможность предоставляется компилятором, который воспринимает имя *двумерного* массива с *одним индексом* как адрес соответствующей индексу строки. Таким образом, обозначение twod[0] отождествляется с адресом строки 0 (совпадающим, естественно, с адресом всего массива twod, как и с адресом его самого первого элемента &twod[0][0]), обозначение twod[1] – с адресом строки 1 и т. д.

Структуры

Структура, как и массив, представляет собой коллекцию данных, объединенных общим именем. Однако если в массиве все элементы должны быть одного типа, структура может содержать элементы (их называют членами структуры), принадлежащие к любым типам данных. В частности, членами структуры могут быть массивы и вложенные структуры.

Структура определяется с помощью ключевого слова struct, за которым обычно следует имя структуры, выступающее в качестве нового типа данных или, точнее, имени конкретного структурного типа, а затем в фигурных скобках описание ее членов. Рассмотрим в качестве примера сугубо условную структуру, служащую для записи данных о фамилии и возрасте человека:

```
struct Men{//Созданный пользователем новый тип
  char* name;//Указатель на фамилию индивидуума
  int age;//Его возраст
  };
```

Описание структуры вводит новый тип данных (в нашем случае Men), что позволяет в дальнейшем создавать в программе конкретные структурные переменные этого типа:

```
Men man1,man2;//Объявлены две структурные переменные man1 и man2 типа men
```

Для обращения к членам структурной переменной используется специальный оператор – точка:

```
man1.name="Силин";
man1.age=35;
man2.name="Перевозчиков";
man2.name=73;
```

Обратите внимание на переменную, служащую для хранения фамилии человека. Почему она объявлена с помощью указателя? Дело в том, что при объявлении ее более простым образом:

```
char name[25];
```

мы столкнемся с трудностями помещения в этот массив конкретной фамилии. Естественная, казалось бы, операция присваивания

```
man1.name="Турандот";
```

в данном случае оказывается совершенно неправильной, так как имя любого массива, как уже отмечалось, является указателем-константой (в отличие от указателя-переменной). Имени массива нельзя присвоить другое значение, так же как числу 5 нельзя присвоить значение 10. Для того чтобы в объявленный выше массив ввести конкретное содержимое, пришлось бы воспользоваться операцией копирования:

```
char name[25];
strcpy(man1.name,"Турандот");//Используем функцию C++ копирования строк
```

Введя же в структуру указатель-переменную, мы получили возможность использовать для инициализации массива более простую операцию присваивания.

Существуют разные способы объявления и инициализации структуры. Если в программе предполагается использование единственной структурной переменной, ради которой только и описана структура, то имя переменной можно указать при объявлении структуры:

```
struct Men{
  char* name;
  int age;
  }man1;//Создание конкретной структурной переменной man1 типа men
```

Но в таком случае лишается смысла наименование типа и его можно опустить:

```
struct{
  char* name;
  int age;
  }man1;//Создание конкретной переменной man1 типа этой структуры
```

Инициализация структурной переменной возможна при ее объявлении, причем перечень значений ее членов (как и элементов массива при инициализации массива) заключается в фигурные скобки:

```
Men man1={"John",55};
```

Естественно, независимо от способа инициализации всегда остается возможность обращения к членам созданной структурной переменной с помощью оператора "." :

```
man1.name="Crony";
```

Описанная выше методика обращения к членам структурных переменных предполагает, что нам известны имена этих переменных (man1 в предыдущем примере). Существует и другой способ объявления и использования структурных переменных – с помощью указателей на них:

```
Men* pman1;//Объявление указателя на структурную переменную типа Men
```

Использование указателя на структуру требует некоторых дополнительных действий. Приведенное выше предложение создает только переменную-указатель, но, во-первых, он не инициализирован никаким разумным адресом и, во-вторых, самой структурной пе-

ременной у нас еще нет. Обе эти операции – создание структурной переменной и передача ее адреса указателю – выполняются в предложении

```
pman1=new Men;
```

Оператор new служит для динамического выделения памяти. В качестве аргумента этот оператор требует указания *типа* той переменной, которой назначается выделенная память. В нашем случае память нужна для структурной переменной типа Men. Оператор new, выделив память, возвращает указатель на нее, который мы и присваиваем переменной m. Часто можно использовать более компактную форму получения указателя на структурную переменную, объединяющую все три операции: объявление указателя, выделение памяти под структурную переменную и инициализация укзателя адресом этой переменной:

```
Men* pman1=new Men;
```

Заметим, что при таком объявлении структурной переменной, у нее не оказыается имени, и обращаться к ней можно только посредством указателя. При этом если структурная переменная объявляется с помощью указателя, то для обращения к членам структуры оператор "." заменяется на "->":

```
pman1->name="Абрикосов";
pman1->age=18;
```

Выше уже отмечалось, что в языке C++ имя скалярной переменной отождествляется с ее значением, а имя массива – с его адресом. Поэтому если нам надо передать в некоторую функцию (о функциях – в последующих разделах) в качестве ее аргумента *адрес* переменной, то обозначение такого аргумента для скалярной и массивной переменных будет выглядеть по-разному:

```
int value;//Объявляется скалярная целочисленная переменная
int array[100];//Объявляется массив целых чисел
Func(&value, array);//В функцию передаются адрес переменной и адрес массива
```

Имя массива уже является его адресом, и в качестве параметра следует указать имя array. Имя же скалярной переменной является не адресом, а значением и если требуется передать адрес этой переменной, следует использовать обозначение &value.

Однако как компилятор языка C++ воспринимает имя структуры? Хотя структура относится к агрегатным данным, ее имя *не* отождествляется с ее адресом. Скорее под именем структуры понимается совокупность значений всех ее членов. Адрес же структуры, так же, как и адрес скалярной переменной, обозначается указанием перед именем структуры знака амперсанда &. С другой стороны, в функцию можно передать как всю структуру (по имени), так и ее адрес:

```
Men man1;
Func1(man1);//В функцию Func1 передается имя структуры, т. е. вся структура
Func2(&man1);//В функцию Func2 передается адрес структуры
```

Приведем пример создания, инициализации и вывода в окно сообщения структурных переменных.

```
/*Программа 3-4. Структуры*/
#include <windows.h>
/*Структура, описывающая счет в банке*/
struct Account{
  int account;//Номер счета
  char* name;//Имя вкладчика
  int balance;//Остаток вклада
  };
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
```

```
  char szText[100];
/*Создадим первую структурную переменную (по имени)*/
  Account acc1;
  acc1.account=346578;
  acc1.name="Груздев";
  acc1.balance=15000;
/*Создадим вторую структурную переменную (посредством указателя)*/
  Account* pacc2=new Account;
  pacc2->account=901005;
  pacc2->name="Блохина";
  pacc2->balance=23500;
/*Выведем данные о счетах клиентов в окно сообщения*/
  wsprintf(szText,"Счет: %d  Клиент: %s  Остаток: %d\n"
                  "Счет: %d  Клиент: %s  Остаток: %d",
                  acc1.account, acc1.name, acc1.balance,
                  pacc2->account, pacc2->name, pacc2->balance);
  MessageBox(NULL,szText,"Info",MB_OK);
  return 0;
  }
```

На рис. 3.11 приведен вывод программы.

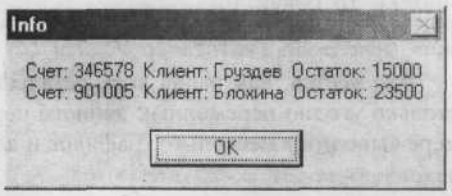


Рис. 3.11. Вывод структурных переменных

Перечислимые типы данных

Перечислимый тип представляет собой своеобразный шаблон, заготовку для типов данных, создаваемых программистом и состоящих из набора определенных заранее *констант*. Переменная конкретного перечислимого типа может принимать только целочисленные значения, соответствующие входящим в данный перечислимый тип константам.

Перечислимый тип объявляется с помощью ключевого слова enum, за которым обычно следует имя создаваемого конкретного типа, а затем в фигурных скобках – перечень идентификаторов, называемых константами перечислимого типа.

Предположим, что в программе, выводящей на экран график некоторого процесса, требуется изменять режим вывода графика, отображая его точками, сплошной линией или в виде ступенчатой кривой (гистограммы). Переменную mode, характеризующую текущий режим вывода, удобно объявить переменной перечислимого типа, для чего предварительно ввести сам перечислимый тип:

```
enum Graphmode {POINTS,CURVE,HISTO};
```

Созданный нами перечислимый тип по имени graphmode включает всего 3 константы – POINTS, CURVE и HISTO, причем по умолчанию они принимают значения по порядку следования 0, 1 и 2.

Далее мы должны объявить переменную (с произвольным удобным нам именем) типа Graphmode:

```
Graphmode mode;
```

Эта переменная сможет принимать только значения POINTS, CURVE и HISTO. Программа, анализируя в дальнейшем значение переменной mode, будет определять режим своей работы:

```
if(mode==POINTS)//Если установлен режим точек,
  Points();//вызывать прикладную функцию Points() вывода графика по точкам
```

```
if(mode==CURVE)//Если установлен режим кривой,
  Curve();//вызвать прикладную функцию Curve() вывода графика в виде линии
else//Если ни то и ни другое, значит, режим гистограммы
  Histo();//Вызвать прикладную функцию Histo() вывода гистограммы
```

В этом фрагменте в зависимости от установленного режима вызывается та или иная прикладная функция для вывода графика.

При желании можно присвоить константам перечислимого типа вполне определенные значения, например 100, 101 и 102. Тогда объявление типа будет выглядеть следующим образом:

```
enum Graphmode {POINTS=100,CURVE=101,HISTO=102};
```

Наконец, конкретные значения констант можно определить с помощью директивы препроцессора #define (о препроцессоре речь будет идти ниже), что даст возможность использовать эти константы и в других конструкциях программы:

```
#define ID_POINTS 100
#define ID_CURVE 101
#define ID_HISTO 102
enum Graphmode {POINTS=ID_POINTS,CURVE=ID_CURVE,HISTO=ID_HISTO};
```

Разумеется, как и для всех остальных типов данных, в программе можно объявить сколько угодно переменных данного перечислимого типа (если, например, в нашем примере выводится несколько графиков и для каждого требуется задавать свой режим):

```
Graphmode mode1,mode2,mode3;
```

Переменную перечислимого типа (одну или несколько) можно объявить совместно с объявлением типа, задав ей при желании требуемые начальные значения:

```
enum Graphmode {POINTS=ID_POINTS,CURVE=ID_CURVE,HISTO=ID_HISTO}
                mode1=POINTS,mode2=POINTS,mode3=HISTO;
```

Если в программе имеется только одна переменная некоторого перечислимого типа, то можно обойтись без объявления имени этого типа (все равно оно нигде не будет использоваться):

```
enum {POINTS,CURVE,HISTO} mode;
```

Некоторые операции над данными

При обработке скалярных данных можно использовать обычные арифметические операторы сложения (+), вычитания (-), умножения (*) или деления (/). Кроме них в языке C++ предусмотрен еще ряд операций не столь очевидного характера. Некоторые из них будут рассмотрены ниже.

Побитовые операции

Побитовые, или поразрядные, операции предназначены для воздействия на отдельные биты переменных и констант. Эти операции, в частности, необходимы для программ, обеспечивающих управление аппаратурой на физическом уровне путем обращения к ее регистрам, поскольку обычно в регистрах аппаратуры каждый бит несет определенную функцию и посылка, например, в некоторое устройство команды включения сводится к установке определенных битов во входном порте этого устройства.

Другая область использования побитовых операций – работа с числами, обычно константами, в которых каждый бит является флагом того или иного свойства или режима. Такие числа-флаги широко используются и в прикладном программировании, и в операционных системах, в частности в системе Windows.

В табл. 3.5 представлены все побитовые операции.

Таблица 3.5. Побитовые операции

Операция	*Назначение*	*Операция*	*Назначение*
~	НЕ	|	Включающее ИЛИ
&	И	>>	Сдвиг вправо
^	Исключающее ИЛИ	<<	Сдвиг влево

Операция побитового отрицания НЕ инвертирует двоичный код числа, т. е. все двоичные единицы заменяет на двоичные нули и наоборот (рис. 3.12).

Исходное число	9	2	C	4
Оно же	1001	0010	1100	0100
Результат операции	0110	1101	0011	1011
Он же	6	D	3	B

Рис. 3.12. Операция побитового отрицания (~)

Операция побитового И выполняется над двумя операндами. В результирующем числе устанавливаются (в единицу) только те двоичные разряды, в которых в обоих операндах установлены единицы. Если хотя бы в одном из операндов в некотором разряде 0, то и в результате в этом разряде будет тоже 0. Обычно один из операндов представляет собой исходное число, а второй – маску, с помощью которой выделяются требуемые биты операнда. На рис. 3.13 представлено выделение 4 старших битов двухбайтового числа.

Исходное число	9	2	C	4
Оно же	1001	0010	1100	0100
Операнд-маска	1111	0000	0000	0000
Результат	1001	0000	0000	0000
Он же	9	0	0	0

Рис. 3.13. Операция побитового И (&)

Операция побитового включающего ИЛИ, которою часто называют просто операцией ИЛИ, также выполняется над двумя операндами, и также обычно один из них является исходным числом, а второй – маской, с помощью которой в исходное число добавляются единицы в тех двоичных разрядах, которые установлены в маске. На рис. 3.14 двоичные единицы добавляются в 0, 1 и 15. Поскольку разряд 15 в исходном числе и так был установлен, его состояние не изменяется.

Исходное число	9	2	C	4
Оно же	1001	0010	1100	0100
Операнд-маска	1000	0000	0000	0011
Результат	1001	0010	1100	0111
Он же	9	2	C	7

Рис. 3.14. Операция побитового включающего ИЛИ (|)

Операция побитового исключающего ИЛИ используется относительно редко. Она отличается от обычного ИЛИ тем, что при наличии в некотором двоичном разряде единицы в обоих операндах соответствующий разряд результата сбрасывается в 0 (рис. 3.15).

Исходное число	9	2	C	4
Оно же	1001	0010	1100	0100
Операнд-маска	1000	0000	0000	0011
Результат	0001	0010	1100	0111
Он же	1	2	C	7

Рис. 3.15. Операция побитового исключающего ИЛИ (^)

Кроме описанных выше побитовых логических операций, в C++ существуют еще побитовые операции сдвига. С их помощью все биты операнда можно сдвинуть влево или вправо на заданное число двоичных разрядов. Таких операций две: сдвиг влево, т. е. в сторону старших битов (<<), и сдвиг вправо, т. е. в сторону младших битов (>>).

```
USHORT value=0xC007;//value=1100 0000 0000 0111 (двоичное)
value=value>>2;//value=0011 0000 0000 0001 (двоичное)
USHORT value=3;//value=0000 0000 0000 0011 (двоичное)
value=value<<8;//value=0000 0011 0000 0000 (двоичное)
```

Обратите внимание, что при сдвиге числа без знака (USHORT) вправо освобождающиеся левые биты заполняются нулями; при сдвиге влево нулями заполняются освобождающиеся правые биты. Числа со знаком ведут себя при сдвиге влево так же, а при сдвиге вправо – иначе:

```
SHORT value=0x8004;//value=1000 0000 0000 0100 (двоичное)
value=value>>2;//value=1110 0000 0000 0001 (двоичное)
```

При сдвиге числа со знаком вправо его знаковый разряд, как говорят, расширяется вправо, заполняя освобождающиеся в результате сдвига биты.

Предположим, мы хотим упаковать в одно машинное слово полную дату (год, месяц и день) в том формате, в котором дата хранится и представляется в DOS (рис. 3.16).

Рис. 3.16. Формат даты, используемый DOS

Такую операцию выполняет приведенный ниже фрагмент:

```
WORD date;          //Слово для полной даты
WORD year=26;       //Год от 1980
WORD month=12;      //Декабрь
WORD day=31;        //День месяца
year=year<<9;       //Сдвинем год в биты 8...15
month=month<<5;     //Сдвинем месяц в биты 5...8
date=day;           //Отправим день в data
date=date|month;    //Добавим биты месяца
date=date|year;     //Добавим биты года
```

Воспользовавшись операциями составного присваивания (см. раздел ниже) и заметив, что последние 3 предложения можно объединить в одно, получим следующую редакцию того же фрагмента:

```
year<<=9;                   //Сдвинем год в биты 8...15
month<<=5;                  //Сдвинем месяц в биты 5...8
date=date|day|month|year;   //Сложим все побитовыми операциями ИЛИ
```

Операции присваивания

В языке C++ помимо естественной для любого языка операции присваивания переменной определенного значения

```
int x=360;
char symbol='*';
char title[]="Учебное пособие";
```

существует еще ряд операций составного присваивания, которые объединяют операции присваивания с арифметическими или логическими операциями. Ниже приведены пары эквивалентных строк обычной записи и составного присваивания. Разумеется, в реальной программе переменная value должна быть объявлена соответствующим образом (например, для операций с дробными величинами как float). Для наглядности в примерах используются константы, хотя во всех случаях константу можно заменить второй переменной.

```
value=value+5;//К переменной value прибавляется 5
value+=5;//То же самое

value=value-12;//Из переменной value вычитается 12
value-=12;//То же самое

value=value*1.25;//Переменная value умножается на 1.25
value*=1.25;//То же самое

value=value/3.5;//Переменная value делится на 3.5
value/=3.5;//То же самое

value=value&0xF;//В переменной value остаются только 4 младших бита
value&=0xF;//То же самое

value=value|0xF;//В переменной value устанавливаются 4 младших бита
value|=0xF;//То же самое

value=value<<5;//Переменная value сдвигается влево на 5 бит
value<<=5;//То же самое

value=value>>8;//Переменная value сдвигается вправо на 8 бит
value>>=8;//То же самое
```

К операциям присваивания относят также операции инкремента, или увеличения на 1 (++), и декремента, или уменьшения на 1 (--), при этом и та и другая операция может быть постфиксной или префиксной.

При постфиксной операции, когда обозначение ++ или -- стоит *после* обозначения переменной, входящей в какое-то выражение, в выражении используется текущее значение переменной и лишь после этого выполняется ее модификация.

При префиксной операции, когда обозначение ++ или -- стоит *перед* обозначением переменной, сначала выполняется модификация переменной, а затем уже новое значение переменной используется в выражении.

Рассмотрим формальный пример, поясняющий действие постфиксной и префиксной операций.

```
char args[]="ABCDEFGHIJKLMNOPQRSTUVWXYZ";//Символьная строка
char symbol;//Переменная для хранения одного символа
int index;//Индекс в строке
index=3;//Пусть индекс = 3
sym=args[i++];
/* Теперь index=4 и указывает на 'E', но sym='D' */
index=3;//Пусть индекс снова = 3
sym=args[++i];
/* Теперь index =4 как и раньше и указывает на 'E', и sym тоже ='E' */
```

Условная операция ?:

Условная операция может в некоторых случаях служить заменой конструкции if...else. Она требует трех операндов и в общем виде имеет следующий формат:

```
условное_выражение  ?  выражение_1 : выражение_2;
```

Если условное выражение истинно, то выполняется *выражение_1*; если условие ложно, выполняется *выражение_2*.

Например, предложение

```
a>b ? x=1:y=10;
```

расшифровывается так: если a действительно больше b, то выполняется присваивание x=1; если a не больше b, выполняется присваивание y=10.

Часто выражения 1 и 2 используют одну и ту же переменную, которой присваивается либо одно, либо другое значение. Тогда условная операция записывается несколько иначе:

```
переменная = условное_выражение ? выражение_1 : выражение_2;
```

В этом случае переменная, стоящая слева, получает при выполнении условия значение выражения 1, а при невыполнении – значение выражения 2, например

```
char ans=(x<=max)?'Y':'N';
```

Если x<=max, то переменная ans принимает значение 'Y', если x>max, то ans будет равна 'N'. Скобки в этом примере введены только для наглядности. Действительно, операции сравнения, в частности <=, имеют более высокий приоритет, чем операция присваивания. Поэтому сначала проверяется условие и лишь затем выполнятся условное присваивание.

Операция sizeof

Операция sizeof, напоминающая по форме вызов функции, служит для определения размера операнда в байтах. Она может использоваться как с обозначением переменной, так и с ее типом (в последнем случае операнд следует заключить в круглые скобки). Операция sizeof особенно полезна для определения размеров агрегатных переменных – массивов и структур.

Примеры использования операции sizeof:

```
char str[]="Символьная строка";//Символьный массив
int StrLength=sizeof str;//Размер конкретного массива (18 байт)

RECT r;//Переменная для хранения координат прямоугольника
int SizeOfRect=sizeof r;//Размер структуры RECT (4*4 байт = 16 байт)
```

Управление ходом программы

Во многих случаях дальнейшее выполнение программы от некоторой точки должно идти по разным путям в зависимости от определенных условий. Например, при выборе одного пункта меню программа должна приступить к выполнению ветви 1, а при выборе другого пункта – к выполнению ветви 2. Такие *условные переходы* совсем не обязательно связаны с командами пользователя; программа может сама выбрать дальнейший путь, например по результатам некоторой математической операции: при нулевом результате выполнить один фрагмент, при отрицательном – другой, при положительном – третий. В языке C++ имеется группа предложений и операторов, позволяющих осуществлять анализировать определенные условия и осуществлять переходы по результатам анализа. Рассмотрим эти возможности.

Предложение if и операции отношения

Предложение if позволяет проанализировать указанное условие и осуществить выполнение некоторого фрагмента программы только при положительном результате анализа.

Предложение if имеет следующий формат:

```
if(выражение){
  //Блок из одного или нескольких предложений C++
  }
```

Если блок предложений состоит из одного предложения, фигурные скобки можно опустить, но можно и оставить для того, чтобы подчеркнуть, что́ именно входит в блок.

Помещенный в фигурные скобки блок предложений выполняется лишь в том случае, если указанное в скобках после ключевого слова if *выражение* истинно. Понятие истинности несколько различается для выражений разного рода. Результат арифметической операции (сложения, вычитания и пр.) считается ложным, если он равен нулю, и истинным в противном случае. Это же правило относится к переменным и константам. Так, числа 1, 25 и -456 истинны, а 0 (и только 0) ложен. Переменная var истинна, если она в момент анализа имеет любое значение, кроме нуля, и ложна, если она оказалась равна нулю. Операции отношения, т. е. сравнения данных и проверки на заданное условие (больше, меньше, больше или равно, меньше или равно и др.) истинны, если условие имеет место. Логические операции И, ИЛИ и НЕ (см. следующий подраздел) схожи с арифметическими: нулевой результат считается ложным, ненулевой – истинным.

В качестве анализируемого выражения в предложении if чаще всего используется одна из операций отношения (сравнения). Всего таких операций 6 (табл. 3.6).

Таблица 3.6. Операции отношения

Операция	*Описание*	*Операция*	*Описание*
==	Равно	>=	Больше или равно
>	Больше	<=	Меньше или равно
<	Меньше	!=	Не равно

Приведем несколько примеров использования предложений if.

```
int data[1000];//Массив экспериментальных данных
for(int i=0;i<1000;i++)//Цикл по всем элементам массива
  if(data[i]<=0)
    data[i]=1;//Преобразуем отрицательные и нулевые значения в 1
```

```
int data[500];//Массив экспериментальных данных
int counter;//Счетчик нулей в массиве данных
for(int i=0;i<500;i++)//Цикл по всем элементам массива
  if(data[i]==0)
    counter++;//Если элемент массива=0, инкремент счетчика
```

Логические операции

Наряду с операциями отношения в предложениях if широко используются *логические операции*. Объединяя их с операциями отношения, можно создавать комбинированные конструкции проверки данных. Всего имеется 3 логические операции (табл. 3.7).

Таблица 3.7. Логические операции

Операция	*Значение*
!	НЕ
&&	И
||	ИЛИ

Приведем несколько формальных примеров действия операций отношения и логических в предложениях if.

```
if ((Func1()==0)||(Func2()==0)){//Если хотя бы одна из функций вернула 0, то
  Message();//вызвать функцию Message()
  return 0;//и завершить выполнение
  }
...//В противном случае продолжить выполнение программы
```

В приведенном примере выражение условия в предложении if совмещено с реализацией программного алгоритма. "Внутри" выражения условия последовательно вызываются функции Func1() и Func2(), выполняющие заданные действия. Если хотя бы одна из этих функций возвращает нулевое значение, что говорит о каких-то неполадках при ее выполнении, то выполняется следующий далее блок из двух предложений. В этом блоке сначала вызывается функция Message(), выводящая аварийное сообщение, а затем оператором return программа завершается с кодом возврата 0. Если же обе функции, и Func1(), и Func2(), выполнились успешно, то блок if пропускается и продолжается выполнение программы.

Заметим, что с учетом приведенных выше правил определения истинности предложение if последнего примера можно записать более изящно:

```
if (!Func1()||!Func2()){...
```

Действительно, если некоторая функция Func() возвращает нулевое значение, тогда выражение Func() обозначает нуль, т. е. FALSE, а выражение !Func() – не нуль, т. е. TRUE, что и приведет к выполнению условного блока предложения if.

Другой пример использования условного предложения:

```
int results[500];//Массив для хранения результатов
int index=0;//Индекс в массиве результатов
if ((var>0) && (Func(var)!=0)){//Если var положительна и Func() вернула не 0,
  results[index]=var;//сохраним var в массиве
  index++;//и переместимся к следующему элементу массива
  }
```

Конструкция if...else

Конструкция if else имеет следующий формат:

```
if(выражение){
  //Блок из одного или нескольких предложений C++
  }
else{
  //Блок из одного или нескольких предложений C++
  }
```

Предложение if позволяет только выполнить или не выполнить некоторый блок предложений; конструкция if...else дает возможность при удовлетворении условия выполнить один блок, а при неудовлетворении – другой. Так же как и в случае if, если один или другой блок состоит лишь из одного оператора, фигурные скобки можно не использовать.

```
if (id==1)
  Process1();//Если id=1, вызвать функцию Process1()
else
  Process0();//В противном случае вызвать функцию Process0()
Func();//После этого в любом случае вызвать функцию Func()
```

В приведенном примере если анализируемая переменная id равна единице, вызывается функция Process1(); если переменная id не равна единице, то вызывается функция Process0(). В любом случае после завершения той или другой из этих функций программа продолжается вызовом функции Func().

Конструкция switch...case

Конструкция switch...case позволяет в зависимости от значения некоторого выражения выбрать один из многих вариантов продолжения программы. Эта относительно сложная конструкция имеет следующий формат:

```
switch(выражение){
  case(константное_выражение_1):
    //Блок из одного или нескольких предложений C++
    break;
  case(константное_выражение_2):
    //Блок из одного или нескольких предложений C++
    break;
  ...//следующие операторы case
  default:
    //Блок из одного или нескольких предложений C++
  }
```

В качестве выражения при ключевом слове switch обычно используется переменная типа int или char, хотя можно использовать и более сложные выражения, в которые входят, например, арифметические или логические операции над несколькими переменными и константами.

В качестве константных выражений при ключевом слове case обычно используются просто константы (в числовой форме или в символьной, если они были заранее определены с помощью директивы препроцессора #define), однако могут использоваться и выражения над константами.

Выполнение конструкции switch начинается с вычисления выражения в скобках, которое должно давать целочисленный результат. Этот результат последовательно сравнивается с результатами вычисления константных выражений при ключевых словах case, и, если будет обнаружено равенство результатов, выполняется блок соответствующего case. Если совпадения результатов не обнаружено, выполняется блок при операторе default (по умолчанию), если же оператор default отсутствует, то начинают выполняться предложения, следующие за всей конструкцией switch...case.

Оператор switch, обнаружив равенство результатов, передает управление на соответствующий блок case и считает свою задачу на этом выполненной. Если не принять специальных мер, то после выполнения "правильного" блока case начнут по порядку выполняться все остальные, включая и блок при default. Чтобы избежать этого, в конце каждого блока case обычно ставят оператор break, разрывающий процесс выполнения конструкции switch и передающий управление на первое предложение за ее пределами. Если конструкция switch-case завершает выполнение текущей функции, то вместо оператора break используется оператор return.

Проиллюстрируем составление конструкции switch...case...break, считая, что в переменную id поступает идентификатор нажатой пользователем кнопки, причем специальной обработке подлежат только 3 кнопки с идентификаторами 101, 102 и 103.

```
switch(id){//Идентификатор нажатой кнопки
  case 101:{ //Если нажата кнопка 1, то
     ...//выполнить этот блок
     break; //и выйти из switch
     }
  case 102:{ //Если нажата кнопка 2, то
     ...//выполнить этот блок
     break; //и выйти из switch
     }
  case 103:{ //Если нажата кнопка 3, то
      ... //выполнить этот блок
      break; //и выйти из switch
     }
  default: //Если нажата любая другая кнопка, то
      ... //выполнить этот блок
 }//Конец оператора switch
```

В приведенном примере блоки case расположены в порядке номеров кнопок, что облегчает чтение программы. В действительности порядок блоков case не имеет никакого значения. Точно так же не имеет значения место расположения оператора default со своим блоком (его можно было поместить, например, перед первым case), хотя размещение его в конце конструкции выглядит наиболее естественно.

Оператор goto

Оператор goto реализует *безусловный переход*, т. е. позволяет перейти в любую точку программы, как вперед по тексту программы, так и назад. Точка перехода обозначается с помощью метки, которая представляет собой произвольный идентификатор с двоеточием в конце.

Считается, что использование оператора goto снижает наглядность программы, поэтому им лучше не пользоваться, тем более что практически всегда конструкцию с goto можно заменить другими предложениями. Однако наглядность программы – понятие неабсолютное, и, если в каком-то месте программы конструкция с goto оказывается яснее других, ничего плохого в ее использовании нет.

Предложения цикла

Циклы служат для выполнения некоторого фрагмента программы несколько раз. В редких случаях фрагмент выполняется в каждом последовательном шаге цикла без изменений; чаще каждый шаг цикла несколько отличается от предыдущего. Цикл может выполняться заданное заранее число шагов, а может завершаться при наступлении некоторого условия. Циклически повторяющийся фрагмент программы (его называют телом цикла) может включать любое количество любых предложений языка, в том числе вызовов функций. В языке C++ для организации циклов служат 3 предложения: for, while и do. Рассмотрим кратко возможности, предоставляемые этими предложениями.

Предложение for

Это предложение позволяет выполнить цикл заданное число шагов. Цикл for имеет следующую структуру:

```
for(начальное_значение_счетчика; условие_выполнения; новое_значение_счетчика)
  {
    //Блок из одного или нескольких предложений C++;
  }
```

Типичный пример цикла for:

```
int test[1000000];//Объявляется массив целых чисел
```

```
for(int i=0;i<1000000;i++)//Цикл из 1000000 шагов для i от 0 до 999999
  test[i]=i;//Элементам массива присваиваются значения i
```

Начальное значение счетчика цикла вычисляется один раз при входе в цикл. Обычно это операция присваивания вроде i=0. Как уже говорилось, C++ требует обязательного объявления всех используемых переменных; поскольку в нашем примере переменная i, служащая счетчиком цикла, ранее не встречалась, она прямо в предложении for и объявляется и инициализируется. Необходимо подчеркнуть, что объявленная таким образом переменная будет *локальной* для этого цикла; после выхода из цикла эта переменная перестает существовать и обращаться к ней бесполезно.

Условие выполнения представляет собой операцию отношения, например i<1000000. Пока указанное условие выполняется, цикл продолжается.

Новое значение счетчика вычисляется в каждом очередном шаге цикла. Чаще всего (хотя совсем не обязательно) здесь используется операция инкремента i++.

Если тело цикла состоит лишь из одного предложения, то фигурные скобки, ограничивающие цикл, можно опустить (см. приведенный выше фрагмент). Если же цикл состоит из нескольких предложений, то использование фигурных скобок обязательно (см. приведенный выше формат предложения for), иначе компилятор включит в цикл только первое из этих предложений, а остальные останутся за пределами цикла.

Типичное использование цикла – обработка элементов массива. При этом, изменяя условия выполнения цикла, можно отобрать для обработки нужные элементы массива: только четные, каждый пятый, ряд элементов из середины массива и т. д. Приведем несколько вариантов предложений for. Будем считать, что переменная, используемая в качестве счетчика цикла, уже объявлена.

Счетчик цикла может быть инициализирован любым числом и получать любые приращения, не обязательно 1:

```
for(n=100; n<200; n+=2)//50 шагов, n изменяется от 100 до 198 через 2
```

Счетчик цикла может работать в обратном направлении:

```
for(j=50; j>=0; j--)//51 шаг, j изменяется от 50 до 0 через 1
```

В условии выполнения может анализироваться не счетчик цикла, а какая-то другая переменная:

```
for(k=0; Abcd[k]!=0; k++)/*Просмотр элементов массива Abcd. Цикл завершается,
                           как только встретится элемент массива, равный 0*/
```

Вложенные циклы

Если в тело цикла входит другой цикл, то такой цикл называется вложенным. Вложенные циклы используются, в частности, для обработки двумерных или многомерных массивов.

Предположим, что в программе для анализа и прогнозирования эффективности предприятия требуется иметь таблицу выработки трех производственных участков по годам (рис. 3.17).

	1997	1998	1999	2000	2001
Участок 1	105	110	112	115	116
Участок 2	130	140	142	160	155
Участок 3	125	130	122	135	133

Рис. 3.17. Таблица выработки участков предприятия по годам

Эту таблицу естественно представить в программе в виде массива с размерностью 3x5:

```
int Yield[3][5];
```

Пусть в программе требуется, в частности, найти максимальное значение выработки. Поскольку массив данных двумерный, для его обработки естественно использовать конструкцию вложенного цикла:

```
int MaxYield=0;//Переменная для результата
for(int i=0;i<3;i++)//Внешний цикл по столбцам
  for(int j=0;j<5;j++)//Вложенный цикл по строкам
    if(MaxYield<Yield[i][j])//Поиск элемента
      MaxYield=Yield[i][j];//с максимальным значением
```

Обратите внимание на отсутствие фигурных скобок, ограничивающих тело цикла. Во внутреннем цикле имеется единственное предложение if()..., и заключать его в скобки нет необходимости. Во внешний цикл также входит единственное предложение for(...)..., так как условное предложение, составляющее тело вложенного цикла, рассматривается компилятором как составная часть предложения for. Для наглядности можно было добавить фигурные скобки, ограничивающие тело каждого цикла:

```
for(int i=0;i<3;i++){
  for(int j=0;j<5;j++){
    if(MaxYield<Yield[i][j])
      MaxYield=Yield[i][j];
    }//Конец вложенного for
  }//Конец внешнего for
```

В рассмотренном варианте алгоритма поиска мы узнаем максимальное значение выработки, однако останется неизвестным, к какому участку и к какому году она относится. Для сохранения индексов наиболее благополучного участка и года алгоритм придется усложнить:

```
int MaxYield=0;//Переменная для результата
int MaxBranch=0, MaxYear=0;//Переменные для результата поиска
for(int i=0;i<3;i++)//Внешний цикл по столбцам
  for(int j=0;j<5;j++)//Вложенный цикл по строкам
    if(MaxYield<Yield[i][j]){//Если очередной элемент > предыдущих,
      MaxYield=Yield[i][j];//запомнить его индекс,
      MaxBranch=i;//запомнить номер участка,
      MaxYear=j;//запомнить номер года
      }//Конец предложения if
```

Поскольку здесь при выполнении условия MaxYield<Yield[i][j] надо выполнить не одно, а 3 действия, они должны быть заключены в фигурные скобки с образованием блока предложений.

Предложение while

Если предложение for позволяет повторять цикл заданное число шагов, то предложение while служит для повторения цикла все время, пока выполняется заданное условие. Цикл с предложением while имеет следующую структуру (если блок предложений состоит из одного предложения, фигурные скобки можно опустить):

```
while(выражение)
  {
    //Блок из одного или нескольких предложений C++;
  }
```

Блок предложений в фигурных скобках будет выполняться многократно, пока выражение условия при ключевом слове while остается истинным. Как только выражение станет ложным, управление передается на предложение, следующее за циклом while. Оче-

видно, что тело цикла должно как-то влиять на выражение, определяющее условие выполнения, и в конце концов сделать его ложным, иначе цикл никогда не завершится.

В отличие от цикла for, с помощью которого обычно обрабатываются массивы с известным числом элементов, цикл while удобно использовать в тех случаях, когда размер массива или строки символов неизвестен. Пусть, например, указатель Args указывает на строку символов, в которой мы хотим обнаружить разделительный символ "/" и определить его индекс.

```
char* Args;//Указатель на анализируемую строку
int i=0;//Текущий индекс
int Index;//Номер найденного символа
while(Args[i]!='/')//Продолжать, пока символ не равен "/"
  i++;//Инкремент текущего индекса
Index=i;//Index = номеру найденного символа"/"
```

Здесь в тело цикла входит единственное предложение i++, т. е. в цикле нет ни одного "содержательного" предложения. Действительно, нам ничего не надо делать с символами строки, а только перебирать их до нахождения требуемого. С другой стороны, не следует забывать о том, что предложение while (в отличие от предложения for) не наращивает автоматически значение индекса. Поэтому инкремент индекса, если он используется, необходимо выполнять явным образом.

Как только сравнение Arg[i] и '/' дает положительный результат, предложение i++ пропускается и управление передается за пределы цикла while, т. е. на предложение Index=i. Таким образом, в переменную Index заносится то значение индекса i, которое соответствует байту с искомым символом.

Используя операцию постинкремента, можно представить цикл while в более изящном виде:

```
while(Args[i++]!='/');//Продолжать, пока символ не равен "/"
Index=i-1;//Index = номер найденного символа"/"
```

Еще один пример использования цикла while:

```
while(TRUE)
  {
  //Этот блок будет выполняться вечно
  }
```

В этом примере предложение while использован для того, чтобы организовать бесконечное выполнение некоторого блока (он может быть сколь угодно велик и включать любые алгоритмы и любое количество вложенных функций). Такая конструкция иногда используется в приложениях, инициирующих несколько параллельно выполняемых процессов. Например, блок в предложении while может принимать, обрабатывать и выводить на экран или сохранять некоторые данные, в то время как другой процесс следит за текущим временем. Как только время достигает заданного значения, все приложение завершается вместе с "недовыполненным" циклом while.

Предложение do...while

Цикл, организованный с помощью пары ключевых слов do и while, отличается от цикла while тем, что проверка условия осуществляется не до выполнения тела цикла, а после. Этот цикл описывается следующим образом (если в блоке предложений только одно предложение, фигурные скобки можно опустить):

```
do
  {
    //Блок из одного или нескольких предложений C++;
  }while(выражение)
```

Если после выполнения очередного шага цикла выражение при ключевом слове while оказалось ложным, дальнейшее повторение цикла прекращается. Проверка условия после выполнения тела цикла приводит к тому, что тело цикла в такой конструкции будет выполнено не менее одного раза.

Приведем пример такого цикла.

```
char Bytes[1000];//Байтовый массив с некоторой информацией
char Selection[1000];//Массив-приемник
int i=0;//Индекс
do{
  Selection[i]=Bytes[i];//Переносим байты из Bytes в Selection
  }while(Bytes[i++]!='0');//До (и включая!) первый 0
```

В этом примере конструкция do...while позволяет не только просмотреть в цикле байты массива до обнаружения байта, равного нулю, но и включить этот байт в выполняемую операцию копирования.

Функции

Функцией называется самостоятельная единица программы, решающая конкретную задачу. Программа, вызывая функцию, может передать ей определенные аргументы; функция, обработав эти аргументы, может вернуть в вызывающую процедуру результат своего действия. Можно представить себе функцию, которая не принимает никаких аргументов и не возвращает никаких значений (например, функция, выводящая на экран терминала какое-либо предупреждение), однако чаще всего функции или принимают аргументы, или возвращают результат, или делают и то и другое.

Любое приложение Windows, написанное на языке C++, должно иметь главную функцию с именем WinMain, так как именно это имя характеризует точку входа в приложение после его загрузки в память. Главная функция для выполнения стандартных, предусмотренных языком действий может вызывать по ходу своей работы многочисленные библиотечные функции языка C++; для реализации системных возможностей Windows в программу включают вызовы функций Windows; наконец, программист обычно включает в текст программы собственные функции – подпрограммы.

Разработка и включение в программу прикладных функций преследуют в основном две цели. Во-первых, уменьшается объем программы за счет того, что каждая функция описывается в программе только один раз, а вызываться может многократно (возможно, с различными значениями параметров). Во-вторых, повышается наглядность программы, поскольку технические детали реализации конкретных действий "прячутся" в описание функций, и главная программа получает возможность оперировать не с этими деталями, а с более крупными программными модулями – функциями. Если, например, требуется частично изменить алгоритм программы, гораздо проще переставить местами вызовы функций, чем переносить с места на место крупные фрагменты исходного текста. Точно так же изменить алгоритм выполнения какой-либо комплексной операции проще, если программный фрагмент, описывающий данную операцию, локализован в отдельной функции, а не "размазан" по телу основной программы. Другими словами, использование функций повышает степень модульности, структурированности программы. Однако увлекаться этим не следует, так как при большом числе вызываемых функций, особенно если из них, в свою очередь, вызываются функции следующего уровня, читать программу становится труднее.

Использование в программе функции требует, вообще говоря, выполнения трех действий:

- объявления прототипа функции;
- определения самой функции, т. е. выполняемых ею действий;
- вызова этой функции в одной или нескольких точках программы.

Конкретное воплощение перечисленных действий может заметно различаться в зависимости от требуемых характеристик и стиля разработки приложения. Например, прототипы функций можно описать непосредственно в тексте основного программного модуля, а можно выделить в отдельные включаемые файлы. Может быть и другой вариант, когда главная функция и функции-подпрограммы описываются в разных исходных файлах и компилируются отдельно, а затем объединяются компоновщиком. Исходные тексты функций вообще могут не входить в состав приложения, если их заранее откомпилировать, а полученные объектные модули включить в вызываемые на этапе компоновки объектные библиотеки. Наконец, в современном программировании широко используется концепция библиотек динамического связывания (Dynamic Link Libraries, DLL). Функции, входящие в состав такой библиотеки, могут загружаться и подключаться к выполняемому приложению динамически, по ходу его выполнения. После того как функция выполнила свою задачу, она может быть выгружена из памяти.

Прототип, определение и вызов функции

Прототип функции

Прототип функции имеет следующий формат:

```
тип_возвращаемого_значения имя_функции(тип_аргумента_1,тип_аргумента_2,...);
```

Примеры прототипов:

```
int Center(int);//Функция требует 1 аргумент int и возвращает тоже int
void attention(void);//Функция не требует аргументов и ничего не возвращает
void screen(char*);//Функция требует 1 аргумент char* и ничего не возвращает
int length(char*);//Функция требует 1 аргумент char* и возвращает int
int Search(char*,char,int);//Функция требует 3 аргумента и возвращает int
```

Как видно из приведенных примеров, типы аргументов, передаваемых в функцию при ее вызове, и тип возвращаемого результата никак не связаны друг с другом; функция может принимать в качестве аргументов любое количество переменных (или констант) любых типов; если функция возвращает результат, он также может быть любого типа. Однако вернуть можно только одну переменную в качестве результата; если функция по своей сути вычисляет несколько величин, то в качестве возвращаемого значения может фигурировать лишь одна из них. Как в таком случае получить остальные, будет рассмотрено ниже.

В прототипе нет необходимости указывать конкретные имена аргументов (они игнорируются компилятором), но иногда можно встретиться с таким объявлением прототипа:

```
int NumberOfChars(char* buffer);
```

Эта (избыточная) запись напоминает нам, что в дальнейшем при определении функции NumberOfChars единственный аргумент типа указателя на символьную строку будет именоваться buffer, хотя в действительности его можно будет назвать как угодно.

Если определение функции стоит в начале текста программы, перед главной функцией, то ее прототип можно не объявлять; если же функция описывается в конце текста программы, после строк обращения к ней, прототип обязателен. Лучше, однако, строго следовать рекомендациям языка и объявлять прототип во всех случаях.

Заметим, что при вызове какой-либо функции список передаваемых ей аргументов помещается в круглые скобки:

```
Result=Func(x,y,5);//Вызов функции с передачей ей трех аргументов
```

Если функция не требует аргументов, скобки (пустые) все равно надо указывать:

```
Result=Func1();//Вызов функции без аргументов
```

Отсюда возник литературный прием: указание в тексте (книги, не программы) какого-либо имени с парой пустых скобок говорит о том, что это функция. Таким образом, встретившееся в тексте книги обозначение number скорее всего относится к переменной, а number() – к функции, причем совсем не обязательно не требующей параметров. Естественно, этот прием не является законом, но он удобен и используется, в частности, в настоящей книге.

Определение и вызов функции

В определении функции описываются выполняемые ею действия. Рассмотрим в качестве примера функцию (назовем ее Add()), которая складывает два целых числа и возвращает их сумму. Разумеется, в такой функции нет ни малейшего практического смысла, однако она позволит нам описать правила составления определений и вызова функций. Полный текст программы с прототипом, определением и вызовом функции Add() выглядит следующим образом:

```
/*Программа 3-5. Пример функции*/
#include <windows.h>
/*Прототип функции Add()*/
int Add(int,int);
/*Определение функции Add()*/
int Add(int a,int b){
  int sum;
  sum=a+b;
  return sum;
  }
/*Глобальные переменные*/
  int res;//Переменная для результата функции
  int x=0x11111111;//Первый аргумент
  int y=0x22222222;//Второй аргумент
/*Главная функция WinMain()*/
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
  char szText[100];
  res=Add(x,y);//Вызов функции Add с конкретными аргументами x и y
  wsprintf(szText,"res=%#x",res);
  MessageBox(NULL,szText,"Info",MB_OK);
  return 0;
  }
```

Прототип функции сложения двух чисел должен, очеввидно, выглядеть таким образом:

```
int Add(int, int);
```

Определение функции начинается с ее заголовка, фактически повторяющего прототип, за которым в фигурных скобках следует текст функции:

Отличие заголовка функции от прототипа заключается в том, что в круглых скобках после имени функции указывается не просто перечень типов передаваемых функции аргументов, но также обозначения (произвольные) локальных переменных, в которые будут поступать аргументы. Эти локальные переменные, обозначенные в нашем примере a и b, называются *параметрами* функции (иногда – формальными параметрами).

В тексте функции описываются действия над ее параметрами и при необходимости с помощью оператора return указывается возвращаемое функцией выражение. В нашем примере в функции объявляется еще одна локальная переменная – sum; в которую поступает сумма параметров и которая затем служит возвращаемым значением.

Как осуществляется выполнение функции? При ее вызове с указанием конкретных аргументов

```
res=Add(x,y);//Вызов функции Add с конкретными аргументами x и y
```

эти аргументы копируются в стек и, можно сказать, превращаются в параметры функции a и b. Там же в стеке компилятор выделяет место для локальной переменной sum. В процессе выполнения функции параметры a и b складываются, их сумма заносится в переменную sum, после чего осуществляется возврат из функции с передачей значения sum во внешнюю по отношению к функции переменную. Этот процесс можно наблюдать с помощью отладчика.

На рис. 3.18 приведен кадр отладчика, полученный при выполнении функции Add() непосредственно после сложения параметров a и b. На этом рисунке можно увидеть, во-первых, сами параметры, располагаемые в стеке (по адресам 0x68FD94 и 0x68FD98) и, во-вторых, команды процессора (по адресам 0x40110B и 0x40110E), которые и осуществляют операцию сложения.

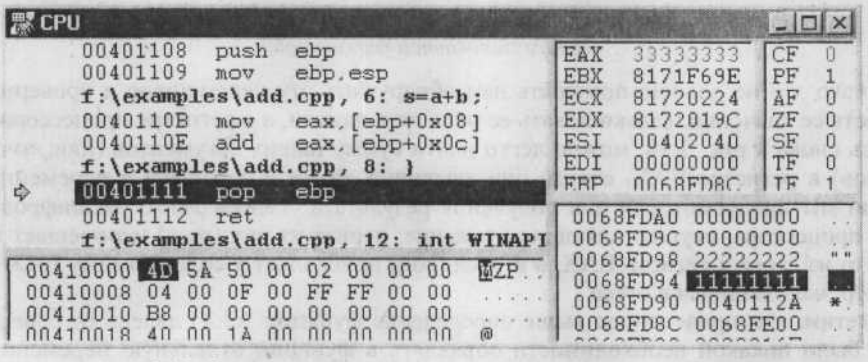


Рис. 3.18. Окно отладчика на момент выполнения функции Add()

После выхода из функции указатель стека возвращается в исходное состояние (см., напр., рис. 1.14), и, следовательно, вся область стека, выделенная под локальные переменные, считается снова свободной и будет использована заново при вызове программой следующих функций. Если попробовать определить значение a или b вне функции с помощью отладчика, мы получим сообщениео том, что она неопределена (рис. 3.19).

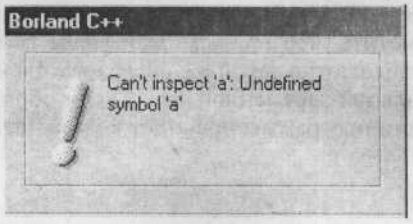


Рис. 3.19. Сообщение отладчика о невозможности инспектирования локальной переменной вне области ее существования

Таким образом, мы еще раз убеждаемся в том, что локальные переменные, объявленные в функции, действуют только пока она выполняется, а после завершения функции все они исчезают.

Реально дело может обстоять не совсем так, как это было описано. Современные компиляторы стремятся оптимизировать обрабатываемые ими программы, и если некоторая локальная переменная используется лишь на ограниченном участке программного кода, компилятор вполне может выделить ей место не в памяти, а в одном из регистров процессора. Понимание этого обстоятельства может помочь при отладке программ. При попытке получить с помощью отладчика значения такой переменной вы получите сообщение о невозможности ее инспектирования по причине ее оптимизации (рис. 3.20).

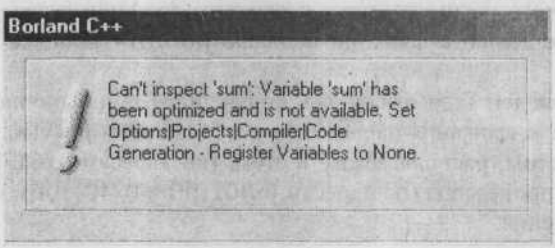


Рис. 3.20. Сообщение отладчика о невозможности инспектирования оптимизированной переменной

Однако это не должно помешать нам обнаружить эту переменную и проверить правильность ее значения, только искать ее надо не в памяти, а в регистре процессора. Обратившись снова к рис. 3.18, можно легко найти сумму наших аргументов (или, точнее параметров) в регистре EAX, откуда она, очевидно, будет возвращена в переменную res, которую мы предусмотрели для получения результата. Теперь легко расшифровать команды процессора, осуществляющие сложение: первая из них (mov) перемещает первый параметр из стека в регистр EAX, а вторая (add) прибавляет к содержимому EAX второй параметр, находящийся в стеке.

Заметим, что приведенное выше определение функции Add() далеко от совершенства. Не было никакой необходимости объявлять в функции отдельную переменную для получения суммы. Достаточно указать выражение для суммы в качестве аргумента оператора return:

```
int Add(int a, int b){
  return a+b;
  }
```

Следует только понимать, что реально сумма a+b где-то должна существовать, следовательно, компилятор все равно выделять для нее место, скорее всего, в том же регистре EAX. Такой промежуточный, невидимым нам результат иногда называют временной мишенью. Любопытно отметить, что приведенный выше "оптимизированный" вариант функции Add() приведет к абсолютно такой же последовательности кодов в памяти, что и наш первый вариант с локальной переменной sum. Таким образом, совсем не обязательно сокращение исходного текста программы приводит к уменьшению размеров исполняемого файла.

Интерфейс с функциями

Под интерфейсом с функциями понимаются способы передачи в них аргументов и получения из них результатов. В языке C++ предусмотрены 3 способа передачи аргументов в вызываемую функцию:

- по значению, когда в функцию передается числовое значение аргумента;
- по адресу, когда в функцию передается не значение аргумента, а его адрес, что особенно удобно для передачи в качестве аргументов массивов и структур;
- по ссылке, когда в функцию также передается адрес аргумента, однако действуют особые (и весьма удобные) правила использования этого адреса в вызываемой функции.

Передача в функцию аргументов по значению

Пусть для решения некоторой математической задачи требуется вычислять суммы натурального ряда чисел на произвольно заданных интервалах от числа a до числа b (не будем пытаться доказывать, что это действительно кому-то надо). Оформим вычисление искомой суммы в виде функции Interval(), для которой входными параметрами будут, очевидно, целые числа a и b, характеризующие границы заданного интервала, а возвращаемое значение должно представлять вычисленную сумму. Будем считать, что сумма не превысит $4*10^9$, и объявим тип возвращаемого значения UINT, т. е. unsigned int.

При двух целочисленных параметрах и возвращаемом значении типа UINT функция Interval() будет иметь такой прототип:

```
UINT Interval(int,int);
```

Определение функции может выглядеть следующим образом:

```
UINT Interval(int x,int y){//Заголовок функции Interval
  UINT temp=0;//Временная переменная для накопления суммы
  for(int i=x;i<=y;i++)//Цикл от числа x до числа y с шагом 1
    temp=temp+i;//Прибавление к текущему значению temp очередного числа
  return temp;//Возврат вычисленного значения
  }//Конец функции Interval
```

При вызове функции нам понадобятся две переменные типа int для передачи аргументов и одна типа UINT для получения из функции результата:

```
int a,b;//Объявление переменных-аргументов
UINT res;// Объявление переменной для результата
  a=1200;//Присвоим границам
  b=1300;//конкретные значения
  res=Interval(a,b);//Вызов функции interval и засылка результата в res
```

Функция Interval(), получив в качестве параметров значения нижней и верхней границ диапазона, выполняет в цикле сложение чисел от нижней границы a до верхней b. После выхода из цикла полученная сумма передается оператором return в вызывающую процедуру. Для заданных в примере границ 1200 и 1300 результат составляет 126250.

Для того чтобы формальным параметрам были правильно присвоены фактические значения, при вызове функции

```
sum=Interval(a,b);
```

в нее надо передавать ровно столько аргументов, сколько указано в ее заголовке (и прототипе), и в том же порядке. Компилятор присваивает формальным параметрам значения фактических аргументов по порядку: первому параметру передает значение первого аргумента, второму – второго и т. д. Имена фактических аргументов, как и имена параметров функции, не имеют никакого значения, важен только их порядок. В другом месте программы можно было вызвать ту же функцию Interval() следующим образом:

```
int n1=100,n2=200;
res=Interval(n1,n2);
```

(в предположении, что переменная res уже была объявлена ранее).

Выше уже говорилось о том, что все локальные переменные размещаются компилятором в стеке программы, при этом если речь идет о формальных параметрах функции, то они действуют только пока выполняется эта функция. При этом как бы функция не изменяла их значения, вызывающая процедура узнать об этом никак не может, потому что за пределами функции этих переменных просто нет. Как говорят, локальные переменные *не видны* в вызывающей процедуре; областью их видимости является функция, в которой они объявлены. Не видны они также и во вложенных функциях, вызываемых из той, где они объявлены.

Между прочим, из сказанного следует, что если в основной процедуре объявлена некоторая переменная, например,

```
int size=128;
```

и такая же по имени и типу переменная объявлена в вызываемой функции

```
int Function(int x,int y){
int size=256;
...
```

то эти две переменные никакого отношения друг к другу не имеют. Первой переменной size можно пользоваться только в основной процедуре; в функции Function() она не видна. Если мы хотим, чтобы функция получила значение этой переменной, ее надо передать в функцию через один из параметров:

```
Function(size,...);
```

Тогда локальная переменная x внутри функции будет по значению совпадать с переменной size, переданной в функцию при ее вызове.

Вторая переменная size, объявленная в функции, наоборот, не видна в вызывающей процедуре. Чтобы сделать ее значение известным вызывающей процедуре, его следует вернуть оператором return:

```
return size;
```

а функцию в этом случае вызывать так, чтобы можно было получить из нее результат:

```
int result=Function(...);
```

Таким образом, передача по значению позволяет функции получить любое количество значений параметров, однако передать назад в вызывающую процедуру функция может только одну-единственную величину. Для того чтобы влиять на большее количество переменных в вызывающей процедуре, функция должна использовать передачу параметров по адресу.

Передача в функцию аргументов по адресу

Передать в функцию можно адрес любой переменной – скалярной, массивной или структурной. Передача по адресу удобна прежде всего потому, что хотя функция не может модифицировать сам передаваемый параметр – адрес, но, обращаясь по этому адресу, она может модифицировать переменную, находящуюся в вызывающей процедуре. Таким образом снимается ограничение на возврат в вызывающую процедуру единственного результата работы функции. При передаче адреса скалярной переменной функция может изменить значение только этой переменной; при передаче адресов нескольких скалярных переменных функция может изменить значения их всех; при передаче адреса массива или структуры функция может "дотянуться" до всех элементов этих объектов.

Если скалярные переменные можно передать в функцию и по значению и по адресу, то массив можно передать только по адресу. Действительно, имя массива в C++ отожде-

ствляется с адресом первого элемента этого массива и передача в качестве параметра имени массива приводит к передаче не всего массива целиком, а только его адреса.

Рассмотрим простую программу, в которой функция, получив в качестве параметра адрес символьной строки, выполняет ее обработку, а именно просматривает строку и преобразует латинские строчные буквы в прописные.

```
/*Прототип функции Caps*/
void Caps(char*);
/*Определение функции caps*/
void Caps(char* s){//Заголовок функции Caps
  int i=0;//Локальная вспомогательная переменная для цикла
  while(s[i]!=0{//Выполнять, пока символ не 0
    if(s[i]>='a'&&s[i]<='z')//Если символ лежит в диапазоне от 'a' до 'z',
      s[i]=s[i]-0x20;//сделать его прописным, вычтя 0x20
    i++;//Увеличим индекс для перехода к следующему символу строки
    }//Конец цикла while
  }//Конец функции Caps
/*Фрагмент вызывающей процедуры с вызовом функции Caps*/
char fname []="D:\figures\fig25.bmp"//Подлежащая преобразованию строка текста
Caps(fname);//Вызов функции Caps с передачей адреса строки String
```

В основной процедуре объявлен символьный массив fname с произвольным текстом (в данном примере – с полной спецификацией некоторого файла). В этом тексте могут встречаться любые символы, в частности как строчные, так и прописные латинские буквы. После вызова функции Caps(fname) строчные латинские буквы будут преобразованы в прописные и строка fname примет следующий вид:

```
D:\FIGURES\FIG25.BMP
```

Как уже говорилось, указатель, как и имя массива, можно использовать в индексных операциях. Таким образом, поскольку параметр s, в который поступает имя, т. е. адрес преобразуемой строки, является указателем, допустимо выражение s[i], которое обозначает i-й элемент строки. Следует обратить внимание на то, что анализ и модификация символа выполняется непосредственно в памяти *вызывающей* процедуры.

Выше уже отмечалось, что в качестве аргументов функции могут фигурировать адреса любых переменных – скалярных, массивных и структурных. Пусть, например, нам нужна функция, которая, получив адрес некторой переменной с плавающей точкой, заменит эту переменную на десятичный логарифм от ее значения. Поскольку эта функция будет обращатья к аргументу через передаваемый ей адрес, возвращать ей нечего и ее прототип будет таким:

```
void Log(float*);
```

Определение функции, если пренебречь необходимыми проверками (аргумент логарифма не может быть отрицательным числом или нулем), будет состоять из одного предложения:

```
void Log(float* px){
  *px=log10(*px);//Снятие ссылки, т. е. обращение к переменной по ее адресу
  }
```

При вызове функции мы должны передать ей *адрес* скалярной переменной:

```
float val;//Изменяемая переменная-аргумент функции
val=12675.56;//Присваиваем val некоторое значение
Log(&val);//Вызов функции - в val теперь log(val)
```

При заданном в примере значении аргумента val он получит значение 4.102967.

Передача в функцию аргументов по ссылке

Передача в функцию параметра по ссылке фактически обозначает ту же передачу адреса параметра, однако делается это специфическим образом. На рис. 3.21 приведен для сравнения синтаксис строк программы, реализующих оба метода передачи параметра.

Передача по адресу	*Передача по ссылке*
Объявление переменной-параметра в вызывающей процедуре	
`int x;`	`int x;`
Прототип функции	
`void func(int*);`	`void func(int&);`
Определение функции	

`void func(int* arg){`
`  *arg=1234;//Снятие ссылки`
`  }`
	`void func(int& arg){`
	`  arg=1234;//Обращение к самой`
	`  }  //переменной`

Вызов функции в вызывающей процедуре	
`func(&x);`	`func(x);`
Результат действия функции	
`x=1234`	`x=1234`

Рис. 3.21. Сравнение методов передачи параметров по адресу и по ссылке

При передаче по ссылке, во-первых, обращение к переданному параметру внутри функции осуществляется не с помощью операции снятия ссылки, а непосредственно и, во-вторых, при вызове функции в качестве ее аргумента указывается не адрес передаваемой переменной, а сама переменная. Другими словами, и при вызове функции, и в ней самой мы как бы работаем с *самой переменной* из вызывающей процедуры, забыв о том, что в действительности параметром является ее адрес.

Возврат из функции результата ее работы

Функция может вернуть в вызывающую процедуру результат любого требуемого типа – целое, с плавающей точкой, символ, указатель и т. д. При этом допустим возврат структурной переменной, но не массива. Возвращаемое значение указывается в качестве аргумента оператора return, которым должно заканчиваться выполнение функции. Обычно оператор return стоит в конце текста функции, хотя это совсем не обязательно. Более того, функция может иметь несколько операторов return, если внутри функции предусмотрен анализ каких-то условий и выполнение условных переходов на тот или иной return. Например, часто для возврата из функции используется конструкция вроде следующей:

```
if(val>=0)
  return val;//Нормальное завершение функции с возвратом искомого результата
else
  return -1;//Возврат -1 говорит о той или иной ошибке при выполнении функции
```

Поскольку в языке C++ можно работать только с заранее объявленными переменными, то переменная, в которую возвращается результат работы функции, должна существовать, т. е. быть заранее объявлена в вызывающей процедуре:

```
int result;//Объявление переменной в вызывающей процедуре
result=func();//Вызов функции func и получение из нее результата в result
```

Функция может вернуть адрес переменной, однако это не должна быть локальная переменная, определенная внутри функции, поскольку все такие переменные исчезают по-

сле завершения вызванной функции. Нельзя, например, написать текст функции следующим образом:

```
char* function(){
  char str[]="Внимание!";//Символьная строка, объявленная внутри функции
  return str;//Возврат адреса этой строки - недопустимо!
  }
```

Однако имеет право на существование (и часто используется) такая конструкция:

```
char* func(char* dest, char* src){
  ...//Действия над строками src и dest
  return dest;
```

Здесь параметры-адреса получают свои значения извне функции; следовательно, они существуют и любой из них можно вернуть в качестве возвращаемого значения.

Синтаксические правила языка C++ позволяют писать весьма компактные (с точки зрения объема исходного текста) программы. Этому, в частности, способствует допустимость включения вызова функции в логические или арифметические выражения, а также в оператор return. Во всех таких случаях в соответствующее выражение подставляется результат, возвращаемый функцией после ее выполнения (откуда следует, что в выражениях можно использовать только те функции, которые возвращают результат). Например, имеет право на существование предложение вроде следующего:

```
if(!Func1())
  UINT res=Func2() | Func3();
```

Здесь прежде всего вызывается функция Func1(), которая может выполнять сколь угодно сложную работу. Если она завершилась успешно, о чем свидетельствует нулевое возвращаемое значение, выполняется последовательный вызов функций Func2() и Func3() (возможно, с какими-то аргументами) и побитовая сумма возвращаемых ими значений присваивается переменной res.

Функции и структуры

Как уже отмечалось, передача в функцию массива допускается только по адресу. Передать же в функцию структуру можно и по значению и по адресу. При передаче по значению в качестве передаваемого параметра указывается имя структуры, а обращение внутри функции к членам структуры осуществляется с помощью оператора "точка" (.). При передаче по адресу в качестве передаваемого параметра указывается адрес структуры, а обращение к членам структуры осуществляется с помощью оператора ->.

Первый способ менее эффективен, так как при передаче по значению внутри функции создается полная локальная копия передаваемой структурной переменной, с которой функция затем и работает. Если структура велика по объему, такое копирование приводит к значительному расходу памяти и процессорного времени. Поскольку локальные переменные теряются после завершения функции, результат обработки локальной копии следует вернуть в вызывающую процедуру. Возврат структуры по значению требует еще одного копирования локальной копии в переменную, принимающую результат.

При передаче адреса структуры не происходит ни выделения памяти под структуру, ни ее копирования. Внутри функции память расходуется только для хранения адреса структуры. Функция с помощью адреса структуры может обращаться непосредственно к ее членам в вызывающей процедуре, поэтому возвращать структуру нет необходимости (хотя, разумеется, можно вернуть, например, значения ее отдельных членов или результат каких-либо манипуляций с ними). Приведем примеры передачи в функцию структуры тем и другим способом.

Пусть у нас определена структура с именем Men, в которой имеются два члена – символьная строка с именем человека и целое данное с его возрастом. Предусмотрим функции Set() и SetPtr(), выполняющие одинаковые операции – инициализацию элементов структуры задаваемыми значениями. Различие функций Set() и SetPtr() будет заключаться в том, что в первую функцию аргумент (структурная переменная) будет передаваться по значению, а во вторую – по адресу. Ниже приведен соответствующий программный фрагмент:

```
/*Определим структуру типа Men*/
struct Men{
  char* name;
  int age;
  };
/*Прототипы функций инициализации*/
Men Set(Men, char*, int);//Передача и возврат структуры Men по значению
void SetPtr(Men*, char*, int);//Передача адреса структуры Men
/*Определим функции Set() и SetPtr()*/
Men Set(Men m, char* n, int a){//Получение в функцию всей структуры типа Men
  m.name=n;//Инициализация имени
  m.age=a;;//Инициализация возраста
  return m;//Возврат всей локальной структуры m по значению
  }
void SetPtr(Men* pm, char* n, int a){//Получение в функцию адреса структуры
  pm->name=n;//Инициализация имени
  pm->age=a;; //Инициализация возраста
  }//Возвращать ничего не требуется
/*Главная функция WinMain():*/
/*Создадим две переменные типа men*/
  Men man001;//Объявление структурных переменных типа Men
  Men man002;
/*Вызовем функции инициализации*/
man001=Set(man001, "Дроздов", 22);
SetPtr(&man002, "Канарейкин", 44);
```

Основы аппарата макросов

В состав современных систем программирования наряду с компилятором входит еще дополнительное средство, называемое препроцессором. Исходный текст программы обрабатывается сначала препроцессором и лишь после этого поступает на вход компилятора. Задачей препроцессора является поиск в исходном тексте программы директив препроцессора и изменение текста программы в соответствии с этими директивами. Все директивы препроцессора начинаются со знака #. В примерах этой книги мы столкнемся с двумя директивами: #include – для включения в исходный текст программы дополнительных файлов и #define – для определения макросов.

Макросы, или макроопределения, – это конструкции, создаваемые с помощью директивы препроцессора #define. В простейшем случае макрос представляет собой просто определение константы или новое символическое обозначение для уже имеющегося ключевого слова:

```
#define FILE_SIZE 1000//Вводится константа FILE_SIZE со значением 1000
#define FALSE 0//Вводится константа FALSE, равная 0
#define VOID void//Вводится новое обозначение VOID, равносильное void
#define PPVOID void**//Вводится новое обозначение PPVOID, равносильное void**
```

Первое из приведенных предложений определяет пользовательскую константу FILE_SIZE. Три других предложения взяты из системных файлов среды программирования Borland C++.

Как уже отмечалось, обрабатывает макросы препроцессор. Обработка простейших макросов заключается в подстановке в программу значений описанных в макросе символических обозначений (или просто символов, как их часто называют, буквально, хотя и неправильно, используя английский термин). Так, всюду в программе обозначение FILE_SIZE будет заменено числом 1000, а обозначение VOID – типом void.

Более типичны макросы с аргументами. В процессе обработки макровызова (в процессе макрорасширения) препроцессор подставляет фактические аргументы вместо формальных параметров макроса. Рассмотрим, например, задачу выделения из 2-байтового слова старшего и младшего байтов. Выделить младший байт просто: достаточно приравнять наше слово байтовой переменной. В нее, естественно, поместится только младший байт 2-байтового слова. Для выделения же старшего байта слово надо сдвинуть вправо на 8 разрядов. Если к этой операции приходится обращаться многократно, ее разумно оформить в виде макроса (в действительности такой макрос уже имеется в файлах среды программирования):

```
#define HIBYTE(w) w>>8
/*Вызывающая процедура*/
UINT value=0x3456;
BYTE hi=HIBYTE(value);//hi получает значение старшего байта value, т. е. 0x34
  }
```

Оператор >>, входящий в макрос, сдвигает содержимое слова вправо на указанное число битов, т. е. переносит его старший байт на место младшего.

В приведенном макросе имеются два недостатка. Первый заключается в том, что в предложении вызова макроса будет происходить неявное преобразование типа. Действительно, в этом предложении происходит присвоение 1-байтовой переменной hi 2-байтового значения value. Ведь операция побитового сдвига вправо на 8 разрядов только сдвинет все биты переменной value, но не изменит ее размер. В подобных случаях компилятор выполняет автоматическое преобразование типов переменных, конкретно – тип WORD преобразуется в тип BYTE, т. е. unsigned char. В приведенном выше примере такое неявное преобразование по умолчанию будет работать правильно, однако надежнее выполнять преобразование типов в явной форме (такое преобразование называется приведением типа), хотя бы для того, чтобы подчеркнуть, что присвоение слова байту произошло не в силу ошибки программиста, а вполне умышленно. Для выполнения приведения типа в C++ имеется специальная конструкция вида

```
(назначаемый_тип) преобразуемое_выражение
```

Для нашего примера использование этой конструкции приведет к следующему усовершенствованию макроса:

```
#define HIBYTE(w) (BYTE)(w>>8)
```

Второй недостаток серьезнее. Если в качестве параметра нашего макроса используется не просто переменная, а некоторое выражение, то макрорасширение может дать совершенно неправильные результаты. Пусть мы хотим получить старший байт не самой переменной value, а выражения value+0x300. Очевидно, что эта сумма равна 0x3756 и старший байт суммы составляет 0x37.

Если аргумент макроса записан в виде value+0x300, то все будет правильно. Если, однако, нам надо не только получить в переменную hi старший байт аргумента макроса,

но и сохранить для value новое, увеличенное значение, то аргумент макроса удобно записать в форме

```
value+=0x300
```

Это сокращенная форма предложения, в котором переменной value присваивается новое, увеличенное на 0x300, значение:

```
value=value+0x300;
```

Такого рода сокращения типичны для языка C++, где имеется много возможностей сэкономить чернила за счет снижения наглядности. Выражение

```
BYTE hi=HIBYTE(value+=0x300);
```

не только даст совершенно неправильный результат 0x59, но и value приобретет странное значение 0x3459. Все эти неприятности происходят из-за неудачной формы макроса и пренебрежения нами приоритетом операций. Поскольку препроцессор никак не анализирует расширение макроса, а просто механически подставляет в него аргумент, макрорасширение с аргументом value+=0x300 будет иметь следующий вид:

```
value+=0x300>>8
```

В каком порядке будут выполняться операции сложения и сдвига? Составные операции типа +=, *= и др. имеют очень низкий приоритет и выполнятся в последнюю очередь. У операции >> приоритет выше. Поэтому сначала будет выполнен сдвиг 0x300>>8, который даст значение 0x0003, а затем этот результат будет прибавлен к value с получением 0x3459. В байтовую переменную hi пойдет младший байт этого числа, т. е. 0x59, а значение value останется равным 0x3459.

Почему эта ошибка не проявляется в случае аргумента value+0x300? Потому что знак плюс имеет более высокий приоритет, чем операции сдвига, и сначала выполняется прибавление к value числа 0x300 и лишь затем сдвиг результата.

Для избежания всех этих неприятностей макрос необходимо описать следующим образом:

```
#define HIBYTE(w) (BYTE) ((w)>>8)
```

а еще лучше, имея в виду возможное использование его в сложных выражениях,

```
#define HIBYTE(w) ((BYTE) ((w)>>8))
```

Тогда макрорасширение для случая аргумента с операцией составного присваивания примет такой вид:

```
((value+=0x300)>>8)
```

и независимо от старшинства операций сначала выполнится выражение в скобках и лишь затем произойдет сдвиг результата (при таком относительно простом выражении внешние скобки значения не имеют). В результате value примет значение 0x3756, а переменная-приемник (выше она была обозначена hi) – 0x37.

Макросы для выделения байтов из 2-байтового слова включены во вспомогательный файл среды программирования WINDEF.H и широко используются в приложениях Windows. Теперь читателю будет понятна их форма

```
#define LOBYTE(w) ((BYTE)((w))
#define HIBYTE(w) ((BYTE)(((WORD)(w)>>8) & 0xFF))
```

Мы видим, что для повышения надежности и универсальности макрос HIBYTE предусматривает дополнительные преобразования. Сначала аргумент w приводится к типу слова без знака (WORD), затем это слово сдвигается вправо на 8 бит (>>8), после чего из

него оператором &, обозначающим побитовую операцию И, выделяется младший байт и, наконец, результат приводится к типу байта без знака (BYTE).

Макросы часто используются для переименования имен функций и уточнения типов возвращаемых ими значений. Например, в Windows имеется функция GetStockObject(), позволяющая получить со "склада" Windows некоторые хранящиеся там графические объекты: перья, кисти и шрифты. Для описания графического объекта вообще в Windows используется обобщенный тип HGDIOBJ (от Handle to GDI Object, дескриптор объекта интерфейса графических устройств). Функция GetStockObject(), как видно из ее прототипа, как раз и возвращает такой обобщенный дескриптор объекта, если ей передать в качестве параметра символическое обозначение требуемого объекта:

```
HGDIOBJ GetStockObject(int nObject);
```

Однако практически функция GetStockObject() применяется в конструкциях, где от нее требуется получить дескриптор конкретного объекта, например типа HPEN для перьев, HBRUSH для кистей или HFONT для шрифтов. Поэтому при ее использовании приходится прибегать к явному приведению типа:

```
HBRUSH hNewBrush=(HBRUSH)GetStockObject(BLACK_BRUSH);
```

Здесь возвращаемый функцией GetStockObject() дескриптор черной кисти сначала преобразуется в тип HBRUSH и лишь затем присваивается программной переменной hNewBrush, где он будет храниться с целью дальнейшего использования. При необходимости получить со склада дескриптор пера возвращаемое функцией значение следует преобразовать в тип HPEN и т. д.

Для облегчения программирования в файл WINDOWSX.H включено несколько макросов, содержащих в себе описанное выше преобразование:

```
#define GetStockPen(i)    ((HPEN)GetStockObject(i))
#define GetStockBrush(i)  ((HBRUSH)GetStockObject(i))
#define GetStockFont(i)   ((HFONT)GetStockObject(i))
```

Подключив файл WINDOWSX.H к исходному тексту программы (оператором #include, о котором пойдет речь в следующем разделе), мы может использовать вместо обобщенной функции GetStockObject() более наглядные макросы GetStockPen(), GetStockBrush() и GetStockFont(), которые не требуют приведения типов.

Между прочим, описанная методика приводит к неожиданностям при поиске требуемых программисту функций в интерактивном справочнике среды разработки. Увидев в каком-нибудь программном примере "функцию" GetStockPen() и желая уточнить правила работы с ней, вы обращаетесь к интерактивному справочнику и не находите ее, потому что это не функция, а макрос. Расшифровку этого макроса придется искать в одном из включаемых файлов среды разработки, причем неизвестно, в каком именно. Конкретно этот макрос, как уже говорилось, определен в файле WINDOWSX.H, однако другие интересующие вас макросы могут оказаться, например, в файлах WINUSER.H, WINBASE.H, COMMCTRL.H или каких-то еще.

Вообще с макросами при разработке Windows-приложений приходится сталкиваться очень часто и следует приобрести навыки их расшифровки.

Ключевые слова #include и typedef

Директива препроцессора #include и включаемые файлы

Мы упомянули в предыдущем разделе, что макросы LOBYTE и HIBYTE определены в файле WINDEF.H. Для того чтобы эти макросы можно было использовать в прикладной программе, файл WINDEF.H должен быть подсоединен к исходному тексту программы. Это подсоединение осуществляется с помощью директивы препроцессора #include.

Взглянем на исходный текст программы 2-1. В нем имеется предложение с директивой #include:

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

С ее помощью в исходный текст программы включается содержимое указанного в ней файла (в результате чего размер исходного текста может увеличиться во много раз). Чаще всего в качестве включаемых используют заголовочные файлы, или файлы заголовков, имеющие обычно расширение .h или .hpp (от слов header – заголовок или header plus plus – заголовок для языка C++). В нашем примере включаемый заголовочный файл является системным (в том смысле, что он входит в систему программирования); в дальнейшем мы будем использовать и собственные заголовочные файлы. Поскольку заголовочные файлы подсоединяются к исходному тексту программы, очевидно, что они должны представлять собой текстовые файлы, состоящие из предложений, понимаемых компилятором языка программирования.

В заголовочных файлах хранится главным образом информация следующего вида:

- прототипы функций;
- определения констант, используемых функциями Windows;
- определения новых типов данных;
- макросы разного рода.

Выше мы рассматривали понятие прототипа функции. Для *всех* вызываемых функций в программу должны быть включены их прототипы. При этом, как уже отмечалось, функции можно разбить на 3 категории: стандартные функции языка C++; функции Windows; функции, написанные прикладным программистом.

Прототипы функций Windows содержатся в многочисленных заголовочных файлах, входящих в любую среду разработки приложений (Borland C++, Microsoft Visual C++ и др.). Часто достаточно в явной форме подсоединить единственный файл WINDOWS.H; ссылки на остальные файлы заголовков могут содержаться в этом файле. Так, в файле WINDOWS.H, используемом в 32-разрядных программах (в действительности в пакет среды программирования входят несколько файлов с таким именем), можно найти следующий фрагмент:

```
#include <windef.h>
#include <winbase.h>
#include <wingdi.h>
#include <winuser.h>
#include <winnls.h>
#include <wincon.h>
#include <winver.h>
#include <winreg.h>
#include <winnetwk.h>
```

Таким образом, через файл WINDOWS.H мы получаем доступ к содержимому целого ряда заголовочных файлов Windows и, соответственно, к прототипам функций Windows. Например, в файле WINUSER.H содержатся прототипы функций общего назначения; в файле WINGDI.H описаны прототипы всех функций, связанных с отображением на экране (вывод текста, рисование фигур, задание цвета объектов и пр.); в файле WINBASE.H имеются прототипы функций обслуживания ресурсов Windows, и в частности прототип главной функции Windows WinMain().

Иначе обстоит дело с прототипами функций C++. Они определены в многочисленных заголовочных файлах C++, и для использования в программе какой-либо функции необходимо подсоединить к программе файл с ее прототипом. Узнать, в каком файле описан прототип той или иной функции C++, можно с помощью интерактивного справочника. Оттуда мы узнаем, например, что прототип функции memset() входит наряду с

другими функциями обслуживания памяти в файл MEM.H; прототипы ряда математических функций – в файл MATH.H; прототипы функций обслуживания строк – в файл STRING.H и т. д.

В заголовочных файлах Windows, помимо прототипов, содержится масса определений символических констант. Так, константа LTGRAY_BRUSH (равная 1), характеризующая цвет окна приложения Windows, описана в файле WINGDI.H; константа IDC_ARROW (равная 39512), задающая форму курсора мыши в виде стрелки, описана в файле WINUSER.H; в тот же файл входит определение константы SW_SHOWNORMAL, которая устанавливает режим отображения главного окна приложения (нормальный, т. е. тех размеров, которые заданы в программе). Эти заголовочные файлы автоматически подключаются к исходному тексту программы, если в него включен файл WINDOWS.H.

Наконец, в заголовочных файлах Windows содержатся макроопределения многочисленных макросов, в частности рассмотренных ранее макросов HIBYTE, LOBYTE и GetStockBrush.

Оператор typedef и создание новых типов данных

Ключевое слово typedef позволяет дать новое имя имеющемуся (или только что созданному) типу данных. Если, например, определить

```
typedef unsigned char BYTE;
```

то в дальнейшем тексте программы для символьных переменных без знака (т. е., по существу, байтовых) вместо громоздкого unsigned char можно будет использовать более наглядный описатель BYTE:

```
BYTE c1='0',c2=0,c3=255;
```

как это, впрочем, и делается при программировании для Windows. Новое имя типа часто обозначают прописными буквами, чтобы подчеркнуть его "искусственный" характер; это, однако, не обязательно.

Как можно увидеть из приведенного примера, оператор typedef имеет следующий формат:

```
typdef имеющийся_тип  вводимый_тип
```

где *имеющийся_тип* – это имя уже имеющегося типа данных, а *вводимый_тип* – новое имя, присваиваемое этому типу.

Для приведенного выше простого примера использование оператора typedef дает тот же конечный результат, что и определение идентификатора BYTE с помощью директивы препроцессора #define:

```
#define BYTE unsigned char
```

(обратите внимание на отсутствие знака ";" в строке с директивой #define и наличие его в предложении с оператором typedef, а также противоположный порядок операндов: директива #define присваивает *левому* операнду значение правого, а оператор typedef придает *правому* операнду смысл левого), однако возможности и область использования оператора typedef шире, чем у директивы #define. Это и понятно, если учесть, что директива #define обрабатывается препроцессором, а оператор typedef – компилятором, который имеет возможность осуществлять дополнительные проверки правильности исходного текста программы, недоступные препроцессору.

Заголовочные файлы, используемые при программировании для Windows (WINDOWS.H, WINDEF.H, WINUSER.H и др.), содержат большое количество определений разнообразных типов данных и функций, выполненных как с помощью директивы #define, так и с помощью оператора typedef. Так, аналогично приведенному выше опре-

делению типа BYTE определяется масса новых типов, в частности WORD, DWORD и UINT:

```
typedef unsigned short WORD;//Слово
typedef unsigned long DWORD;//Двойное слово
typedef unsigned int UINT;//Целое без знака
typedef long LONG;//Длинное слово
```

На основании этих типов, имеющих относительно общий характер, определяются более специфические типы, используемые во многих функциях Windows:

```
typedef UINT WPARAM;//Word parameter, параметр-слово
typedef LONG LPARAM;//Long parameter, параметр-длинное слово
typedef LONG LRESULT;//Long result, длинный результат
```

С помощью оператора typedef описывается огромное количество структур, используемых в Windows. Например, для задания координат точки на экране используется структура POINT:

```
typedef struct tagPOINT//Описывается структура, задающая координаты точки
{
  LONG x; //Координата x
  LONG y; //Координата y
} POINT, *PPOINT;//Описанной структуре назначается имя типа POINT
```

В приведенном выше фрагменте файла WINDEF.H заключены сразу два определения. Конструкция

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

задает состав структуры, которой назначается имя типа tagPOINT. Это определение вложено в конструкцию

```
typedef tagPOINT{...} POINT, *PPOINT;
```

в которой, во-первых, типу tagPOINT дается другое имя – POINT, которое и используется в программах, и, во-вторых, создается тип PPOINT, являющийся *указателем* на структурную переменную типа POINT, о чем говорит символ *.

Аналогично описываются многие другие структуры Windows. Например, для задания координат прямоугольной области экрана в программах используется структура типа RECT, которая также определяется в файле WINDEF.H:

```
typedef struct tagRECT
{
  LONG left; //Левая сторона (x-координата)
  LONG top; //Верх (y-координата)
  LONG right;//Правая сторона (x-координата)
  LONG bottom; //Низ (y-координата)
} RECT, *PRECT;
```

Здесь также определяются не только новый структурный тип RECT, включающий 4 координаты прямоугольника, но и указатель PRECT на такого рода структуры, которым удобно пользоваться в тех случаях, когда в качестве параметра некоторой функции выступает адрес структуры RECT. Как уже отмечалось выше, хотя передать в функцию структурную переменную можно и по значению, однако чаще используется гораздо более эффективная передача по адресу.

Глава 4
Основы разработки приложений Windows

Простейшая программа с главным окном

Приступим, наконец, к освоению программирования в системе Windows. Для начала рассмотрим в деталях программу 4-1, представляющую собой простейшее работоспособное приложение Windows, выводящее на экран пустое главное окно со строкой заголовка и стандартным набором управляющих кнопок (рис. 4.1).

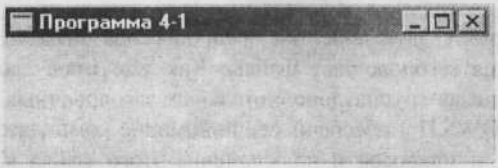


Рис. 4.1. Главное окно приложения, выводимое программой 4-1

```
/*Программа 4-1. Простейшая программа с главным окном*/
/*Операторы препроцессора*/
#include <windows.h>//Два файла с определениями, макросами
#include <windowsx.h>//и прототипами функций Windows
/*Прототип используемой в программе функции пользователя*/
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);//Оконная функция
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  char szClassName[]="MainWindow";//Произвольное имя класса главного окна
  char szTitle[]="Программа 4-1";//Произвольный заголовок окна
  MSG Msg;//Структура Msg типа MSG для получения сообщений Windows
  WNDCLASS wc;//Структура wc типа WNDCLASS для задания характеристик окна
/*Зарегистрируем класс главного окна*/
  ZeroMemory&wc,sizeof(wc));//Обнуление всех членов структуры wc
  wc.lpfnWndProc=WndProc;//Определяем оконную процедуру для главного окна
  wc.hInstance=hInst;//Дескриптор приложения
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);//Стандартная пиктограмма
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);//Стандартный курсор мыши
  wc.hbrBackground=GetStockBrush(LTGRAY_BRUSH);//Светло-серый фона окна
  wc.lpszClassName=szClassName;//Имя класса окна
  RegisterClass(&wc);//Вызов функции Windows регистрации класса окна
/*Создадим главное окно и сделаем его видимым*/
  HWND hwnd=CreateWindow(szClassName,szTitle,//Класс и заголовок окна
       WS_OVERLAPPEDWINDOW,10,10,300,100,//Стиль окна,координаты,размеры
       HWND_DESKTOP,NULL,hInst,NULL);//Родитель, меню, другие параметры
  ShowWindow(hwnd,SW_SHOWNORMAL);//Вызов функции Windows показа окна
/*Организуем цикл обнаружения сообщений*/
  while(GetMessage(&Msg,NULL,0,0))//Если есть сообщение, передать его нам
        DispatchMessage(&Msg);//и вызвать оконную процедуру WndProc
  return 0;//После выхода из цикла вернуться в Windows
  }//Конец функции WinMain
```

```
/*Оконная функция WndProc главного окна*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){//Переход по значению msg - номеру сообщения
    case WM_DESTROY://При завершении приложения пользователем
      PostQuitMessage(0);//вызвать функцию Windows завершения приложения
      return 0;//Возврат в Windows
  default://В случае всех остальных сообщений Windows обработка
    return(DefWindowProc(hwnd,msg,wParam,lParam));//их по умолчанию
    }//Конец оператора switch
  }//Конец функции WndProc
```

Структура приложения Windows

Программа 4-1 состоит из нескольких отчетливо выделенных блоков (разделов):

- группы операторов препроцессора;
- раздела прототипов используемых в программе прикладных функций;
- главной функции WinMain();
- оконной функции главного окна.

Рассмотрим последовательно эти блоки.

Программа начинается с двух директив препроцессора #include, с помощью которых к программе подключаются заголовочные файлы. Как уже отмечалось, заголовочный файл WINDOWS.H (а также целая группа дополнительных заголовочных файлов, подключаемых "изнутри" файла WINDOWS.H) обеспечивает понимание компилятором смысла типов данных Windows, констант и макросов и подключение этого файла к исходному тексту программы является обязательным. Часть определений, используемых в программах (например, макроса GetStockBrush() и других с ним сходных, или макроса HANDLE_MSG, о котором речь будет идти ниже), содержится в файле WINDOWSX.H, который также необходимо включать практически во все приложения Windows.

Вслед за операторами препроцессора в нашем примере идет раздел прототипов, где определяется прототип единственной в данной программе прикладной функции WndProc(). Вообще-то в программе обязательно должны быть указаны прототипы всех используемых функций, как прикладных, так и системных. Вызовов системных функций Windows у нас довольно много: RegisterClass(), CreateWindow(), GetMessage() и др. Однако прототипы всех этих функций уже определены в заголовочных файлах системы программирования. Так, прототип функции WinMain() описан в файле WINBASE.H:

```
int WINAPI WinMain(
  HINSTANCE  hInstance,//Дескриптор текущего экземпляра приложения
  HINSTANCE  hPrevInstance,//Дескриптор предыдущего экземпляра приложения
  LPSTR  lpszCmdLine,//Указатель на параметры командной строки
  int  nCmdShow//Константа, характеризующая начальный вид окна
  );
```

Легко увидеть, что заголовок функции WinMain() в нашей программе *в точности* соответствует приведенному выше прототипу (за исключением того, что мы опустили неиспользуемые параметры). Иначе и быть не может. Достаточно изменить хотя бы немного характеристики нашей функции WinMain(), как программа либо не пройдет этапа компиляции, либо не будет загружаться для выполнения, либо загрузится, но не будет работать.

Прототипы остальных использованных в программе функций Windows определены в файле WINUSER.H. Таким образом, о прототипах функций Windows можно не заботиться.

Иное положение с оконной функцией WndProc(). Это прикладная функция, ее имя может быть каким угодно, и системе программирования это имя неизвестно. Более того,

при наличии в приложении нескольких окон (а практически всегда так и бывает) в программе описывается несколько оконных функций, по одной для каждого класса окон. Для всех используемых в программе оконных функций необходимо указывать их прототипы.

С другой стороны, *формат* оконной функции, т. е. количество и типы входных параметров функции, как и тип возвращаемого ею значения, определены системой Windows и не могут быть произвольно изменены. Действительно, оконная функция вызывается из Windows при поступлении в приложение того или иного сообщения. При ее вызове Windows передает ей вполне определенный список параметров, и функция должна иметь возможность эти параметры принять и работать с ними. Поэтому в интерактивном справочнике системы программирования дается *шаблон* (template) оконной функции, который по своему виду очень похож на прототип, однако является не прототипом конкретной функции, а заготовкой для прикладного программиста:

```
LRESULT CALLBACK WindowProc(
  HWND  hwnd,//Дескриптор окна
  UINT  uMsg,//Код сообщения
  WPARAM  wParam,//Первый параметр сообщения
  LPARAM  lParam //Второй параметр сообщения
  );
```

Опять можно заметить, что наша оконная функция, имея другое имя, в точности соответствует приведенному выше шаблону: принимает 4 параметра указанных типов и возвращает (в Windows) результат типа LRESULT. Кроме того, она объявлена с описателем CALLBACK. Что обозначает этот описатель?

В файле WINDEF.H символическое обозначение CALLBACK объявляется равнозначным ключевому слову языка C++ __stdcall, которое определяет правила взаимодействия функций с вызывающими процедурами. В Win32 практически для всех функций действует так называемое соглашение стандартного вызова. Это соглашение определяет, что при вызове функции ее параметры помещаются в стек в таком порядке, что в глубине стека оказывается последний параметр, а на вершине – первый. Сама функция, разумеется, знает о таком расположении ее параметров и выбирает их из стека в правильном порядке. Для 16-разрядных функций Windows действует соглашение языка Паскаль, при котором порядок помещения параметров в стек обратный.

Функционирование приложения Windows

Итак, программа 4-1 состоит из двух функций: главной функции, всегда имеющей имя WinMain, и так называемой оконной функции (оконной процедуры), которой в нашем случае дано имя WndProc. При запуске приложения Windows управление всегда передается функции WinMain(), которая, таким образом, должна присутствовать в любой программе. Более того, эта функция, имея в принципе циклический характер, выполняется в течение всей жизни приложения. Основное назначение функции WinMain() – выполнение инициализирующих действий и организация цикла обнаружения сообщений.

Инициализирующие действия, выполняемые в функции WinMain(), заключаются в регистрации класса главного окна, создании этого окна и его показе, т. е. выводе на Рабочий стол Windows. Эти действия осуществляются вызовом соответствующих функций Windows: RegisterClass(), CreateWindow() и ShowWindow().

Выполнив необходимые инициализирующие действия, главная функция WinMain() входит в бесконечный цикл обнаружения сообщений.

Сообщения Windows являются, пожалуй, самой важной концепцией этой операционной системы. Каждый раз, когда происходит какое-то событие, затрагивающее интересы

программы (например, пользователь выбирает пункт меню или нажимает на кнопку в окне приложения), Windows посылает приложению сообщение об этом событии. Задача функции WinMain() заключается в приеме этого сообщения и передаче его второму важнейшему компоненту любого приложения Windows – оконной функции.

Цикл обработки сообщений в простейшем виде состоит из одного предложения

```
while(GetMessage(&Msg,NULL,0,0))
      DispatchMessage(&Msg);
```

Функция GetMessage() анализирует очередь сообщений приложения. Если в очереди обнаруживается сообщение, GetMessage() изымает его из очереди и передает в приложение (конкретно – в структурную переменную типа MSG, в нашем случае – в переменную с именем Msg). Выполняемая далее функция DispatchMessage() вызывает оконную функцию, передавая ей все данные, входящие в сообщение.

Оконная функция состоит из стольких фрагментов, сколько конкретных сообщений предполагается обрабатывать в данном приложении. Если, например, приложение должно отслеживать координаты мыши, то в оконную функцию следует включить фрагмент обработки сообщений о перемещении мыши; если, кроме этого, приложение должно реагировать на сигналы от таймера, в оконную функцию включается фрагмент обработки сообщений от таймера. В нашем примере в оконной функции обрабатыва ется единственное сообщение WM_DESTROY, посылаемое в приложение при его завершении. Все остальные сообщения, которые могут поступить в приложение, но которые нас не интересуют, передаются функции Windows DefWindowProc(), выполняющей минимально необходимую обработку любых сообщений.

В программе 4-1 не предусмотрено выполнение каких-либо прикладных действий; пользователь может только изменять с помощью мыши размеры окна, перемещать его по Рабочему столу или, нажав на кнопку с крестиком, завершить приложение. С другой стороны, все элементы программы 4-1 входят с теми или иными вариациями практически в любое приложение Windows. В этом плане рассматриваемую программу можно считать заготовкой, шаблоном для разработки более сложных и содержательных приложений Windows.

Главная функция WinMain()

Венгерская нотация

Главная функция приложения WinMain() начинается в нашем примере с определения переменных, которые будут использоваться в программе. Как уже отмечалось ранее, в программах на языке C++ объявления переменных могут быть разбросаны по телу программы. В программе описаны четыре переменные: две символьные строки с именем класса главного окна и его заголовком и две структурные переменные типов MSG и WNDCLASS. Почему для строковых переменных выбраны такие странные имена?

Программы для Windows, даже относительно простые, содержат большое количество переменных, и желательно, чтобы в их именах был какой-то порядок. Естественно стараться давать переменным содержательные имена, и, если смысл переменной передается двумя или тремя словами, лучше каждое слово начинать с прописной буквы. С этой точки зрения достаточно разумными выглядят имена Title или ClassName. Однако в программах для Windows часто идут еще дальше, включая в имя каждой переменной еще и информацию о ее *типе*. Это делается с помощью так называемой венгерской нотации.

Суть венгерской нотации заключается в том, что имя переменной или функции предваряется одной или несколькими буквами – префиксом, говорящим о типе этой переменной. Так, префикс n обозначает целочисленную переменную, sz – символьную строку, заканчивающуюся двоичным нулем, h – дескриптор того или иного объекта. Отсюда и взялись несколько неуклюжие на первый взгляд имена переменных нашей программы szClassName и szTitle. Венгерская система широко используется в описаниях внутренних структур Windows. Так, в структуре типа WNDCLASS, с которой имеет дело функция Register(), имеются члены lpszClassName (lpsz – long pointer string zero, дальний указатель на символьную строку), lpfnWndProc (lpfn – long pointer function, дальний указатель на функцию), hIcon (h – handle, дескриптор) и др. Представление о венгерской нотации несколько облегчает изучение содержимого этих структур и облегчает использование их элементов.

В табл. 4.1 приведены наиболее употребительные префиксы венгерской нотации.

Таблица 4.1. Префиксы венгерской нотации (разрядность дана для 32-разрядных приложений)

Префикс	*Расшифровка*	*Значение*
b	Bool	Логическая (булева) переменная, 32 бита
c	Character	Символ, 1 байт
dw	DoubleWord	Двойное слово без знака, 32 бита
f	Function	Функция
h	Handle	Дескриптор объекта
l	long	Длинное целое со знаком, 32 бита
lp	LongPointer	Дальний указатель, 32 бита
lpsz	LongPointer StringZero	Дальний указатель на символьную строку, заканчивающуюся двоичным нулем, 32 бита
n	iNt	Целое со знаком, 32 бита
p	Pointer	Ближний указатель, 32 бита
pt	PoinT	X- и y-координаты точки, упакованные в 64 бита
rgb	RedGreenBlue	Цвет из красной, зеленой и синей составляющих, упакованный в 32 бита
sz	StringZero	Символьная строка, заканчивающаяся двоичным нулем
u	Uint	Целое без знака, 32 бита
w	Word	Слово без знака, 16 бита

Еще раз напомним, что в Win32 все указатели являются ближними, хотя и занимают 4 байта. Разница между ближними и дальними указателями существовала только в Win16.

Параметры функции WinMain()

Запуская приложение (из среды разработки или с помощью кнопки "Пуск"), мы фактически передаем управление программам Windows, которые загружают в память нашу программу и вызывают ее главную функцию, которая должна иметь имя WinMain и описатель WINAPI. Напомним, что прототип функции WinMain() описан в файле WINBASE.H следующим образом:

```
int WINAPI WinMain(
  HINSTANCE hInstance,//Дескриптор экземпляра приложения
  HINSTANCE hPrevInstance,//В Win32 не используется
  LPSTR lpszCmdLine,//Адрес параметров командной строки
  int nCmdShow);//Режим запуска
```

Вызывая функцию WinMain(), Windows передает ей 4 обусловленных аргумента, которые, поступая в функцию WinMain(), становятся ее внутренними локальными параметрами.

Первый параметр типа HINSTANCE, которому для краткости дано имя hInst, представляет собой дескриптор данного экземпляра приложения. Он назначается приложению при его запуске системой Windows и служит для его идентификации. Многие функции Windows используют этот дескриптор в качестве входного параметра, поэтому в дальнейшем мы будем сохранять его в глобальной переменной, чтобы сделать доступным всем функциям приложения. В данной программе сохранение hInst не предусмотрено, так как в программе нет никаких функций, кроме главной.

Второй параметр того же типа, который в документации назван hPrevInstance, в 16-разрядных приложениях являлся дескриптором предыдущего экземпляра этого же приложения и принимал ненулевое значение, если приложение запускалось в нескольких экземплярах. Анализируя значение параметра hPrevInstance, можно было определить, является ли данный экземпляр приложения первым. В Win32 этот параметр всегда равен нулю и не имеет смысла. Соответственно нет необходимости предусматривать для него локальную переменную, поэтому в заголовке функции WinMain() указан только тип второго параметра (HINSTANCE), но нет обозначения переменной.

Следующий параметр, lpszCmdLine, представляет собой указатель на строку, содержащую параметры командной строки запуска приложения, если оно запускалось через кнопку "Пуск" и при его запуске кроме имени программы были указаны еще какие-либо параметры. Поскольку мы не предполагаем запускать наше приложение таким образом и этот параметр функции WinMain() нам не нужен, в заголовке функции указан только его тип LPSTR, а имя параметра опущено.

Наконец, последний параметр, nCmdShow, характеризует режим запуска. Режим запуска любого приложения можно установить, если, создав ярлык для приложения, открыть для него окно "Свойства", перейти на вкладку "Ярлык" и раскрыть список "Окно". Если, далее, выбрать в этом списке пункт "Свернутое в значок", Windows, запуская приложение, будет свертывать его в пиктограмму. В этом случае из Windows в WinMain() поступает значение параметра nCmdShow, равное символической константе SW_SHOWMINNOACTIVE=7. Если же включен режим Стандартный размер, окно приложения на экране развертывается до размера, заданного в самом приложении, а в WinMain() поступает константа SW_SHOWNORMAL=1. Полученную через параметр nCmdShow константу можно затем использовать в качестве параметра при вызове функции Windows ShowWindow(). Мы поступили проще, указав при вызове ShowWindow() явным образом константу SW_SHOWNORMAL.

Разумеется, внутренние, действующие в программе имена для параметров функции WinMain, как и в принципе для любой другой функции, пользователь может выбирать по своему усмотрению. Однако это общее правило не всегда справедливо. Позже мы увидим, что параметры оконной функции должны иметь вполне определенные имена.

Состав функции WinMain()

В типичном приложении Windows главная функция WinMain() должна выполнить по меньшей мере 3 важные процедуры:

- Зарегистрировать в системе Windows класс главного окна. Если помимо главного окна планируется выводить на экран внутренние, порожденные окна, то их классы также необходимо зарегистрировать. Windows выводит на экран и обслуживает только зарегистрированные окна.
- Создать главное окно и показать его на экране. Порожденные окна также необходимо создать, хотя это можно сделать и позже, и не обязательно в функции WinMain().
- Организовать цикл обнаружения сообщений, поступающих в приложение, а также процедуру их обработки. Вся дальнейшая жизнь приложения будет фактически состоять в бесконечном выполнении этого цикла и в обработке поступающих в приложение сообщений. Запущенное приложение Windows обычно функционирует до тех пор, пока пользователь не подаст команду его завершения с помощью системного меню или вводом команды Alt+F4. Эти действия приводят к завершению главной функции и удалению приложения из списка действующих задач.

Класс окна и его характеристики

Как уже отмечалось, для вывода на экран любого окна, в частности главного окна приложения, необходимо прежде всего зарегистрировать *класс окна*, в котором задаются наиболее общие характеристики всех окон данного класса. Это действие выполняется в первой, инициализирующей части функции WinMain().

Действия по регистрации класса окна заключаются в заполнении структуры типа WNDCLASS, служащей для описания характеристик класса регистрируемого окна, и вызове затем функции Windows RegisterClass(), которая и выполняет регистрацию данного класса.

Структура WNDCLASS, определенная в файле WINUSER.H, содержит ряд членов, задающих наиболее общие характеристики окна:

```
typedef struct tagWNDCLASS {
  UINT      style;//Стиль класса окна
  WNDPROC   lpfnWndProc;//Имя оконной функции
  int       cbClsExtra;//Число байтов дополнительной информации о классе
  int       cbWndExtra;//Число байтов дополнительной информации об окне
  HINSTANCE hInstance;//Дескриптор приложения
  HICON     hIcon;//Дескриптор пиктограммы приложения
  HCURSOR   hCursor;//Дескриптор курсора приложения
  HBRUSH    hbrBackground;//Дескриптор кисти для фона окна
  LPCSTR    lpszMenuName;//Адрес строки с именем меню окна
  LPCSTR    lpszClassName;// Адрес строки с именем класса окна
  } WNDCLASS;//Новое имя типа данной структуры
```

В большинстве случаев нет необходимости определять все члены этой структуры; для упрощения дела мы сначала обнуляем всю структуру вызовом функции Windows ZeroMemory()

```
ZeroMemory(&wc,sizeof(wc));
```

которая записывает, начиная с адреса &wc, нули в количестве sizeof(wc) штук, а затем задаем значения только интересующим нас членам.

При заполнении структурной переменной типа WNDCLASS необходимо обеспечить нулевое значение всех элементов, которым мы *не присваиваем* конкретные значения. Нулевое значение элемента структуры обозначает для Windows, что характеристики этого элемента должны устанавливаться по умолчанию. Это правило, кстати, характерно и для других структур, описывающих те или иные объекты Windows, например для структуры

OPENFILENAME, служащей для вывода на экран стандартного диалога Windows "Открытие файла", или структуры LOGFONT, позволяющей создать шрифт требуемого начертания.

Можно не обнулять структуру Msg явным образом, а воспользоваться тем обстоятельством, что глобальные переменные обнуляются автоматически. Если переменную wc расположить *перед* функцией WinMain(), то при загрузке программы в память она будет заполнена нулями и в вызове функции ZeroMemory() не будет необходимости. Однако слишком увлекаться глобальными переменными не следует. Если некоторая переменная нужна только в конкретной функции, ее лучше объявить внутри этой функции и сделать тем самым локальной. Локализация переменных уменьшает расход памяти и повышает надежность программы. Так или иначе, какое-то средство обнуления использовать необходимо; в противном случае структура окажется заполненной "мусором", что при запуске программы скорее всего приведет к драматическим последствиям.

Наиболее важными для дальнейшего функционирования программы являются три поля структурной переменной wc: hInstance, где хранится дескриптор данного приложения, lpszClassName, куда заносится произвольное имя, присвоенное нами окнам данного класса (у нас это окно единственное) и lpfnWndProc – имя оконной функции. Именно с помощью структуры WNDCLASS Windows, обслуживая нашу программу, определяет адрес оконной функции, которую она должна вызывать при поступлении в окно сообщений. Поле lpszClassName мы заполняем из переменной szClassName, определенной нами в начале программы, а значение дескриптора приложения переносим из параметра hInst функции WinMain().

Менее важными в принципиальном плане, но существенными для разумного поведения приложения являются поля hIcon, hCursor и hbrBackground, куда следует записать дескрипторы пиктограммы, курсора и кисти, которая используется для закраски рабочей области окна.

Курсор относится к *ресурсам* Windows; ресурсы обычно загружаются из специально созданного файла ресурсов с помощью соответствующих функций Windows, в частности LoadCursor(). В качестве первого аргумента этих функций указывается дескриптор приложения, в котором хранится требуемый ресурс, а в качестве второго – имя ресурса. Однако можно обойтись и встроенными ресурсами Windows. Для этого в качестве первого аргумента этих функций указывается NULL (обычно NULL на месте дескриптора приложения обозначает саму систему Windows); второй аргумент надо выбрать из списка встроенных ресурсов Windows. В табл. 4.2 приведены несколько изображений встроенных курсоров с их символическими именами.

Таблица 4.2. Встроенные курсоры

Имя курсора	*Вид курсора*	*Имя курсора*	*Вид курсора*
IDC_ARROW		IDC_SIZENS	
IDC_CROSS		IDC_IBEAM	
IDC_SIZE		IDC_WAIT	

Очевидно, что для главного окна приложения целесообразно выбирать стандартный курсор IDC_ARROW. Остальные курсоры используются в специальных случаях, как правило, загружаясь динамически лишь в определенных ситуациях и на некоторое время. Например, курсор в форме песочных часов традиционно говорит о том, что в системе

протекает какой-то процесс, не позволяющий приложению работать обычным образом, – загрузка файла, вычисления и пр.

Другим ресурсом Windows, указываемым в классе окна, является пиктограмма. Как и курсор, пиктограмма может быть разработана программистом специально для данного приложения и храниться в файле ресурсов приложения или в отдельном файле с расширением .ICO; для учебных задач проще использовать одну из стандартных пиктограмм Windows. В табл. 4.3 приведены 4 встроенные пиктограммы с указанием их символических имен.

Таблица 4.3. Встроенные пиктограммы

Имя пиктограммы	*Вид пиктограммы*	*Имя пиктограммы*	*Вид пиктограммы*
IDI_APPLICATION		IDI_HAND	
IDI_ASTERISK		IDI_QUESTION	

Для того чтобы придать приложению требуемую пиктограмму, надо при заполнении структурной переменной wc типа WNDCLASS загрузить дескриптор пиктограммы в элемент wc.hIcon. Для получения дескриптора следует воспользоваться функцией LoadIcon():

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

Цвет фона окна определяется дескриптором кисти, записанным в структуру WNDCLASS. В принципе кисти можно придать любой цвет, но проще всего воспользоваться одной из встроенных кистей, хранящихся на "складе" Windows. Для получения встроенной кисти следует воспользоваться макросом GetStockBrush(), в качестве единственного аргумента которого указывается константа, характеризующая цвет кисти. Возможные значения констант (их очень немного) перечислены в табл. 4.4.

Таблица 4.4. Предопределенные кисти Windows

Имя кисти	*Цвет*	*Имя кисти*	*Цвет*
BLACK_BRUSH	Черный	LTGRAY_BRUSH	Светло-серый
DKGRAY_BRUSH	Темно-серый	NULL_BRUSH	Прозрачный
GRAY_BRUSH	Серый	WHITE_BRUSH	Белый

Можно ли закрасить главное окно приложения каким-нибудь другим цветом, например зеленым или желтым? Можно, однако таких кистей нет на "складе", и их придется создавать заново. Для создания кисти служит функция CreateSolidBrush(), в качестве единственного параметра которой указывается требуемый цвет в виде трех чисел, характеризующих интенсивность красной, зеленой и синей составляющих цвета. Для упаковки этих трех чисел в одно 4-байтовое слово и преобразования полученного значения в тип COLORREF, требуемый функцией CreateSolidBrush(), служит макрос RGB(). Таким образом, чтобы сделать главное окно приложения, например, бирюзовым, надо заполнить элемент hbrBackground структурной переменной wc следующим образом:

```
wc.hbrBackground=CreateSolidBrush(RGB(0,255,255));//Бирюзовая кисть для фона
```

В гл. 5 вопрос о формировании цвета и использовании цветных кистей и перьев будет рассмотрен более подробно.

Отметим еще назначение *стиля* класса окна, хотя в нашем примере этот элемент структуры не используется. Стиль представляет собой целое число (32 бита), отдельные биты которого закреплены за теми или иными общими характеристиками всех окон, принадлежащих

регистрируемому классу; каждому биту соответствует своя символическая константа. Так, установка бита 3 (0x00000008, константа CS_DBLCLKS) позволяет программе реагировать на двойные щелчки мышью в области окна; установка битов 0 (0x00000001, константа CS_VREDRAW) и 1 (0x00000002, константа CS_HREDRAW) заставляет Windows перерисовывать окно заново при каждом изменении его размеров по горизонтали и вертикали; установка бита 9(0x00000200, константа CS_NOCLOSE) запрещает закрытие окна пользователем и т. д.

Функция RegisterClass() требует в качестве аргумента адрес структурной переменной типа WNDCLASS, поэтому предложение вызова этой функции выглядит так:

```
RegisterClass(&wc);
```

Зарегистрировав класс окна, можно приступить к его созданию и выводу на Рабочий стол.

Создание и показ окна

Для создания окна, в частности главного окна приложения, используется функция Windows CreateWindow(), прототип которой можно найти в интерактивном справочнике:

```
HWND CreateWindow(
  LPCSTR  lpClassName,//Адрес строки с именем класса
  LPCSTR  lpWindowName,//Адрес строки с заголовком окна
  DWORD  dwStyle,//Стиль окна
  int  x,//Горизонтальная позиция окна на Рабочем столе
  int  y,//Вертикальная позиция окна на Рабочем столе
  int  nWidth,//Ширина окна
  int  nHeight,//Высота окна
  HWND  hWndParent,//Дескриптор родительского окна
  HMENU  hMenu,//Дескриптор меню
  HINSTANCE  hInstance,//Дескриптор приложения
  LPVOID  lpParam //Адрес дополнительных данных
  );
```

Как уже отмечалось ранее, существенными элементами прототипа являются порядок и типы параметров; указание в справочнике имен параметров следует воспринимать как необязательную рекомендацию. В программе 4-1 строка с именем класса окна названа szClassName, строка с заголовком окна – szTitle, а в качестве большинства остальных параметров использованы символические или числовые константы.

Первый параметр lpClassName – адрес строки с именем регистрируемого класса окна. Передача функции CreateWindow() имени класса позволяет использовать при создании окна те общие характеристики окна, которые были определены в структуре WNDCLASS, например форму курсора и цвет фона. Вызывая функцию Create Window() многократно, можно создать много окон данного класса, различающихся, например, размерами и местоположением на экране. Однако пока мы создаем главное окно приложения, которое, очевидно, должно быть в одном экземпляре.

Параметр lpWindowName определяет адрес строки с заголовком, который появится в верхней части окна.

Параметр dwStyle определяет стиль окна. С помощью стиля задаются такие характеристики окна, как вид окружающей его рамки, наличие или отсутствие строки заголовка и целый ряд других. Стиль представляет собой целое число (32 бита), отдельные биты которого закреплены за теми или иными элементами оформления или свойствами окна; каждому биту соответствует своя символическая константа. Так, установка бита 18 (0x00040000, константа WS_THICKFRAME) придает окну толстую рамку; установка бита 19 (0x00080000, WS_SYSMENU) снабжает окно системным меню; бит 21 (0x00200000,

WS_VSCROLL) отвечает за появление линейки вертикальной прокрутки и т. д. Операция побитового ИЛИ (знак |) позволяет "набрать" требуемый комплект свойств.

Обычно главное окно описывается константой WS_OVERLAPPEDWINDOW (0x00CF0000), в которую входят элементы стиля WS_OVERLAPPED (перекрывающееся окно), WS_CAPTION (строка заголовка), WS_SYSMENU (системное меню), WS_THICKFRAME (толстая рамка), WS_MINIMIZEBOX (кнопка минимизации, т. е. свертывания окна в пиктограмму) и WS_MAXIMIZEBOX (кнопка максимизации, т. е. развертывания окна на весь экран). Если требуется создать главное окно с меньшим количеством элементов, весь необходимый набор можно указать явным образом. Так, для создания главного окна без кнопки максимизации константу стиля надо задать следующим образом:

```
WS_THICKFRAME | WS_SYSMENU | WS_MINIMIZEBOX
```

(строка заголовка появляется в главном окне в любом случае, а константа WS_OVERLAPPED равна нулю, и ее указание не влияет на стиль).

С другой стороны, воспользовавшись операторами побитовых операций И (&) и НЕ (~), можно не набирать заново всю комбинацию констант, а просто исключить ненужный элемент WS_MAXIMIZEBOX из полного набора:

```
WS_OVERLAPPEDWINDOW & ~WS_MAXIMIZEBOX
```

Следующие 4 параметра определяют x- и y-координаты левого верхнего угла окна относительно начала Рабочего стола и размеры окна по горизонтали и вертикали (в пикселах).

В качестве параметра hWndParent указывается дескриптор родительского окна. Для главного окна, у которого нет родителя, используется константа HWND_ DESKTOP.

Параметр hMenu позволяет задать меню окна. Если меню нет (как в нашем случае) или используется меню класса, заданное в структуре WNDCLASS, этот параметр должен быть равен нулю (NULL).

Параметр hInstance идентифицирует экземпляр приложения. Значение дескриптора приложения было получено нами через параметр hInst функции WinMain().

Последний параметр lpParam является адресом дополнительных данных, которые обычно не требуются; соответственно мы указали "пустой" адрес (NULL).

Символическое обозначение NULL (нуль-указатель) используется в тех случаях, когда для некоторой функции надо указать нулевое значение параметра, являющегося *указателем*. В Borland C++ для Win32 NULL определяется как 0, так что формально во всех случаях вместо NULL можно использовать просто 0; однако обозначение NULL напоминает, что данный параметр является адресом.

Функция CreateWindow() возвращает (при успешном выполнении) дескриптор созданного окна. Этот дескриптор (локальная переменная hwnd) передается в функцию ShowWindow(), которая организует вывод созданного окна на экран. Теперь для правильного функционирования приложения необходимо организовать цикл обнаружения сообщений.

Цикл обнаружения сообщений

Цикл обнаружения сообщений в простейшем виде состоит из одного предложения языка

```
while(GetMessage(&Msg,NULL,0,0))
    DispatchMessage(&Msg);
```

В этом бесконечном (если его не разорвать изнутри) цикле вызывается функция Windows GetMessage(), и, если она возвращает ненулевое значение, вызывается функция Dis-

patchMessage(). Назначение цикла обработки сообщений – получение сообщений, поступающих в приложение, и вызов в ответ на каждое сообщение оконной процедуры для обработки этого сообщения.

Если в течение какого-то времени сообщения в приложение не поступают, цикл обработки сообщений "крутится" вхолостую. Как только в очереди обнаруживается сообщение, функция GetMessage() забирает его из очереди и переносит в структурную переменную, предназначенную для приема сообщений (у нас это переменная Msg). Вызываемая далее функция DispatchMessage() вызывает оконную процедуру того окна, которому предназначено данное сообщение, и передает ей (через ее параметры) содержимое сообщения из структуры Msg. Задача оконной процедуры – выполнить требуемую обработку поступившего сообщения. Когда оконная процедура завершит обработку полученного сообщения, управление вернется в цикл обработки сообщений, который продолжит свое циклическое выполнение в ожидании нового сообщения.

Для того чтобы разобраться в деталях этого очень важного механизма, нам надо познакомиться поближе с самим понятием сообщений.

Сообщения Windows

Возникновение сообщений

Сообщения являются реакцией системы Windows на различные происходящие в системе события: движение мыши, нажатие клавиши, срабатывание таймера и т. д. Отличительным признаком сообщения является его код, который может принимать значения (для системных сообщений) от 1 до 0x3FF. Каждому коду соответствует своя символическая константа, имя которой достаточно ясно говорит об источнике сообщения. Так, при движении мыши возникают сообщения WM_MOUSEMOVE (код 0x200), при нажатии на левую клавишу мыши – сообщение WM_LBUTTONDOWN (код 0x201), при срабатывании таймера – WM_TIMER (код 0x113).

Перечисленные события относятся к числу *аппаратных*; однако сообщения могут возникать и в результате *программных* действий системы или прикладной программы. Так, по ходу создания и вывода на экран главного окна Windows последовательно посылает в приложение целую группу сообщений, сигнализирующих об этапах этого процесса: WM_GETMINMAXINFO для уточнения размеров окна, WM_ERASEBKGND при заполнении окна цветом фона, WM_SIZE при оценке размеров рабочей области окна, WM_PAINT для получения от программы информации о содержимом окна и многие другие. Некоторые из этих сообщений Windows обрабатывает сама; другие обязана обработать прикладная программа.

Может быть и обратная ситуация, когда сообщение создается в прикладной программе по воле программиста и посылается в Windows для того, чтобы система выполнила требуемые действия (например, заполнила конкретной информацией окно со списком или сообщила о состоянии некоторого элемента управления). Сообщения такого рода тоже стандартизованы и имеют определенные номера.

Наконец, программист может предусмотреть собственные сообщения и направлять их в различные окна приложения для оповещения о тех или иных ситуациях.

Рассмотрим процедуру пересылки и состав аппаратного сообщения на примере сообщения WM_MOUSEMOVE о движении мыши (рис. 4.2). Это сообщение возникает при каждом перемещении мыши на определенное (очень небольшое) расстояние.

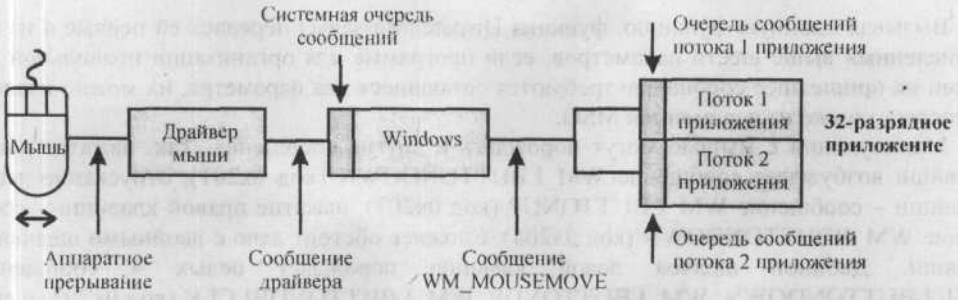


Рис. 4.2. Процедура создания и пересылки сообщения от мыши

Перемещение мыши возбуждает сигнал аппаратного прерывания, который, поступив в компьютер, активизирует драйвер мыши, входящий в состав Windows. Драйвер мыши формирует пакет данных и пересылает его в форме сообщения в системную очередь сообщений Windows, откуда оно поступает в ту или иную очередь приложения.

В Win32 единицей работы компьютера считается поток выполнения. Каждое приложение создает по меньшей мере один, первичный поток, однако может создать и много потоков. Концепция потоков позволяет организовать в рамках одного приложения параллельное выполнение нескольких фрагментов программы.

Для каждого потока в 32-разрядном приложении создается своя очередь сообщений. Эти очереди не имеют фиксированного размера, а могут неограниченно расширяться. Сообщения из системной очереди передаются не в приложение в целом, а распределяются по его потокам. Сообщения от мыши обычно (хотя не всегда) адресованы тому окну, над которым находится ее курсор. Действительно, щелкая по пункту меню или по кнопке в некотором окне, мы хотим вызвать действие именно для этого окна. Окна же создаются с помощью функции CreateWindow() тем или иным потоком приложения. Поэтому сообщения от мыши направляются в очередь того потока, который создал данное окно.

Рассмотрим теперь, из чего состоит каждое сообщение. В начале главной функции приложения WinMain мы объявили структурную переменную Msg. Это важнейшая переменная, с помощью которой в программу передается содержимое сообщений Windows. Каждое сообщение представляет собой пакет из шести данных, описанных в файле WINUSER.H с помощью структуры типа MSG:

```
typedef struct tagMSG {
  HWND  hwnd;//Дескриптор окна, которому адресовано сообщение
  UINT  message;//Код данного сообщения
  WPARAM  wParam;//Дополнительная информация
  LPARAM  lParam;//Дополнительная информация
  DWORD  time;//Время отправления сообщения
  POINT  pt;//Позиция курсора мыши на момент отправления сообщения
  } MSG;//Новое имя для типа tagMSG
```

Для сообщения WM_MOUSEMOVE структура Msg заполняется следующей информацией:

Msg.hwnd – дескриптор окна под курсором мыши;

Msg.message – код сообщения WM_MOUSEMOVE=0x200;

Msg.wParam – комбинация битовых флагов, индицирующих состояние клавиш мыши (нажаты/не нажаты), а также клавиш Ctrl и Shift;

Msg.lParam – позиция курсора мыши относительно рабочей области окна;

Msg.time – время отправления сообщения;

Msg.pt – позиция курсора мыши относительно границ экрана.

Вызывая оконную функцию, функция DispatchMessage() передает ей первые 4 из перечисленных выше шести параметров; если программе для организации правильной реакции на пришедшее сообщение требуются оставшиеся два параметра, их можно извлечь непосредственно из переменной MSG.

Манипуляции с мышью могут порождать и другие сообщения. Так, нажатие левой клавиши возбуждает сообщение WM_LBUTTONDOWN (код 0x201), отпускание левой клавиши – сообщение WM_LBUTTONUP (код 0x202), нажатие правой клавиши – сообщение WM_RBUTTONDOWN (код 0x204). Сложнее обстоит дело с двойными щелчками клавиш. Двойной щелчок левой клавиши порождает целых 4 сообщения: WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDBLCLK (код 0x203) и снова WM_LBUTTONUP. Программист может обрабатывать как все эти сообщения, так и только сообщения о двойном нажатии, не обращая внимания на остальные. Механизм образования всех этих сообщений в точности такой же, как и для сообщения WM_MOUSEMOVE (аппаратное прерывание, формирование драйвером мыши пакета данных, установка сообщения в системную очередь, пересылка сообщения в очередь приложения, вызов оконной функции). Даже пакеты данных для этих сообщений не различаются.

Схожим образом формируются сообщения, например, от клавиатуры: WM_KEYDOWN о нажатии любой "несистемной" клавиши (т. е. любой клавиши, не сопровождаемой нажатием клавиши Alt), WM_KEYUP об отпускании несистемной клавиши, WM_SYSKEYDOWN о нажатии "системной" клавиши (т. е. любой клавиши совместно с клавишей Alt) и др.

Рассмотренные сообщения относятся к сообщениями нижнего уровня – они оповещают об аппаратных событиях практически без всякой их обработки Windows. Некоторые аппаратные события предварительно обрабатываются Windows, и в приложение поступает уже результат этой обработки. Так, при выборе щелчком мыши того или иного пункта меню аппаратное прерывание поглощается системой Windows и вместо сообщения WM_LBUTTONDOWN формируется сообщение WM_COMMAND, в число параметров которого входит идентификатор того пункта меню, над которым был курсор мыши. Это избавляет нас от необходимости анализа положения курсора мыши и выделения всех положений, соответствующих прямоугольной области данного пункта меню. Другой пример – сообщение WM_NCLBUTTONDOWN, которое формируется Windows, если пользователь нажал левую клавишу мыши в нерабочей области окна, т. е. на его заголовке (с целью, например, перетащить окно приложения на другое место экрана).

Рассмотренный выше механизм прохождения сообщений справедлив главным образом для аппаратных сообщений. Большая часть программных сообщений, т. е. сообщений, прямо не связанных с аппаратными событиями, а возникающими по ходу протекания программных процессов в приложениях или в самой Windows, обслуживаются системой иным образом.

Рассмотрим, например, сообщение WM_CREATE. Оно генерируется системой Windows в процессе создания окна, чтобы программист, перехватив это сообщение, мог выполнить необходимые инициализирующие действия: установить системный таймер, загрузить требуемые ресурсы (шрифты, растровые изображения), открыть файлы с данными и т. д. Сообщение WM_CREATE не поступает в очередь сообщений приложения и, соответственно, не изымается оттуда функцией GetMessage(). Вместо этого Windows непосредственно вызывает оконную функцию WndProc и передает ей необходимые параметры (рис. 4.3). С точки зрения программиста обычно не имеет особого значения, каким образом вызывается оконная функция – функцией DispatchMessage() или непосредствен-

но программами Windows. Полезно, однако, иметь в виду, что при обработке, например, сообщения WM_MOUSEMOVE все содержимое этого сообщения находится в структурной переменной Msg, а при обработке WM_CREATE мы имеем дело только с параметрами, переданными Windows в оконную функцию. В переменной Msg в это время находится старое, уже обработанное сообщение, т. е. "мусор".

Рис. 4.3. Прохождение программного сообщения WM_CREATE

Аналогично, т. е. помимо очереди сообщений приложения и структурной переменной Msg, обрабатываются, например, сообщения WM_INITDIALOG (инициализация диалога), WM_SYSCOMMAND (выбор пунктов системного меню), WM_DESTROY (уничтожение окна) и многие другие.

Обработка сообщений

Как уже упоминалось, функция GetMessage() анализирует очередь сообщений приложения. Если в очереди обнаруживается сообщение, GetMessage() изымает его из очереди и передает в структуру Msg, после чего завершается с возвратом значения TRUE. В этом случае сразу вызывается функция DispatchMesage(), активизирующая оконную процедуру. Но что происходит в приложении, если в настоящий момент в очереди текущего потока сообщения отсутствуют?

В Win32 единицей работы компьютера считается поток. Система распределяет процессорное время между потоками на регулярной основе, предоставляя каждому потоку по очереди определенный квант времени порядка 20 мс. Зацикливание одного из потоков нарушит нормальное выполнение конкретно этого потока, однако не отразится на работоспособности остальных потоков и всей системы. Такая организация вычислительного процесса получила название вытесняющей многозадачности.

В действительности очередность переключения задач (или, точнее, потоков) оказывается более сложной, так как передавая управление от потока к потоку, система учитывает их приоритеты. Например, системному потоку, отвечающему за ввод с клавиатуры или от мыши, система назначает более высокий приоритет, чтобы обеспечить быструю реакцию на ввод пользователем новых данных. Более высоким приоритетом также обладают потоки приложения переднего плана (т. е. приложения, окно которого расположено на Рабочем столе поверх всех остальных). При этом система динамически изменяет в некоторых пределах приоритеты выполняющихся потоков по определенному алгоритму, чтобы обеспечить их более эффективное выполнение.

При наличии вытесняющей многозадачности каждое выполняемое приложение периодически получает квант процессорного времени. Что именно делает это приложение, – опрашивает ли свою очередь сообщений или обрабатывает поступившее сообщение, – не имеет значения. Блок Windows, отвечающий за распределение процессорного времени, может в любой момент времени прервать активную задачу и передать управление следующей.

Таким образом, в 32-разрядных приложениях функция GetMessage(), не обнаружив сообщений в очереди "своего" потока, может продолжить опрос очереди до истечения кванта времени или до обнаружения в очереди сообщения. Однако это повлекло бы нерациональную трату процессорного времени. В действительности, если при выполнении функции GetMessage() оказывается, что очередь сообщений пуста, система останавливает выполнение данного потока, переводя его в "спящее" состояние. "Спящий" поток не потребляет процессорного времени и не тормозит работу системы. Поток возобновит свою работу, как только в очереди появится сообщение. Это событие снова вызовет к жизни функцию GetMessage(), которая выполнит предназначенную ей работу – перенос сообщения из очереди сообщений в структурную переменную Msg.

В любом случае функция GetMessage() завершится (с возвратом значения TRUE) лишь после того, как очередное сообщение попадет в переменную Msg.

Далее в цикле while вызывается функция DispatchMessage(). Ее назначение – вызов оконной функции для того окна, которому предназначено очередное сообщение. После того как оконная функция обработает сообщение, возврат из нее приводит к возврату из функции DispatchMessage() на продолжение цикла while. Весь этот процесс схематически изображен на рис. 4.4.

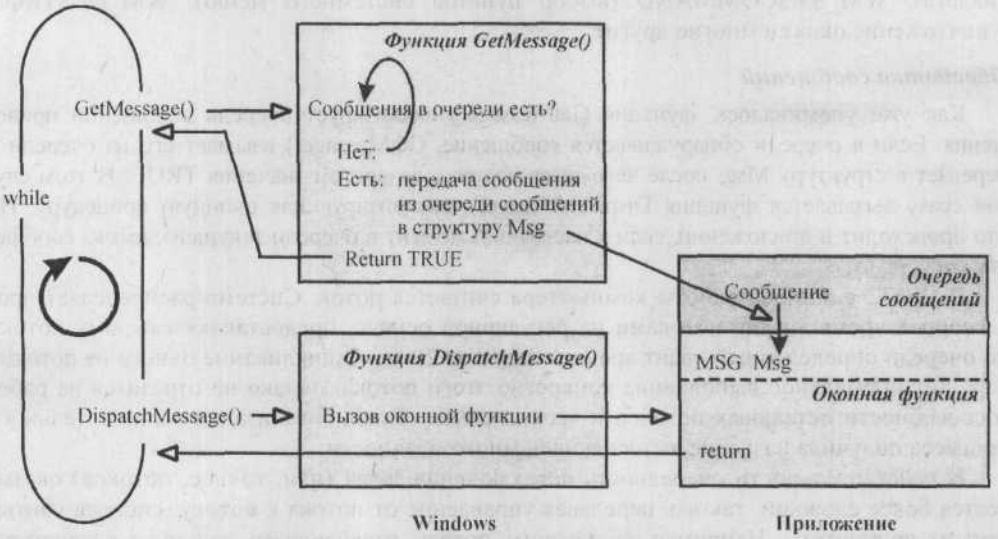


Рис. 4.4. Действие цикла обработки сообщений

Функция GetMessage() требует 4 параметра. Первый из них – адрес структуры Msg, в которую GetMessage() должна передать изъятое из очереди сообщение. Второй параметр типа HWND позволяет определить окно, чьи сообщения будут изыматься функцией GetMessage(). Если этот параметр равен NULL, GetMessage() работает со всеми сообщениями данного приложения.

Два последних параметра определяют диапазон сообщений, которые анализируются функцией GetMessage(). Если, например, в качестве этих параметров указать константы WM_KEYFIRST и WM_KEYLAST, GetMessage будет забирать из очереди только сообщения, относящиеся к клавиатуре; константы WM_MOUSEFIRST и WM_MOUSELAST позволят работать только с сообщениями от мыши. Чаще всего надо анализировать все

сообщения. Чтобы исключить фильтрацию сообщений, оба параметра должны быть равны нулю.

Особая ситуация возникает, если функция GetMessage() обнаруживает в очереди сообщение WM_QUIT с кодом 0x12. В этом случае GetMessage() сразу же завершается с возвратом значения FALSE. Однако цикл while выполняется, лишь если GetMessage() возвращает TRUE. Возврат FALSE приводит к завершению цикла и переходу на предложение

```
return 0;
```

т. е. к завершению функции WinMain() и всего приложения. Таким образом, условием завершения приложения является появление сообщения WM_QUIT. Как оно возникает, будет рассмотрено ниже.

Оконная функция

Структура оконной функции

Как было показано в предыдущем разделе, оконная функция вызывается, как только в структуру Msg попадает очередное сообщение, извлеченное из входной очереди. Задача оконной функции – определить природу сообщения и обработать его соответствующим образом.

Из заголовка оконной функции

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

видно, что она получает при активизации ее (функцией DispatchMessage()) 4 параметра. Первый параметр (hwnd) – дескриптор окна, которому предназначено данное сообщение. Это тот самый дескриптор, который был получен нами как результат работы функции CreateWindow(). Теперь этот же дескриптор вернулся к нам из Windows как параметр оконной функции. Он особенно полезен в тех случаях, когда на базе одного класса создается несколько различающихся чем-то окон. Если класс один, то и оконная функция для всех этих окон одна; анализируя тогда параметр hwnd, программа может определить, в какое именно окно пришло сообщение. У нас окно одно, однако аргумент hwnd все же понадобится.

Второй параметр (msg) определяет код пришедшего сообщения. Поскольку сообщений много, оконная функция должна прежде всего проанализировать этот код и осуществить переход на фрагмент обработки соответствующего сообщения. В настоящем (не лучшем, как будет видно из дальнейшего) варианте программы анализ аргумента msg осуществляется с помощью конструкции switch...case.

Оконная функция должна обрабатывать *все* поступающие в нее сообщения, и ее текст должен быть приблизительно таким:

```
switch(msg)
  {
    case WM_CREATE:
      ...//Обработка сообщения WM_CREATE
    case WM_DESTROY:
      ...//Обработка сообщения WM_DESTROY
    case WM_KEYDOWN:
      ...//Обработка сообщения WM_KEYDOWN
    case WM_MOUSEMOVE:
      ...//Обработка сообщения WM_MOUSEMOVE
    case WM_TIMER:
      ...//Обработка сообщения WM_TIMER
//и т. д. для всех возможных сообщений
```

Однако реально в любой программе, по существу, требуется обрабатывать далеко не все сообщения. Например, если программа управляется только мышью, в ней нет необходимости обрабатывать сообщения от клавиатуры; совсем не обязательно программа имеет дело с таймером и т. п. Надо еще иметь в виду, что приложение получает большое количество сообщений системного характера, например о перерисовке отдельных элементов окна, определении его размеров, изменении положения окна и т. д. Все эти сообщения мы, возможно, даже не умеем обрабатывать должным образом. Для того чтобы дать возможность программисту работать только с теми сообщениями, которые ему нужны, в Windows предусмотрена специальная функция DefWindowProc() (от Default Window Procedure, процедура Windows по умолчанию), которая умеет правильно обрабатывать практически все сообщения Windows. Единственным сообщением, не входящим "в юрисдикцию" DefWindowProc(), является сообщение об уничтожении окна WM_DESTROY. Поэтому в простейшем случае, когда мы вообще не предполагаем работать с сообщениями, оконная функция все же должна включать обработку сообщения WM_DESTROY:

```
switch(msg)
  {
    case WM_DESTROY:
      PostQuitMessage(0);
      return 0;
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
  }
```

Такая оконная функция обеспечит обработку по умолчанию всех поступающих в главное окно сообщений. Остается выяснить вопрос, как возникает сообщение WM_DESTROY и в чем должна заключаться его обработка.

Процедура завершения приложения

Обычно приложения Windows завершают свою работу по команде пользователя. Он может щелкнуть по кнопке завершения  в правом верхнем углу окна, дважды щелкнуть по маленькой пиктограмме  в левом верхнем углу окна, вызвать системное меню одиночным щелчком по этой пиктограмме и выбрать пункт "Закрыть" или, наконец, нажать клавиши Alt+F4. Во всех этих случаях Windows убирает с экрана окно приложения и посылает в приложение сообщение WM_DESTROY (не в очередь приложения, а с непосредственным вызовом оконной функции).

В чем должна состоять обработка этого сообщения? В процессе своего выполнения программа могла использовать те или иные ресурсы Windows: создать кисти, перья или шрифты, установить таймеры, динамически выделить память и т. д. Перед завершением приложения эти ресурсы следует освободить, иначе можно вывести из строя всю систему. Возможно также, что программа использовала какие-то средства (не связанные с Windows), которые перед завершением работы следует привести в порядок, например, выключить включенную аппаратуру. Наконец, вам может захотеться вывести на экран предупреждающее сообщение.

Выполнив все эти завершающие действия, программа должна вызвать функцию Windows PostQuitMessage(). Эта функция генерирует сообщение Windows WM_QUIT, поступающее в очередь сообщений приложения. Функция GetMessage() содержит в себе своеобразный алгоритм: обнаружив в очереди приложения сообщение WM_QUIT,

она тут же завершается с возвратом значения FALSE. Это приводит к разрыву цикла while обработки сообщений, выполнения последнего оператора return функции WinMain и завершению программы. Описанная процедура схематически изображена на рис. 4.5.

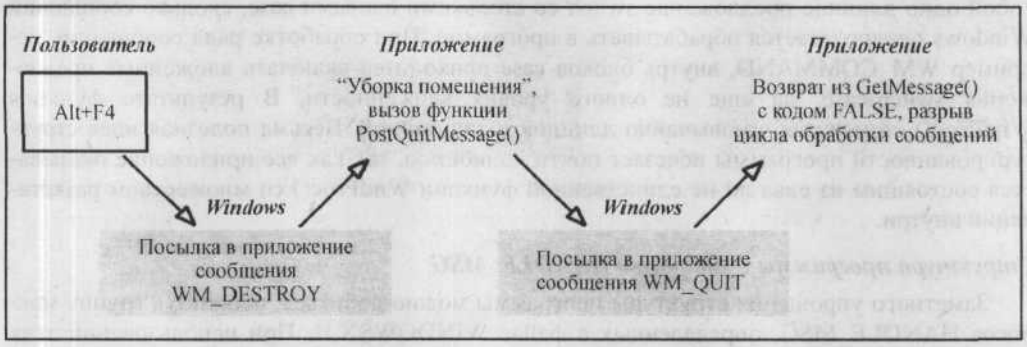


Рис. 4.5. Процедура завершения приложения

Подведем итоги. Типичное приложение Windows имеет стандартную структуру, изображенную на рис. 4.6. В программе можно выделить два основных элемента: главную функцию и оконную функцию главного окна. В главной функции прежде всего осуществляются подготовительные действия: регистрация в Windows класса главного окна, а также создание и показ этого окна. Затем главная функция входит в бесконечный цикл обнаружения сообщений. Каждое поступившее в окно сообщение вызывает к жизни оконную функцию.

Главная функция WinMain()

1. Регистрация класса окна
2. Создание и показ окна
3. Цикл обработки сообщений

Оконная процедура

Если WM_KEYDOWN, то ...
Если WM_MOUSEMOVE, то ...
Если WM_DESTROY, то PostQuitMessage()
Если любое ненужное сообщение, то DefWindowProc()

Рис. 4.6. Структура типичного приложения Windows

Оконная функция главного окна состоит из стольких фрагментов, сколько предполагается обрабатывать сообщений. Конкретное содержимое каждого фрагмента определяет программист. Все сообщения, не нуждающиеся в прикладной обработке, поступают в функцию обработки сообщений по умолчанию DefWindowProc() и обрабатываются системой Windows.

Если в главном окне приложения предполагается создать внутренние, порожденные окна (как оно обычно и бывает), то подготовительные действия усложняются, так как необходимо зарегистрировать в Windows классы всех этих окон, а затем создать их и показать на экране. В этом случае для каждого класса окон предусматривается своя оконная функция.

Макрос HANDLE_MSG

Как было показано в предыдущем разделе, оконная функция должна представлять собой одно длинное предложение switch со столькими блоками case, сколько сообщений Windows предполагается обрабатывать в программе. При обработке ряда сообщений, например WM_COMMAND, внутрь блоков case приходится включать вложенные предложения switch-case, да еще не одного уровня вложенности. В результате функция WndProc() становится чрезвычайно длинной и запутанной. Весьма полезная идея структурированности программы исчезает почти полностью, так как все приложение оказывается состоящим из едва ли не единственной функции WndProc() со множеством разветвлений внутри.

Структура программы с макросом HANDLE_MSG

Заметного упрощения структуры программы можно добиться, используя группу макросов HANDLE_MSG, определенных в файле WINDOWSX.H. При использовании этих макросов все процедуры обработки сообщений выделяются в *отдельные функции*, а в оконной функции WndProc() остаются только строки переключения на эти функции при приходе того или иного сообщения. Оконная функция, даже при большом количестве обрабатываемых сообщений, становится короткой и чрезвычайно наглядной; наличие же для обработки каждого сообщения отдельной функции также весьма упрощает разработку их алгоритмов, и особенно отладку.

Модифицируем программу 4-1, введя в ее оконную функцию макрос HANDLE_MSG. Фактически изменению подвергнется только оконная функция, однако ради удобства сравнения мы приведем здесь модифицированную программу полностью, дав ей имя 4-2 и изменив соответствующим образом заголовок главного окна.

```
/*Программа 4-2. Макрос HANDLE_MSG*/
/*Операторы препроцессора*/
#include <windows.h>//Два файла с определениями, макросами
#include <windowsx.h>//и прототипами функций Windows
/*Прототипы используемых в программе функций пользователя*/
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);// Оконная функция
void OnDestroy(HWND);// Прототип функции OnDestroy
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  char szClassName[]="MainWindow";//Произвольное имя класса главного окна
  char szTitle[]="Программа 4-2";//Произвольный заголовок окна
  MSG Msg;//Структура Msg типа MSG для временного хранения сообщений Windows
  WNDCLASS wc;//Структура wc типа WNDCLASS для задания характеристик окна
/*Зарегистрируем класс главного окна*/
  ZeroMemory (&wc,sizeof(wc));//Обнуление всех членов структуры wc
    wc.lpfnWndProc=WndProc;//Определяем оконную процедуру для главного окна
    wc.hInstance=hInst;//Дескриптор приложения
    wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);//Стандартная пиктограмма
    wc.hCursor=LoadCursor(NULL,IDC_ARROW);//Стандартный курсор мыши
    wc.hbrBackground=GetStockBrush(LTGRAY_BRUSH);//Светло-серая кисть фона
    wc.lpszClassName=szClassName;//Имя класса главного главного окна
  RegisterClass(&wc);//Вызов функции Windows регистрации класса окна
/*Создадим главное окно и сделаем его видимым*/
  HWND hwnd=CreateWindow(szClassName,szTitle,//Класс и заголовок окна
       WS_OVERLAPPEDWINDOW,10,10,250,100,//Стиль окна, координаты и размеры
       HWND_DESKTOP,NULL,hInst,NULL);//Родитель, меню, другие параметры
  ShowWindow(hwnd,SW_SHOWNORMAL);//Вызов функции Windows показа окна
/*Организуем цикл обнаружения сообщений*/
  while(GetMessage(&Msg,NULL,0,0))// Если есть сообщение, передать его нам
```

```
    DispatchMessage(&Msg);// и вызвать оконную процедуру WndProc
  return 0;//После выхода из цикла вернуться в Windows
  }//Конец функции WinMain
/*Оконная функция WndProc главного окна*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){//Переход по значению msg - коду сообщения
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);//При завершении пользователем
    default://В случае всех остальных сообщений Windows обработка их
      return(DefWindowProc(hwnd,msg,wParam,lParam));//по умолчанию
    }//Конец оператора switch
  }//Конец функции WndProc
/*Функция OnDestroy обработки сообщения WM_DESTROY*/
void OnDestroy(HWND){
  PostQuitMessage(0);//Вызов функции Windows завершения приложения
  }//Конец функции OnDestroy
```

В программе 4-2 обрабатывается единственное сообщение WM_DESTROY. Соответственно в программу введена функция обработки этого сообщения OnDestroy(). В документации к Windows рекомендуется образовывать имена функций обработки сообщений из имени класса окна, значка подчеркивания, слова On (при, в случае) и имени соответствующего сообщения. Применительно к нашей программе имена функций обработки сообщений должны выглядеть таким образом:

```
MainWindow_OnDestroy();
MainWindow_OnPaint();
MainWindow_OnCommand();
```

и т. д. Однако мы для функций обработки сообщений, поступающих в *главное* окно, будем ради краткости опускать префикс, характеризующий класс. Для функций, относящихся к внутренним окнам, префикс класса придется использовать, так как разные функции должны, разумеется, иметь разные имена.

Введение в программу новой функции требует определения ее прототипа. Соответственно в раздел прототипов приложения включена строка

```
void OnDestroy(HWND);//Прототип функции обработки сообщения WM_DESTROY
```

Сама же функция OnDestroy() помещена в конце программы, после оконной функции WndProc(). Разумеется, порядок функций в исходном тексте программы не имеет никакого значения и может выбираться по усмотрению программиста, исходя из соображений максимальной наглядности текста программы.

При описании прототипов функций обработки отдельных сообщений и при составлении текстов самих этих функций возникает вопрос об их параметрах и возвращаемых значениях. Наша функция OnDestroy() ничего не возвращает и требует один параметр типа HWND (очевидно, дескриптор главного окна). Однако в случае других функций это не так. Вообще говоря, для каждой функции обработки того или иного сообщения характерен свой набор параметров и свой тип возвращаемого значения. По существу, состав параметров определяется, конечно, характером сообщения, формально же состав и порядок параметров задаются макросами HANDLE_MSG, которые будут подробнее описаны ниже.

Однако извлечь интересующую нас информацию о прототипе функции из текста макроса довольно затруднительно, даже если заниматься исследованием его структуры. Для облегчения программирования в файле WINDOWSX.H для каждого сообщения приведен прототип соответствующей функции с указанием типа, порядка и в какой-то степени смысла ее параметров. Более детальную информацию о данных, поступающих в приложение вместе с сообщением, можно получить с помощью интерактивного справочника среды программирования Borland C++, вызвав справку по интересующему нас со-

общению (например, WM_DESTROY). Таким образом, при написании функций обработки сообщений приходится постоянно обращаться к файлу WINDOWSX.H и справочной системе среды разработки.

В табл. 4.5 приведены прототипы некоторых наиболее употребительных функций обработки сообщений.

Таблица 4.5. Прототипы функций обработки сообщений

Сообщение	*Прототип функции обработки сообщения*
WM_CHAR	void Cls_OnChar(HWND hwnd, UINT ch, int cRepeat);
WM_COMMAND	void Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify);
WM_CTLCOLOR*xxx*	HWND hwnd, HDC hdc, HWND hwndCtl, int constId
WM_CREATE	BOOL Cls_OnCreate(HWND hwnd, CREATESTRUCT FAR* lpCreateStruct);
WM_DESTROY	void Cls_OnDestroy(HWND hwnd);
WM_GETMINMAXINFO	void Cls_OnGetMinMaxInfo(HWND hwnd, MINMAXINFO FAR* lpMinMaxInfo);
WM_INITDIALOG	BOOL Cls_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam);
WM_KEYDOWN	void Cls_OnKeyDown(HWND hwnd, UINT vk, BOOL fDown, int cRepeat, UINT flags);
WM_KEYUP	void Cls_OnKeyUp(HWND hwnd, UINT vk, BOOL fDown, int cRepeat, UINT flags);
WM_KILLFOCUS	void Cls_OnKillFocus(HWND hwnd, HWND hwndNewFocus);
WM_LBUTTONDOWN, WM_LBUTTONDBLCLK	void Cls_OnLButtonDown(HWND hwnd, BOOL fDoubleClick, int x, int y, UINT keyFlags);
WM_LBUTTONUP	void Cls_OnLButtonUp(HWND hwnd, int x, int y, UINT keyFlags);
WM_MOUSEMOVE	void Cls_OnMouseMove(HWND hwnd, int x, int y, UINT keyFlags);
WM_NOTIFY	BOOL Cls_OnNotify(HWND hwnd, INT idCtrl, NMHDR* pnmh);
WM_PAINT	void Cls_OnPaint(HWND hwnd);
WM_QUIT	void Cls_OnQuit(HWND hwnd, int exitCode);
WM_RBUTTONDOWN, WM_RBUTTONDBLCLK	void Cls_OnRButtonDown(HWND hwnd, BOOL fDoubleClick, int x, int y, UINT keyFlags);
WM_RBUTTONUP	void Cls_OnRButtonUp(HWND hwnd, int x, int y, UINT flags);
WM_SETCURSOR	BOOL Cls_OnSetCursor(HWND hwnd, HWND hwndCursor, UINT codeHitTest, UINT msg);
WM_SETFOCUS	void Cls_OnSetFocus(HWND hwnd, HWND hwndOldFocus);
WM_SHOWWINDOW	void Cls_OnShowWindow(HWND hwnd, BOOL fShow, UINT status);

Сообщение	*Прототип функции обработки сообщения*
WM_SIZE	void Cls_OnSize(HWND hwnd, UINT state, int cx, int cy);
WM_SYSCHAR	void Cls_OnSysChar(HWND hwnd, UINT ch, int cRepeat);
WM_SYSCOMMAND	void Cls_OnSysCommand(HWND hwnd, UINT cmd, int x, int y);
WM_SYSKEY	void Cls_OnSysKey(HWND hwnd, UINT vk, BOOL fDown, int cRepeat, UINT flags);
WM_TIMER	void Cls_OnTimer(HWND hwnd, UINT id);

Стоит отметить, что большинство функций обработки сообщений не имеет возвращаемых значений. Это создает дополнительные удобства: при обработке сообщений непосредственно в теле оконной функции не надо каждый раз выяснять с помощью справочной системы, какое значение следует возвращать после обработки данного сообщения.

Расширение макроса HANDLE_MSG

Макрос HANDLE_MSG, коротко говоря, разворачивается в предложение языка C++ с ключевым словом case. Общий же для всех ключевых слов case оператор switch включается в текст оконной функции в явной форме:

```
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_PAINT,OnPaint);//case WM_PAINT: OnPaint();
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy;//case WM_DESTROY: OnDestroy();
```

В действительности, однако, расширение HANDLE_MSG оказывается не таким простым.

Для макроса HANDLE_MSG в составе файла WINDOWSX.H имеется следующее определение:

```
#define HANDLE_MSG(hwnd,message,fn)\
    case(message):return HANDLE_##message((hwnd),(wParam),(lParam),(fn))
```

(знак обратной косой черты (\) в определении макроса обозначает переход на следующую строку). Таким образом, предложения

```
HANDLE_MSG(hwnd,WM_PAINT,OnPaint);
HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy;
```

преобразуются в промежуточные макрорасширения

```
case (WM_PAINT):return HANDLE_WM_PAINT((hwnd),(wParam),(lParam),(OnPaint));
case (WM_DESTROY):return HANDLE_WM_DESTROY((hwnd),(wParam),(lParam),(OnDestroy));
```

Знак ## в составе макроопределения обозначает сцепление (конкатенацию) и в данном случае служит для получения составных имен новых макросов HANDLE_WM_PAINT, HANDLE_WM_DESTROY и др. Для каждого сообщения Windows в составе файла WINDOWSX.H имеется отдельный макрос такого вида, причем их макроопределения уже неодинаковы и зависят от характеристик конкретного сообщения.

Для макроса HANDLE_WM_DESTROY дано следующее макроопределение:

```
#define HANDLE_WM_DESTROY(hwnd,wParam,lParam,fn) ((fn)(hwnd),0L)
```

Подставив это определение вместо HANDLE_WM_DESTROY и опустив ненужные скобки, получим окончательное макрорасширение:

```
case WM_PAINT:return (OnDestroy (hwnd),0L);
```

Таким образом, по ходу расширения макроса убрались лишние (для данного сообщения) параметры wParam и lParam и образовалось синтаксически правильное предложение case. Как выполняется это предложение? Если пришло сообщение WM_DESTROY и аргумент msg функции WndProc() равен коду этого сообщения, то выполняется оператор return с двумя аргументами. Прежде всего выполняется предложение языка, стоящее на месте первого аргумента, т. е. вызывается функция OnDestroy (hwnd). Эта функция не должна возвращать каких-либо значений. После ее завершения срабатывает оператор return, возвращающий указанное значение – длинный 0. Любопытно, что завершающий знак ";", который обязательно должен быть в конце любого предложения языка C++, переходит в окончательный текст из *нашей* строки с макросом HANDLE_MSG.

Схожим образом расширяется строка для сообщения, например, WM_PAINT:

```
case WM_PAINT:return(OnPaint (hwnd),0L);
```

Для других сообщений макросы вида HANDLE_*сообщение* имеют более сложные определения, в которых выполняются необходимые преобразования аргументов функции WndProc() в параметры функций обработки сообщений. Приведем в качестве еще одного примера расширение макроса HANDLE_WM_COMMAND:

```
#define HANDLE_WM_COMMAND(hwnd,wParam,lParam,fn(\
   ((fn)((hwnd),(int)(wParam),(HWND)LOWORD(lParam),(UINT)(HIWORD(lParam)),0L)
```

В результате наша строка

```
HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);
```

сначала преобразуется в

```
case (WM_COMMAND):return HANDLE_WM_COMMAND((hwnd),(wParam),(lParam),(OnCommand));
```

а затем окончательно в

```
case WM_COMMAND:return(OnCommand(hwnd,(int)wParam,(HWND)LOWORD(lParam),
                       (UINT)HIWORD(lParam)),0L)
```

Функция обработки сообщения WM_COMMAND должна использоваться в соответствии со своим прототипом

```
void Cls_OnCommand(HWND hwnd,int id,HWND hwndCtl,UINT codeNotify);
```

В результате в эту функцию, которая вызывается в случае выбора пользователем пункта меню или элемента управления диалогового окна, поступают следующие параметры:

- дескриптор окна hwnd;
- преобразованный в тип int аргумент wParam функции WndProc(), который определяет идентификатор выбранного пункта меню или элемента управления (например, кнопки);
- преобразованное в тип HWND младшее слово двухсловного аргумента lParam, которое определяет дескриптор окна, образующего выбранный элемент управления;
- преобразованное в тип UINT старшее слово того же двухсловного аргумента lParam, которое определяет код извещения, т. е. действие, выполненное над элементом управления.

Из этого примера видно, что использование макроса HANDLE_MSG не только существенно повышает структурированность программы, выделяя процедуры обработки сообщений в отдельные функции, но и позволяет распаковать аргументы функции WndProc() и выделить из них данные, которые должны служить параметрами функций обработки конкретных сообщений. При этом в процессе выделения параметров выполняется необходимое преобразование их типов. В литературе макросы HANDLE_MSG часто называют "распаковщиками сообщений".

Окна сообщений

Окна сообщений являются важнейшим средством диалога системы и прикладной программы с пользователем. Как правило, окна сообщений выводятся системой при возникновении в приложении каких-либо сбоев (рис. 4.7), однако в прикладных программах окна сообщений чрезвычайно удобно использовать для вывода содержательных данных – результатов вычислений, информации о ходе выполнения программы и т. д. Окна сообщений могут также использоваться для управления программой, хотя это и не очень удобно.

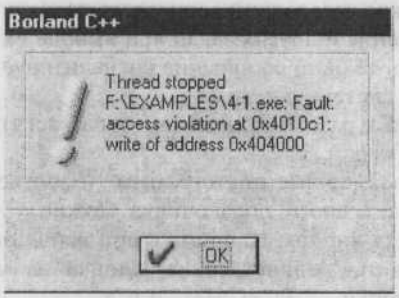


Рис. 4.7. Окно сообщения, выведенное системой при обнаружении сбоя приложения

Воспользуемся окном сообщений, чтобы исследовать адресное пространство программы 4-2. Выведем содержимое сегментных регистров команд, данных и стека, а также смещение главной функции WinMain и одного из данных, например строки с именем класса. Для этого в какое-либо место функции WinMain, например перед регистрацией класса окна, надо вставить следующий фрагмент:

```
char szText[80];
wsprintf(szText,"CS=%X, DS=%X, SS=%X\nWinMain=%X\nszClassName=%X",
        _CS,_DS,_SS,WinMain,szClassName);
MessageBox(NULL,szText,"Из программы",MB_ICONINFORMATION);
```

Результатом выполнения этого фрагмента будет вывод на экран сообщения с таким содержимым (рис. 4.8).

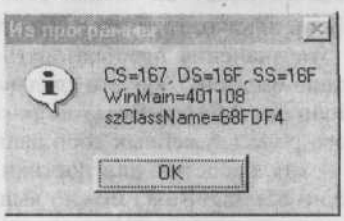


Рис. 4.8. Окно сообщения с данными об адресах программы

Функция MessageBox() позволяет создать и вывести на экран в требуемой точке программы служебное окно с заданным текстом, причем в текст можно включить как текстовые строки, так и числовые данные (значения переменных). Прототип этой функции выглядит следующим образом:

```
int MessageBox(
  HWND hWnd,//Дескриптор родительского окна
  LPCTSTR lpText,//Выводимый текст
  LPCTSTR lpCaption,//Заголовок окна сообщения
  UINT uType //Стиль окна сообщения
  );
```

Как видно из прототипа, функция MessageBox() требует четыре аргумента. Первый аргумент идентифицирует окно, являющееся родительским для окна сообщения. В большинстве случаев вместо дескриптора родительского окна на месте этого аргумент а можно указать NULL – дескриптор Рабочего стола. Второй аргумент представляет собой адрес строки с текстом, выводимым в окно, или саму эту строку, заключенную в кавычки. Третий аргумент –

адрес строки или сама строка с текстом заголовка окна сообщения. Наконец, через четвертый аргумент в функцию передается константа, определяющая вид и поведение окна сообщения. С помощью этой константы задаются 3 элемента окна сообщения:

- вид пиктограммы, характеризующей тип сообщения, – информационное, предупреждающее и пр.;
- набор управляющих кнопок в окне и надписи на них;
- задание кнопки, выбранной по умолчанию (при наличии в окне нескольких кнопок).

Кроме внешнего вида окна, последний аргумент позволяет задать степень блокирования остальных окон приложения и других задач при выводе на экран данного сообщения. Это важно в тех случаях, когда окно сообщения сигнализирует о серьезных неисправностях в приложении или всей системе в целом.

Константы, относящиеся к разным элементам окна, могут объединяться с помощью операции побитового ИЛИ (знак |).

В табл. 4.6 приведены возможные пиктограммы и соответствующие им константы; в статье MessageBox интерактивного справочника можно также найти константы, задающие наборы управляющих кнопок. Например, при использовании константы MB_OK в окно сообщения выводится единственная кнопка с надписью OK; константа MB_YESNO выводит две кнопки – "Да" и "Нет" и т. д.

Таблица 4.6. Возможные пиктограммы окна сообщения

Символические константы	*Значок*	*Символические константы*	*Значок*
MB_ICONASTERISK (или MB_ICONINFORMATION)		MB_ICONHAND (или MB_ICONSTOP)	
MB_ICONEXCLAMATION		MB_ICONQUESTION	

Если текст, выводимый в окно сообщения, известен заранее, его можно просто включить в вызов функции MessageBox() в качестве параметра, например:

```
MessageBox(NULL,"Вы забыли задать данные!","Ошибка",MB_ICONSTOP);
```

Если, однако, требуется вывести значения каких-либо числовых переменных, сначала надо эти значения преобразовать в символьную форму, для чего удобно использовать функцию wsprintf(), как это сделано в приведенном выше фрагменте.

Функция MessageBox() широко используется в прикладных программах для вывода всякого рода служебных сообщений. Нас она будет больше интересовать в другом качестве – как средство диагностики и отладки разрабатываемых программ. С помощью функции MessageBox() можно выводить по ходу программы значения интересующих нас переменных, как это сделано в приведенном выше фрагменте, с целью их сравнения и проверки.

Вернемся к рис. 4.8. Он служит подтверждением некоторых особенностей работы систем Windows, о которых уже говорилось в гл. 1. Для любых 32-разрядных приложений селектор сегмента команд всегда равен 0x167, а селектор совпадающих сегментов данных и стека – 0x16F. Мы знаем, что 32-разрядные приложения работают в модели плоской памяти, где базовые адреса всех сегментов равны нулю, в результате чего виртуальные смещения объектов программы совпадают с их линейными адресами. И программа и данные расположены в 1-м гигабайте линейного адресного пространства, хотя и разнесены в нем на значительное расстояние; конкретно, главная процедура WinMain() находится в 5-м мегабайте, а строка с данными, объявленная в программе как локальная переменная – в 7-м.

Глава 5
Интерфейс графических устройств – GDI

Интерфейс графических устройств (Graphical Device Interface, GDI) представляет собой совокупность программных средств Windows, организующих вывод на различные устройства вывода – прежде всего на экран, но также и на устройства печати или в файлы – всего многообразия графических объектов: текстовых строк, геометрических фигур, растровых изображений и др. GDI предоставляет программисту более двухсот функций для управления режимами вывода и построения на экране требуемых изображений. Сюда относятся функции для создания инструментов рисования (цветные кисти и перья, шрифты различных гарнитур); функции управления цветами; функции получения и задания режимов рисования; функции собственно вывода тех или иных объектов и многие другие.

Помимо самого вывода изображений, в задачу интерфейса GDI входит наблюдение за границами, в которых осуществляется этот вывод. Так, программа может попытаться вывести на экран очень длинную текстовую строку или изображение большого размера. Однако GDI отобразит на экране только те части этих объектов, которые попадают внутрь окна приложения. В результате приложения никогда не затирают друг друга, сосуществуя в пределах своих окон.

GDI участвует также и в организации типичного для Windows и других операционных систем с графической оболочкой многослойного экранного кадра, когда на экране может быть одновременно изображено несколько окон одного или различных приложений, причем окна верхнего уровня отображаются целиком, а от окон нижних уровней видны только выступающие части. Для того чтобы это было возможным, перерисовка каждого окна должна вестись только в пределах видимых в настоящий момент границ.

Для успешного использования графических возможностей Windows в прикладных программах программист должен не только знать состав и особенности применения многочисленных функций, связанных с изображением на экране графических объектов (это, кстати, наиболее простая задача), но и понимать принципы динамического взаимодействия системы с приложением в процессе организации экранного кадра. Одним из наиболее важных системных средств такого рода является сообщение WM_PAINT. Программа 5-1 служит для демонстрации основных правил обработки этого сообщения.

```
/*Программа 5-1. Сообщение WM_PAINT*/
/*Операторы препроцессора*/
#include <windows.h>
#include <windowsx.h>
/*Прототипы используемых в программе функций пользователя*/
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
void OnPaint(HWND);//Прототип функции OnPaint
void OnDestroy(HWND);//Прототип функции OnDestroy
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  char szClassName[]="MainWindow";
  char szTitle[]="Программа 5-1";
  MSG Msg;
```

```
  WNDCLASS wc;
/*Зарегистрируем класс главного окна*/
  ZeroMemory (&wc,sizeof(wc));
  wc.lpfnWndProc=WndProc;
  wc.hInstance=hInst;
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);//Белый фон окна
  wc.lpszClassName=szClassName;
  RegisterClass(&wc);
/*Создадим главное окно и сделаем его видимым*/
  HWND hwnd=CreateWindow(szClassName,szTitle,
    WS_OVERLAPPEDWINDOW,10,10,300,100,
    HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_SHOWNORMAL);
/*Организуем цикл обработки сообщений*/
  while(GetMessage(&Msg,NULL,0,0))
    DispatchMessage(&Msg);
  return 0;
  }
/*Оконная функция главного окна*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_PAINT,OnPaint);//Обработка сообщения WM_PAINT
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);//Обработка сообщения WM_DESTROY
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
/*Функция обработки сообщения WM_DESTROY*/
void OnDestroy(HWND){
  PostQuitMessage(0);//Вызов функции Windows завершения приложения
  }
/*Функция обработки сообщений WM_PAINT*/
void OnPaint(HWND hwnd){
  char szText[]="Строка текста для вывода в главное окно";
  PAINTSTRUCT ps;//Структура, требуемая для рисования в рабочей области
  HDC hdc=BeginPaint(hwnd,&ps);//Получение контекста устройства
  TextOut(hdc,5,30,szText,strlen(szText));//Вывод строки текста с точки 5,30
  EndPaint(hwnd,&ps);//Освобождение контекста устройства
  }//Конец функции OnPaint()
```

Программа 5-1 по структуре совпадает с предыдущей (4-2), однако в нее внесены некоторые изменения и дополнения. Как и раньше, в ней имеется главная функция WinMain() с участком подготовительных действий и циклом обнаружения сообщений, а также оконная функция WndProc(). В главной функции имеется одно изменение: при регистрации класса окна для фона окна выбрана не светло-серая, а белая кисть (константа WHITE_BRUSH). На рис. 5.1 приведен результат работы программы.

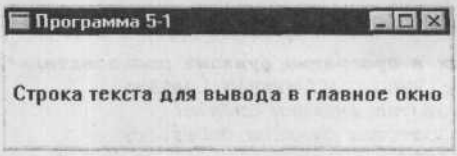


Рис. 5.1. Вывод в главное окно строки текста

Наиболее принципиальные отличия коснулись оконной функции. Ранее в ней обрабатывалось лишь одно сообщение WM_DESTROY. В настоящем примере прикладной

обработке подвергается также сообщение WM_PAINT. Цель обработки – вывод на экран заданной текстовой строки. Каждый раз, когда в приложение поступает сообщение WM_PAINT, эта строка заново выводится в главное окно приложения. В результате при любых манипуляциях с окном приложения – его сокращении или растягивании, сворачивании в значок или разворачивании на весь экран – наша строка всегда будет выводиться в одно и то же место главного окна.

Программный блок обработки сообщений WM_PAINT выделен в функцию OnPaint(). Соответственно в раздел прототипов включен прототип этой функции.

Рассмотрим более детально роль сообщения WM_PAINT и процедуру его обработки.

Обработка сообщений WM_PAINT

Сообщения WM_PAINT выделяются среди всех остальных тем, что их обработка включается практически в любое приложение Windows, если в нем хоть что-нибудь рисуется на экране. Общее правило рисования заключается в том, что вывод в окно приложения любых графических объектов – текстовых строк, геометрических фигур, отдельных точек, растровых изображений – должен выполняться *исключительно в процедуре обработки сообщения WM_PAINT*. Только в этом случае графическое содержимое окна не будет теряться при загораживании данного окна окнами других приложений. Каким образом вообще Windows поддерживает содержимое своих многочисленных окон? Здесь следует выделить несколько типичных ситуаций.

Если мы с помощью мыши или клавиатуры перемещаем окно приложения по экрану, то этот процесс не затрагивает само приложение. Копирование содержимого окна по мере его перемещения на новые места экрана обеспечивают системные программы Windows. Следовательно, о сохранении содержимого окна в этом случае можно не думать.

Если в приложении имеется линейка меню, то при разворачивании пунктов меню они перекрывают часть окна и, следовательно, при их сворачивании заслоненную ранее область окна надо перерисовать. Эту задачу Windows также берет на себя, сохраняя в своей памяти заслоняемое изображение и выводя его на экран при сворачивании меню.

По-другому обстоит дело, если перерисовка окна потребовалась в результате разворачивания окна, свернутого ранее в пиктограмму, а также при растягивании ранее сжатого окна или при перемещении по пространству главного окна порожденного окна диалога. В этих случаях, когда поврежденной может оказаться значительная часть окна или даже окно целиком (как при разворачивании пиктограммы), Windows уже не берет на себя задачу сохранения и восстановления изображения в окне, а вместо этого посылает в приложение сообщение WM_PAINT. Программа в ответ на сообщение WM_PAINT должна сама восстановить все, что должно изображаться в окне. При этом Windows сообщает в приложение, какая часть окна требует перерисовки, и приложение в принципе может перерисовывать только поврежденную часть окна, что заметно сократило бы временные издержки на вывод изображения. Практически, однако, такой алгоритм рисования составить слишком сложно, если вообще возможно, и в ответ на сообщение WM_PAINT программа вынуждена заново рисовать все, что должно изображаться в окне.

Главное окно приложения обычно имеет заголовок с управляющими кнопками и толстую рамку, а также во многих случаях линейку меню. Все эти элементы окна образуют нерабочую область окна (nonclient area, неклиентская область) и обычно недоступны программе. Остальная часть окна, куда программа может выводить что угодно, называется рабочей областью (client area, область клиента) (рис. 5.2). Восстановление нерабочей

области при любых манипуляциях с окном Windows берет на себя, программа обязана восстанавливать только рабочую область.

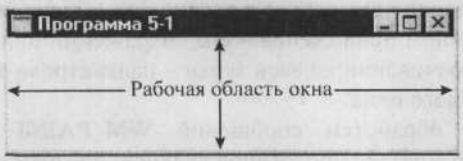


Рис. 5.2. Рабочая область окна

Обработка сообщения WM_PAINT связана с использованием важнейшего поля данных Windows, называемого *контекстом устройства*.

Контекст устройства представляет собой системную область памяти, закрепляемую за рабочей областью окна, в которой хранятся текущие значения режимов, связанных с рисованием, а также дескрипторы инструментов рисования – кисти, пера, шрифта и пр. Все графические функции GDI используют контекст устройства для определения режима рисования и характеристик применяемых ими инструментов. Например, функция вывода линии получает из контекста устройства (через дескриптор пера) толщину и цвет пера, которым должна быть нарисована линия; функции вывода геометрических фигур, в добавление к характеристикам пера, получают цвет кисти для закрашивания (заливки) рисуемых фигур; функции вывода текста получают (через дескриптор шрифта) все необходимые характеристики шрифта – гарнитуру, размер, цвет, насыщенность (жирность) и пр. Сам же контекст устройства становится известен графическим функциям GDI через *дескриптор контекста*, который для всех этих функций служит первым параметром.

Контекст устройства относится к числу системных ресурсов, количество которых в системе может быть ограничено; работа с такого рода ресурсами всегда протекает одинаково: сначала надо получить у системы требуемый ресурс, а закончив работу с ним, вернуть его системе.

Таким образом, для того чтобы вывести в окно некоторое изображение, необходимо выполнить следующие действия, последовательность которых и определяет алгоритм обработки сообщения WM_PAINT:

- получить у системы контекст устройства для данного окна;
- изменить при необходимости режимы рисования или характеристики конкретных инструментов;
- сформировать с помощью графических функций GDI требуемое изображение;
- вернуть Windows занятый у нее контекст устройства, приведя его предварительно в исходное состояние.

Все перечисленные действия выполняются в функции OnPaint().

Для получения контекста устройства предусмотрена функция BeginPaint(), требующая указания при ее вызове двух аргументов. Первый аргумент представляет собой дескриптор того окна, в котором мы предполагаем рисовать и для которого требуется контекст устройства. Второй аргумент – это адрес структурной переменной типа PAINTSTRUCT, которую функция BeginPaint() заполняет некоторыми данными. В случае своего успешного выполнения функция BeginPaint() возвращает дескриптор контекста устройства, который имеет тип HDC (Handle of Device Context). В нашей программе дескриптор контекста устройства поступает в переменную hdc.

В рассматриваемом простом примере изображение на экране представляет собой просто строку текста; для вывода строки используется функция GDI TextOut(), которая в качестве первого аргумента требует указания дескриптора контекста устройства. Поскольку никакие элементы контекста устройства в программе не изменяются, вывод строки осуществляется шрифтом, действующим по умолчанию. Способы настройки контекста устройства, так же как и другие графические функции GDI, будут рассмотрены ниже.

Возврат контекста в Windows осуществляется функцией EndPaint(), использующей те же аргументы, что и функция BeginPaint().

Функции BeginPaint() и EndPaint() используют структурную переменную (в нашем случае ps) типа PAINTSTRUCT. В то же время в программе она никак не используется. Что содержит эта структура?

Структура PAINTSTRUCT, заполняемая Windows каждый раз, когда приложение перехватывает обработку сообщения WM_PAINT, описана в файле WINUSER.H и содержит следующие элементы:

```
typedef struct tagPAINTSTRUCT {
  HDC  hdc;//Дескриптор выделяемого контекста устройства
  BOOL fErase;//Флаг перерисовки фона окна
  RECT rcPaint;//Область вырезки
  BOOL fRestore;//Зарезервировано
  BOOL fIncUpdate;//Зарезервировано
  BYTE rgbReserved[32];//Зарезервировано для использования Windows
  } PAINTSTRUCT;
```

Элемент структуры hdc – это тот же самый дескриптор контекста устройства, который служит для функции BeginPaint() возвращаемым значением.

Флаг перерисовки окна обычно равен нулю. Если, однако, в процессе регистрации класса окна не определить элемент wc.hbrBackground в структуре WNDCLASS, т. е. не задать кисть для закрашивания фона окна, функция BeginPaint() заполнит поле ps.fErase ненулевым значением. Для программы это будет означать, что она должна сама закрашивать окно, которое иначе будет прозрачным. Практически такой режим используется редко.

Пожалуй, наиболее важным элементом структуры PAINTSTRUCT (правда, не в практическом, а в принципиальном плане) является *область вырезки* rcPaint, которая сама представляет собой структуру типа RECT, служащую для описания прямоугольной области:

```
typedef struct tagRECT {
  int left;//x-координата левого верхнего угла прямоугольника
  int top;//y-координата левого верхнего угла прямоугольника
  int right;//x-координата правого нижнего угла прямоугольника
  int bottom;//y-координата правого нижнего угла прямоугольника
  } RECT;
```

Переменные типа RECT чрезвычайно широко используются в программах для Windows, поскольку области экрана, с которыми имеет дело Windows, всегда имеют прямоугольную форму. В данном случае переменная ps.rcPaint описывает ту область окна, которая в процессе манипуляций с окном была повреждена и требует перерисовки. Рассмотрим этот вопрос более подробно.

Как уже отмечалось, Windows посылает в приложение (точнее говоря, в окно приложения, в данном случае в главное окно, поскольку других окон в нашем приложении пока нет) сообщение WM_PAINT, во-первых, при первичном создании окна и, во-вторых, всякий раз, когда все окно или его часть, закрытая ранее другим объектом на экране, ос-

вождается и, соответственно, требует перерисовки. Мы также отмечали, что в принципе приложение обязано перерисовать только эту поврежденную часть окна, однако такой алгоритм рисования реализовать очень трудно, и практически всегда окно перерисовывается целиком, хотя на это понапрасну уходит драгоценное процессорное время. В то же время Windows предоставляет нам возможность повысить эффективность программы, передавая при каждом вызове функции BeginPaint() координаты области вырезки. Что представляет собой эта область?

Координаты области вырезки определяются относительно начала рабочей области окна. В программе 5-1 размеры всего окна составляют 300x100 пикселов, а размеры рабочей области оказываются (за вычетом толщины рамки и ширины линейки заголовка) меньше, а именно 292x73 пиксела. При обработке первого сообщения WM_PAINT функция BeginPaint() передает в программу через переменную ps.rcPaint именно эти координаты:

```
ps.rcPaint={0,0,292,73}
```

Чтобы убедиться в этом, проведите такой эксперимент. Запустите приложение 5-1 в отладчике (выбором в среде IDE Borland C++ команды меню Tool➢Turbo Debugger) и установите курсор на строке

```
EndPaint(hwnd,&ps);//Освобождение контекста устройства
```

т. е. после вывода в окно текстовой строки, но еще в пределах функции OnPaint(), где известны значения всех локальных переменных этой функции. Запустите выполнение программы до курсора (клавиша F4). Программа остановится, что будет соответствовать первому поступлению в окно сообщения WM_PAINT. Для вывода на экран содержимого структуры ps следует поставить курсор на имя ps и выбрать команду меню Data➢Inspect. В окне просмотра данных можно будет увидеть значения переменных элемента структуры rcPaint, которые должны составить последовательность {0,0,292,73} (рис. 5.3).

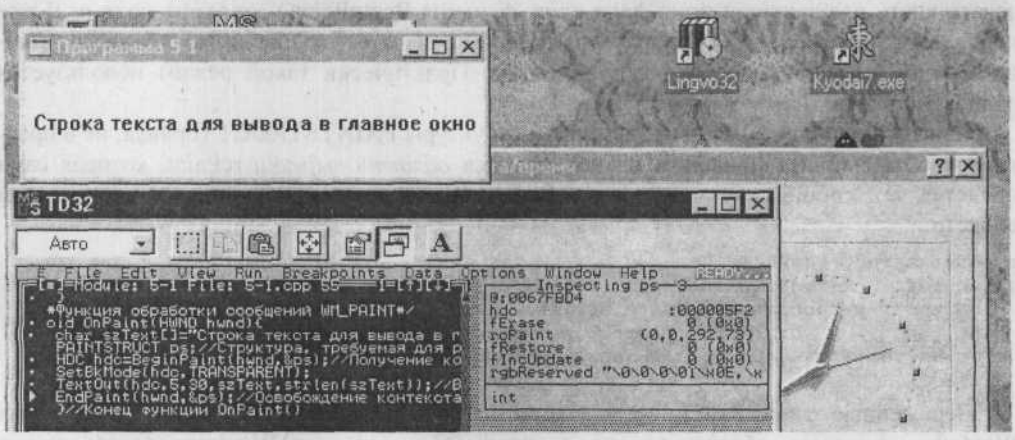


Рис. 5.3. Окно приложения на переднем плане

Продолжите выполнение программы до того же предложения, нажав еще раз клавишу F4. Закройте часть окна приложения каким-либо другим окном, например окном приложения даты/времени (окно этого приложения лучше заранее установить так, чтобы оно частично перекрывало окно программы 5-1, как это показано на рис. 5.3). В результате часть нашего окна скроется (рис. 5.4).

```
    DispatchMessage(&msg);
  return 0;
  }
/*Оконная процедура WndProc главного окна*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_CREATE,OnCreate);
    HANDLE_MSG(hwnd,WM_PAINT,OnPaint);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
/*Функция обработки сообщения о создании окна*/
BOOL OnCreate(HWND hwnd,LPCREATESTRUCT){
  hBitmap=CreateCompatibleBitmap(hdc,450,400);//Создаем совместимую память
  HDC hdc=GetDC(hwnd);//Получим контекст устройства
  hdcMem=CreateCompatibleDC(hdc);//Создаем совместимый контекст памяти
  SelectBitmap(hdcMem,hBitmap);//Выбираем память в совместимый контекст
  PatBlt(hdcMem,0,0,450,400,WHITENESS);//Закрасим память белым
  ReleaseDC(hwnd,hdc);//Освободим контекст устройства
  return TRUE;
  }
/*Функция OnPaint обработки сообщения WM_PAINT*/
void OnPaint(HWND hwnd){
  PAINTSTRUCT ps;
  HDC hdc=BeginPaint(hwnd,&ps);
  BitBlt(hdc,0,0,450,400,hdcMem,0,0,SRCCOPY);//Копирование памяти в окно
  EndPaint(hwnd,&ps);
  }
/*Функция OnDestroy обработки сообщения WM_DESTROY*/
void OnDestroy(HWND){
  DeleteBitmap(hBitmap);//Освобождаем совместимую память
  DeleteDC(hdcMem);//Удаляем контекст памяти
  PostQuitMessage(0);
  }
/*Функция рисования изображения и вывода его в растр в памяти*/
void Draw(HWND hwnd){
  int Radius=45;
  HDC hdc=GetDC(hwnd);//Только для демонстрации
  for(double Phi=100;Phi>0.1;Phi-=0.001){
    int x=Radius*cos(Phi)/(0.002*Phi+0.2);
    int y=Radius*sin(Phi)/(0.01*Phi+0.2);
    SetPixel(hdc,x+220,y+180,RGB(255,0,0));//Рисуем в окне (для демонстрации)
    SetPixel(hdcMem,x+220,y+180,RGB(0,0,255));//Рисуем в памяти
    }
  ReleaseDC(hwnd,hdc);
  }
```

Если в программе 9-2 все действия по вычислению формы изображения и собственно выводу осуществлялись в функции OnPaint(), то теперь процедура образования изображения в окне распадается на две отдельные задачи: вычисление и формирование изображения в совместимой области памяти, что можно выполнить в любом удобном месте программы (и кстати, параллельно с другими процессами), и перенос уже сформированного изображения, т. е. фактически образа окна, из совместимой памяти в окно, что должно выполняться в функции обработки сообщения WM_PAINT.

Процедура формирования изображения в рассматриваемом примере для повышения наглядности программы выделена в отдельную подпрограмму Draw(). Эта функция, к которой мы еще вернемся, вызывается в данном случае в главной функции WinMain() по-

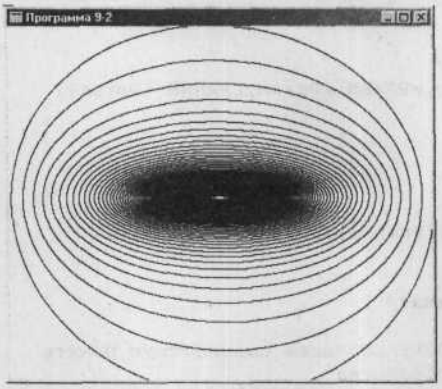


Рис. 9.11. Изображение математической кривой, вычисляемой в процессе вывода на экран

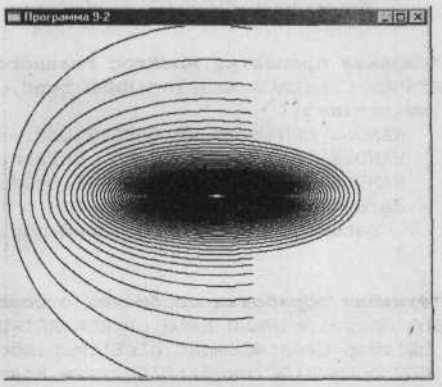


Рис. 9.12. "Недорисованное" изображение после его частичного затирания

Использование совместимой памяти

Рассмотрим теперь усложненный вариант предыдущего примера, в котором рисование кривой осуществляется не непосредственно в окне, а в совместимой области памяти с последующим переносом в окно функцией BitBlt().

```
/*Программа 9-3. Формирование изображения в совместимой памяти*/
//Файл 9-3.h
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
BOOL OnCreate(HWND,CREATESTRUCT FAR*);
void OnPaint(HWND);
void OnDestroy(HWND);
void Draw(HWND);//Прикладная функция формирования изображения
//Файл 9-3.cpp
/*Операторы препроцессора*/
#include <windows.h>
#include <windowsx.h>
#include <math.h>
#include "9-3.h"
/*Глобальная переменная*/
HDC hdcMem;//Дескриптор совместимого контекста
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  char szClassName[]="MainWindow";
  char szTitle[]="Программа 9-3";
  MSG msg;
  WNDCLASS wc;
  memset(&wc,0,sizeof(wc));
  wc.lpfnWndProc=WndProc;
  wc.hInstance=hInst;
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
  wc.lpszClassName=szClassName;
  RegisterClass(&wc);
  HWND hwnd=CreateWindow(szClassName,szTitle,WS_OVERLAPPEDWINDOW,
                         0,0,450,400,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_SHOWNORMAL);
  Draw(hwnd);//Вызовем функцию формирования изображения
  while(GetMessage(&msg,NULL,0,0))
```

```
  wc.lpszClassName=szClassName;
  RegisterClass(&wc);
  HWND hwnd=CreateWindow(szClassName,szTitle,WS_OVERLAPPEDWINDOW,
                         0,0,450,400,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_SHOWNORMAL);
  while(GetMessage(&msg,NULL,0,0))
    DispatchMessage(&msg);
  return 0;
  }
/*Оконная процедура WndProc главного окна*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_PAINT,OnPaint);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
/*Функция OnPaint обработки сообщения WM_PAINT*/
void OnPaint(HWND hwnd){
  PAINTSTRUCT ps;
  HDC hdc=BeginPaint(hwnd,&ps);//Получим контекст устройства
  int Radius=45;
  for(double Phi=2000;Phi>0.1;Phi-=0.001){
    int x=Radius*cos(Phi)/(0.002*Phi+0.2);//Вычисляем координаты
    int y=Radius*sin(Phi)/(0.01*Phi+0.2);//кривой типа спирали
    SetPixel(hdc,x+220,y+180,RGB(0,0,128));//Выводим точку на экран
    }
/*Функция OnDestroy обработки сообщения WM_DESTROY*/
void OnDestroy(HWND){
  PostQuitMessage(0);
  }
```

Замедление работы программы при перерисовке окна можно обнаружить, только если формирование изображения требует значительного времени, по меньшей мере нескольких секунд. В качестве такого изображения мы выбрали выводимую на экран по точкам математическую кривую типа спирали, которая записывается в параметрической форме следующим образом:

$$x=R\cos\varphi/(0.002\varphi+0.2);$$
$$y=R\sin\varphi/(0.01\varphi+0.2).$$

Вычисление координат точек кривой и вывод их на экран выполняется в цикле for с маленьким шагом (в примере – 0.001) и большим числом шагов (2×10^6), что увеличивает время выполнения вывода до ощутимой величины. При использовании быстрого процессора целесообразно сделать шаг еще меньше. На рис. 9.11 показан результат работы программы 9-2.

Отладив программу, понаблюдайте ее реакцию на изменение размеров окна. Любое, даже самое незначительное увеличение размеров окна приводит к посылке в приложение сообщения WM_PAINT и выполнению *всего* цикла формирования изображения. При этом, хотя процедуры GDI выполняют фактическое рисование только в открывшейся вновь области окна, что, разумеется, сокращает время вывода, наш алгоритм вычисления формы кривой выполняется всегда целиком, что и определяет недопустимо медленную работу приложения. На рис. 9.12 показано, как все изображение начинает рисоваться заново после увеличения размеров частично свернутого окна.

зволяет залить указанную в ее параметрах прямоугольную область любой кистью (не обязательно белой):

```
RECT rect;//Структура,описывающая прямоугольник
rect.left=rect.top=0;//Зальем всю совместимую память
rect.right=800;//размером 800x600 пикселов
rect.bottom=600;
HBRUSH hBrush=CreateSolidBrush(RGB(180,250,250));//Создадим кисть
FillRect(hdcMem,&rect,hBrush);//Заливаем совместимую память
```

В этом примере вся совместимая память закрашивается бледно-бирюзовой кистью. Другой способ закраски совместимой память будет продемонстрирован в следующем разделе.

Проблемы отображения вычисляемых математических функций

Рисование в окне приложения

В предыдущих главах неоднократно отмечалось, что практически весь вывод в окна приложения должен осуществляться в функциях обработки сообщений WM_PAINT, поступающих в эти окна. Только в этом случае обеспечивается восстановление содержимого окон при всяческих манипуляциях с ними – изменении размеров, перекрытии и пр. Однако такая методика имеет свою отрицательную сторону: при любом изменении размеров окна вызывается и выполняется функция OnPaint(), осуществляющая *полную* перерисовку содержимого окна, на что отвлекается время процессора. Особо серьезная проблема возникает в тех случаях, когда изображение, выводимое на экран, не является статическим, а формируется программным образом, представляя собой, например, график математической функции или иной результат каких-либо вычислений. Тогда в ответ на каждое сообщение WM_PAINT все эти вычисления проводятся в программе заново. Программа 9-2 позволяет наглядно проиллюстрировать обсуждаемую проблему.

```
/*Программа 9-2. Вывод математической кривой в процедуре
  обработки сообщения WM_PAINT. Низкая скорость перерисовки*/
/*Файл 9-2.h*/
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
void OnPaint(HWND);
void OnDestroy(HWND);
```

```
//Файл 9-2.cpp
/*Операторы препроцессора*/
#include <windows.h>
#include <windowsx.h>
#include <math.h>//Файл с прототипами математических функций
#include "9-2.h"
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  char szClassName[]="MainWindow";
  char szTitle[]="Программа 9-2";
  MSG msg;
  WNDCLASS wc;
  ZeroMemory (&wc,sizeof(wc));
  wc.lpfnWndProc=WndProc;
  wc.hInstance=hInst;
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
```

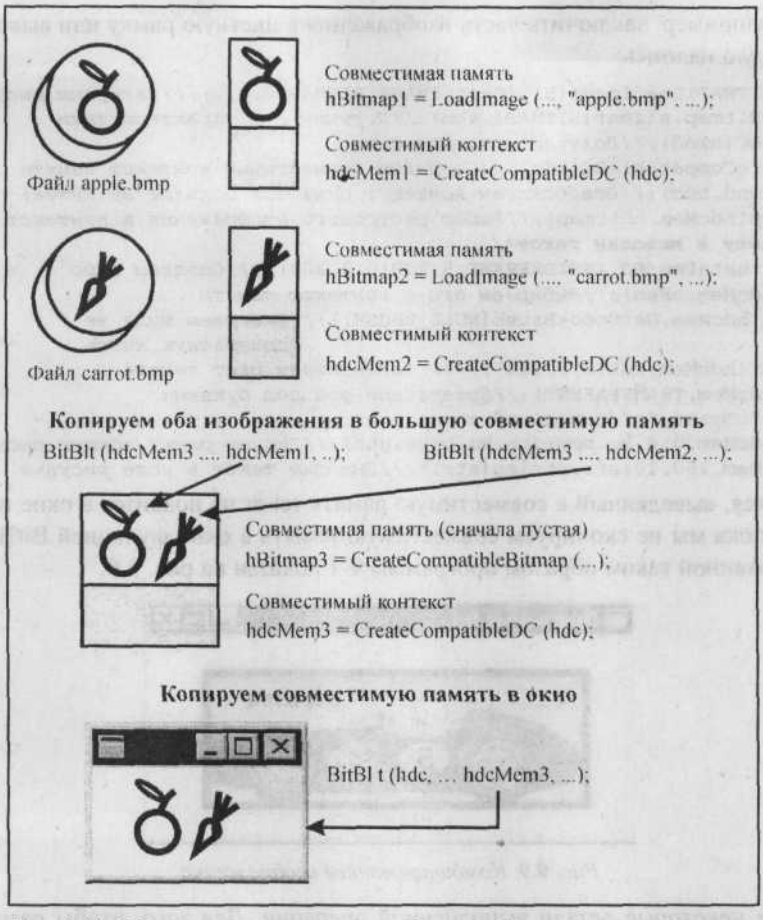


Рис. 9.10. Объединение нескольких изображений в совместимой памяти и их копирование в окно приложения

Однако для выполнения такой процедуры необходимо иметь пустую совместимую память требуемого размера. Для получения такой памяти предусмотрена функция CreateCompatibleBitmap():

```
HBITMAP hBitMap=CreateCompatibleBitmap(hdc,800,600);
```

В этом примере создается совместимая память размером во весь экран (800×600 пикселов). Заметим, что для создания совместимой памяти таким способом так же, как и при создании совместимого контекста, требуется иметь контекст окна hdc. Поскольку пустая совместимая память обычно создается заранее, в функции OnCreate(), контекст hdc получают вызовом функции GetDC().

При создании пустой совместимой памяти с помощью функции CreateCompatibleBitmap() следует иметь в виду, что реально эта память отнюдь не пуста, а может быть заполнена любым "мусором". Если затем эта память заполняется копируемым в нее изображением целиком, то "мусор", разумеется, затирается. Если, однако, графические объекты выводятся в отдельные участки большого блока совместимой памяти, ее следует предварительно очистить. Это можно сделать с помощью функции FillRect(), которая по-

жем далее, например, заключить часть изображения в цветную рамку или вывести поверх него некоторую надпись:

```
hBitmap=(HBITMAP)LoadImage(hI,"Car",IMAGE_BITMAP,0,0,0);//Загрузим рисунок
GetObject(hBitmap,sizeof(BITMAP),&bm);//Получим его характеристики
HDC hdc=GetDC(hwnd);//Получим контекст окна
hdcMem=CreateCompatibleDC(hdc);//Создадим совместимый контекст памяти
ReleaseDC(hwnd,hdc);//Освобождаем контекст окна (он больлше не нужен)
SelectBitmap(hdcMem,hBitmap);//Выбор растрового изображения в контекст памяти
/*Рисуем рамку и выводим текст*/
HPEN hPen=CreatePen(PS_INSIDEFRAME,5,RGB(0,0,200));//Создаем перо
SelectPen(hdcMem,hPen);//Выбираем его в контекст памяти
SelectBrush(hdcMem,GetStockBrush(NULL_BRUSH));//Выбираем туда же
                                              //прозрачную кисть
SetTextColor(hdcMem,RGB(0,0,200));//Устанавливаем цвет текста
SetBkMode(hdcMem,TRANSPARENT);//Прозрачный фон под буквами
char str[]="Образец";//Выводимый текст
Rectangle(hdcMem,0,0,bm.bmWidth,bm.bmHeight);//Рисуем рамку вокруг рисунка
TextOut(hdcMem,150,10,str,strlen(str));//Выводим текст в поле рисунка
```

Разумеется, выведенный в совместимую память текст не появится в окне приложение до тех пор, пока мы не скопируем совместимую память в окно функцией BitBlt(). Вывод модифицированной таким образом программы 9-1 показан на рис. 9.9.

Рис. 9.9. Комбинированное изображение

Отметим некоторые детали выполненной операции. Для того, чтобы рамка выводилась не симметрично границ рисунка, а внутри этих границ, перо, которым она рисуется, и которое загружается в контекст памяти, создано с атрибутом PS_INSIDEFRAME. Далее, чтобы сквозь рамку был виден рисунок, в контекст памяти выбрана прозрачная кисть. Наконец, для размещения рамки точно по границе рисунка, в качестве размеров выводимого прямоугольника использованы размеры изображения bm.bmWidth и bm.bmHeight.

Часто возникает необходимость размещения в окне нескольких изображений либо вывод небольшого рисунка поверх фонового изображения, которое само хранится в файле .BMP и должно быть сначала скопировано в окно. В таких случаях создается несколько областей совместимой памяти и, соответственно, несколько совместимых контекстов. В каждую область совместимой памяти загружается свое изображение, а затем эти области либо по отдельности копируются в окно приложения, либо (что лучше) сначала копируются в отдельную область совместимой памяти большого размера, а та уже копируется в окно (рис. 9.10).

Таблица 9.1. Константы растровых операций

Константа	*Описание действия*
SRCCOPY	Копирование источника (область копирования перед этим затирается)
BLACKNESS	Область копирования закрашивается черным цветом
WHITENESS	Область копирования закрашивается белым цветом
DSTINVERT	Изображение в области копирования инвертируется
NOTSRCCOPY	Копирование инвертированного источника (область копирования перед этим затирается)
SRCINVERT	В область копирования записывается результат операции XOR над пикселами источника и приемника
SRCPAINT	В область копирования записывается результат операции OR над пикселами источника и приемника
MERGEPAINT	В область копирования записывается результат операции OR над пикселами инвертированного источника и пикселами приемника

Описанные операции позволяют, в частности, реализовать движущиеся изображения – мультипликацию. Один из приемов получения движущегося объекта заключается в том, что через некоторое время после вывода изображения оно стирается и тут же выводится с небольшим сдвигом по поверхности окна. Посмотрим, как это делается.

Если вывести в окно некоторое изображение с помощью операции SRCCOPY, а через некоторое время вывести то же изображение еще дважды, один раз с помощью операции NOTSRCCOPY, а затем с операцией SRCPAINT, то исходное изображение будет стерто. Действительно, операция NOTSRCCOPY заменяет исходное изображение инверсным, а операция SRCPAINT (OR) над прямым (источник) и инверсным (приемник) изображениями дает белый фон.

Можно поступить проще, воспользовавшись для стирания одной операцией MERGEPAINT. Она инвертирует источник "внутри себя", и, если приложить ее к прямому изображению, результат будет такой же, как в предыдущем примере, – операция OR над инверсным (источник) и прямым (приемник) изображениями по-прежнему дает белый фон.

Наконец, имеется и третий, пожалуй, самый изящный способ. В нем используется файл не с прямым, а с инвертированным изображением, а вывод осуществляется с помощью последовательных операций SRCINVERT. Первое копирование (на чистый экран) *инверсного* изображения с помощью этой операции приведет к появлению на экране *прямого* изображения. Второй вывод на то же место сотрет изображение (операция XOR над прямым и инверсным изображениями). Следующий вывод (со сдвигом и опять на чистый экран) снова приведет к появлению прямого изображения и т. д.

Компоновка составных изображений

Выше уже отмечалось, что в совместимом контексте, как и в любом контексте устройства, имеются дескрипторы графических инструментов вместе с некоторыми их характеристиками (например, цветом шрифта или фона под ним). Это дает возможность рисовать с помощью этих инструментов в совместимой памяти, как в обычном окне. Выбрав в совместимый контекст дескриптор изображения, загруженного из файла, мы мо-

Хотя с памятью, в которой находится изображение, можно работать, однако она пока еще не отображается на экране. Чтобы увидеть растровое изображение, его надо скопировать в окно приложения с помощью функций BitBlt() или StrechBlt().

Использование функции BitBlt()

Функция BitBlt() позволяет не только просто копировать изображение, но и выполнять с ним некоторые преобразования. Рассмотрим возможности этой функции чуть подробнее.

Вызов функции BitBlt() в общем виде можно записать следующим образом:

```
BitBlt(hdc,x,y,w,h,hdcMem,x0,y0,ROP);
```

Параметры hdc и hdcMem представляют собой дескрипторы контекстов приемника и источника копируемого изображения. В данном случае мы копируем из совместимой памяти (дескриптор hdcMem) в окно приложения (дескриптор hdc). Параметры x, y, w, h, x0, и y0 являются геометрическими характеристиками; их назначение видно из рис. 9.7.

Параметры x0 и y0 задают координаты начальной точки копирования в изображении-источнике. Наиболее естественно положить x0=y0=0, чтобы копировать изображение от самого его начала, но можно и обрезать часть изображения, как это сделано на рис. 9.7. Параметры x и y задают положение копии в окне-приемнике (от начала рабочей области окна), а параметры w и h – размер копируемой части изображения. Если w=bmWidth, а h=bmHeight, то копируется все изображение целиком, но можно скопировать и его часть, как это опять же проиллюстрировано на рис. 9.8.

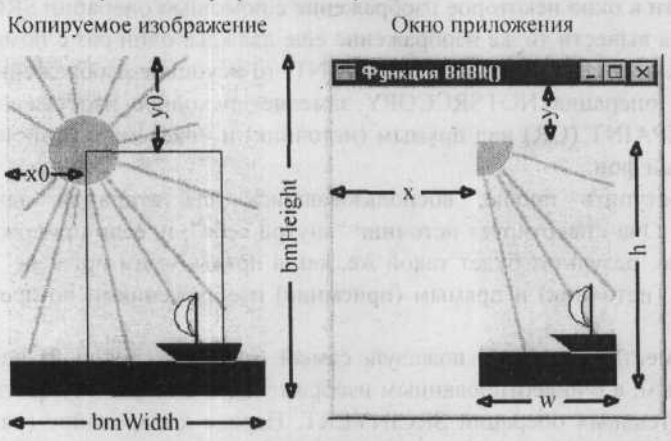


Рис. 9.8. Геометрические характеристики копируемого изображения и его копии в окне

Функция BitBlt() позволяет не только выводить в окна приложения нарисованные заранее растровые изображения, но и закрашивать (и раскрашивать) область копирования, а также выполнять над битами изображения и битами области копирования ряд логических операций (НЕ, ИЛИ и И). С их помощью можно, в частности, стирать выведенное ранее изображение. Вид операции, выполняемой над изображением (вид растровой операции) определяется значением последнего параметра функции BitBlt() ROP.

Значения параметра ROP задаются символическими константами, определенными в файле WINGDI.H. Всего предусмотрено 15 таких констант; некоторые из них приведены в табл. 9.1.

мы получим размеры изображения в виде значений элементов структуры bm.bmWidth (ширина) и bm.bmHeight (высота). Поскольку мы получаем характеристики растра в функции OnCreate(), а использовать их будем в функции OnPaint(), структурная переменная bm типа BITMAP объявлена глобальной.

Хотя изображение попало в память, в этом, можно сказать, нет пока никакого прока, так как для обращения к изображению необходимо иметь совместимый контекст этой области памяти.

Совместимый контекст создается функцией CreateCompatibleDC() совершенно независимо от загрузки изображения в память:

```
hdcMem=CreateCompatibleDC(hdc);
```

Единственный аргумент этой функции – дескриптор контекста того окна, с которым должен быть совместим создаваемый контекст памяти (вспомним, что другая функция получения контекста устройства, BeginPaint(), возвращает еще и область вырезки, для чего ей требуется структурная переменная типа PAINTSTRUCT). Создать совместимый контекст можно как после загрузки изображения в память, так и до этой операции, так как пока эти две области памяти друг к другу отношения не имеют и никак не взаимодействуют.

В примере 9-1 создание совместимого контекста осуществляется в той же функции OnCreate(). Поскольку для этой операции необходим обычный контекст устройства, мы сначала получаем его с помощью функции GetDC(), а затем создаем совместимый с ним контекст памяти функцией CreateCompatibleDC(). После этого обычный контекст можно удалить (точнее, вернуть системе), что мы и делаем с помощью функции ReleaseDC(). Переменную hdcMem с дескриптором совместимого контекста мы будем использовать в функции OnPaint(), поэтому она объявлена глобальной.

Казалось бы, полученным дескриптором совместимого контекста памяти уже можно пользоваться для вывода в память текстов или геометрических фигур. Практически, однако, сделать это никак нельзя, так как в совместимом контексте по умолчанию находится дескриптор растрового изображения размером всего в 1 пиксел, в которое, конечно, ничего не выведешь. Поэтому следующим обязательным шагом является выбор в совместимый контекст дескриптора нашей совместимой памяти с замещением дескриптора растра по умолчанию (того самого единственного пиксела):

```
SelectBitmap (hdcMem,hBitmap);
```

Перед завершением программы надо будет удалить совместимый контекст памяти hdcMem, а также саму совместимую память, характеризуемую ее дескриптором hBitmap. Эти действия выполняются в функции OnDestroy(), причем для удаления совместимого контекста предусмотрена функция DeleteDC(), а совместимую память можно удалить упоминавшейся уже обобщенной функцией DeleteObject() или макросом DeleteBitmap().

После того как в совместимый контекст выбран дескриптор совместимой памяти, контекст становится связан с реально существующей памятью, в которой находится наше растровое изображение. Теперь на это изображение можно при необходимости накладывать тексты или другие геометрические образы с помощью любых подходящих функций GDI, например TextOut(), Rectangle() или SetPixel(). Любопытно отметить, что рисование в совместимой памяти любых объектов можно выполнить в любом месте программы, в частности в той же функции OnCreate(). Эти действия выполняются в оперативной памяти и не имеют отношения к перерисовыванию окна приложения.

ведет к образованию совместимой памяти указанного размера, которая целиком будет занята загружаемым рисунком с изменением (скорее всего, неправильным) его масштаба (рис. 9.7).

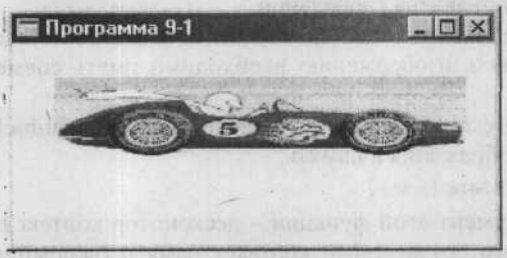


Рис. 9.7. Загрузка рисунка с неправильным указанием размеров совместимой памяти

Функция LoadImage() позволяет загружать не только растровые изображения, но также курсоры и пиктограммы. Во всех случаях функция должна вернуть дескриптор созданного объекта. Поскольку дескрипторы этих графических объектов принадлежат к разным типам данных (HBITMAP, HCURSOR и HICON), функция LoadImage() возвращает обобщенный дескриптор типа HANDLE, который, в сущности, является вариантом типа void*, т. е. указателя на любой тип данных. Поэтому при использовании функции LoadImage() возвращаемое ею значение следует привести к типу загружаемого объекта. В нашем случае это тип HBITMAP.

Во втором варианте, когда файл ресурсов отсутствует, и изображение загружается непосредственно из файла .BMP, функция LoadImage() используется в следующем формате:

```
HBITMAP hBitmap = (HBITMAP)LoadImage (NULL,"apple.bmp",
                                IMAGE_BITMAP, 0,0, LR_LOADFROMFILE);
```

Здесь вместо дескриптора экземпляра приложения указывается NULL, а вместо имени ресурса – имя самого файла с изображением. Кроме того, необходимо указать флаг LR_LOADFROMFILE, определяющий загрузку из файла. Как и ранее, возвращаемое значение следует явно привести к типу HBITMAP. Высказанное выше замечание об указании размеров совместимой памяти справедливо и для этого случая.

В программе 9-1 загрузка изображения в память осуществляется на этапе создания главного окна, в функции OnCreate(). Загруженное изображение может иметь любой размер, который в дальнейшем, при копировании изображения в окно, нам понадобится. Для получения характеристик графических объектов Windows предусмотрена функция GetObject(), которая возвращает информацию о запрошенном объекте в соответствующую этому объекту структуру. Для объекта-растра используется структура типа BITMAP:

```
typedef struct tagBITMAP {
  int     bmType;//Тип битовой матрицы
  int     bmWidth;//Ширина в пикселах
  int     bmHeight;//Высота в пикселах
  int     bmWidthBytes;//Число байтов в строке растра
  BYTE    bmPlanes;//Число цветовых плоскостей
  BYTE    bmBitsPixel;//Число цветовых битов на 1 пиксел
  void FAR* bmBits;//Не используется
  } BITMAP;
```

Вызвав функцию

```
GetObject(hBitmap,sizeof(BITMAP),&bm);
```

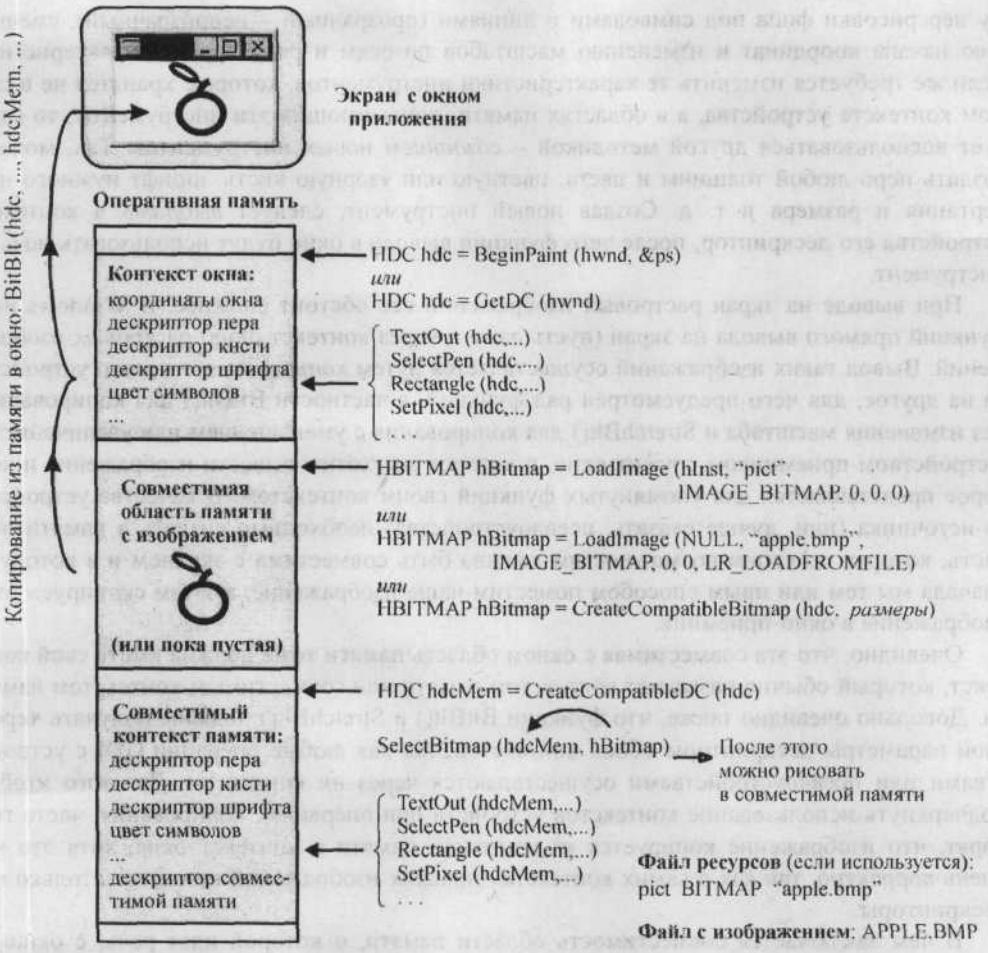


Рис. 9.6. Взаимодействие программы и контекстов при выводе растрового изображения

Процедура вывода растрового изображения

Как уже отмечалось, для загрузки изображения в память предусмотрена функция LoadImage(), которая используется в двух вариантах. Для загрузки изображения из выполнимого файла .EXE (т. е. посредством файла ресурсов) вызов функции выглядит следующим образом:

```
HBITMAP hBitmap = (HBITMAP)LoadImage (hInstance, "pict",
                                      IMAGE_BITMAP, 0, 0, 0);
```

Здесь hInstance – дескриптор экземпляра приложения, передаваемый в программы через первый параметр функции WinMain(), а "pict" – имя нашего ресурса в файле ресурсов. Последний параметр функции предназначен для указания флагов загрузки, которые в данном случае не нужны, а два параметра перед ним позволяют задать размеры создаваемой совместимой памяти. При нулевых значениях этих двух параметров выделяется столько памяти, сколько требуется для хранения изображения. Стоит заметить, что указание в этих параметрах конкретных чисел, не совпадающих с размерами рисунка, при-

му перерисовки фона под символами и линиями (прозрачный – непрозрачный), смещению начала координат и изменению масштабов по осям и ряду других характеристик. Если же требуется изменить те характеристики инструментов, которые хранятся не в самом контексте устройства, а в областях памяти, описывающих эти инструменты, то следует воспользоваться другой методикой – *созданием* новых инструментов. Так, можно создать перо любой толщины и цвета, цветную или узорную кисть, шрифт нужного начертания и размера и т. д. Создав новый инструмент, следует *выбрать* в контекст устройства его дескриптор, после чего функции вывода в окно будут использовать новый инструмент.

При выводе на экран растровых изображений все обстоит сложнее. В Windows нет функций прямого вывода на экран (пусть даже и через контекст окна) растровых изображений. Вывод таких изображений осуществляется путем *копирования* с одного устройства на другое, для чего предусмотрен ряд функций, в частности BitBlt() для копирования без изменения масштаба и StretchBlt() для копирования с уменьшением или увеличением. Устройством-приемником служит окно, в которое мы хотим вывести изображение и которое представляется для упомянутых функций своим контекстом. В качестве устройства-источника (или, лучше сказать, псевдоустройства) необходимо создать в памяти область, которая по своим возможностям должна быть совместима с экраном и в которую сначала мы тем или иным способом поместим наше изображение, а затем скопируем это изображение в окно-приемник.

Очевидно, что эта совместимая с окном область памяти тоже должна иметь свой контекст, который обычно называют контекстом памяти или совместимым контекстом памяти. Довольно очевидно также, что функции BitBlt() и StretchBlt() должны получать через свои параметры дескрипторы обоих контекстов, так как любые операции GDI с устройствами или псевдоустройствами осуществляются через их контексты. Для того чтобы подчеркнуть использование контекстов устройств при операциях копирования, часто говорят, что изображение копируется из контекста памяти в контекст окна, хотя это не очень корректно, так как в самих контекстах никаких изображений нет, а есть только их дескрипторы.

В чем заключается совместимость области памяти, о которой идет речь, с окном? Контекст этой области памяти, если его создать, будет многим напоминать контекст окна. В нем также будут храниться характеристики и дескрипторы графических инструментов. Соответственно в эту область памяти можно выводить, используя обычные функции GDI, текст, точки и геометрические фигуры точно так же, как в окно. Более того, в этот контекст можно загружать (выбирать) новые инструменты, разумеется, предварительно создав их, и затем использовать для формирования требуемого графического кадра. Однако совместимый контекст имеет одно существенное отличие от обычного контекста устройства. В него можно выбрать, помимо обычных инструментов, еще и дескриптор растрового изображения. Выполнив эту операцию, мы получим то самое псевдоустройство-источник, из которого растровое изображение можно скопировать в окно приложения с помощью функций BitBlt() или StretchBlt().

На рис. 9.6 показаны все упоминавшиеся выше объекты, принимающие участие в процедуре вывода в окно приложения изображений, вместе с функциями, используемыми для создания или управления этими объектами. В дальнейших рассуждениях мы будем ссылаться на этот рисунок.

Контексты окна и совместимой памяти

Как известно, при выводе в окно приложения любых графических образов (строк текста, геометрических фигур, отдельных точек) необходимо использовать контекст устройства (фактически – контекст окна). В контексте, в частности, хранятся дескрипторы всех используемых графических инструментов – перьев, кистей и шрифтов.

Поскольку в контексте находятся характеристики, отражающие текущие свойства окна, в частности, координаты окна на Рабочем столе, контекст приходится получать заново каждый раз, когда мы приступаем к выводу в окно тех или иных графических объектов (текстов, линий, точек, рисунков и проч.). Если контекст нужен для рисования в функции обработки сообщения WM_PAINT, то для получения контекста окна следует воспользоваться функцией BeginPaint(), а для освобождения – функцией EndPaint(). Если же требуется получить контекст устройства в любом другом месте программы, не связанном с сообщением WM_PAINT, то контекст надо получать с помощью функции GetDC(), а освобождать функцией ReleaseDC().

На рис. 9.5 схематически показано взаимодействие программы и окна приложения через контекст этого окна. В контексте хранится вся необходимая для рисования в окне информация: координаты всего окна и области вырезки (для контекста, полученного функцией BeginPaint()), дескрипторы таких графических инструментов, как перо, кисть и шрифт, цвет символов и фона под ними, координаты текущей позиции, масштабах отображения по осям и т. д.

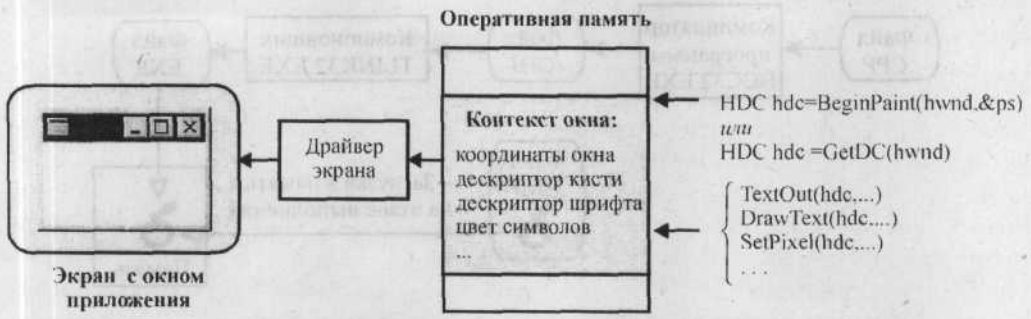


Рис. 9.5. Взаимодействие программы и контекста устройства при выводе в окно

Для рисования в окне программа обращается к функциям GDI, таким, как TextOut(), DrawText(), Rectangle(), Ellipse(), SetPixel() и т. д. Для изменения атрибутов, хранящихся в контексте, предусмотрены функции SetTextColor(), SetBkColor, SetBkMode() и многие другие. Все эти функции (и рисования и изменения атрибутов) в качестве первого аргумента требуют указания контекста устройства, так как они или извлекают из контекста какие-то данные, или, наоборот, изменяют данные, хранящиеся в контексте. Функции, предназначенные собственно для рисования, получив из контекста необходимую информацию и выполнив, возможно, требуемые вычисления (например, координат), вызывают драйвер Windows, управляющий работой физического устройства – экрана, который и выполняет вывод изображения на экран.

Как уже отмечалось выше, те характеристики инструментов, которые непосредственно хранятся в контексте окна, можно относительно просто изменять с помощью предусмотренных для этого функций. Это относится к цвету символов и фона под ними, режи-

Другой способ загрузки изображения в память не требует наличия файла ресурсов; загрузка изображения осуществляется той же функцией LoadImage() непосредственно из файла .BMP. Однако в этом случае приложение может выполняться, только если ему доступны все необходимые файлы с рисунками, поскольку в выполнимом файле рисунков нет. Легко сообразить, что именно такой метод используется в программах, позволяющих просматривать рисунки, хранящиеся на дисках. Оба описанных способа схематически изображены на рис. 9.4.

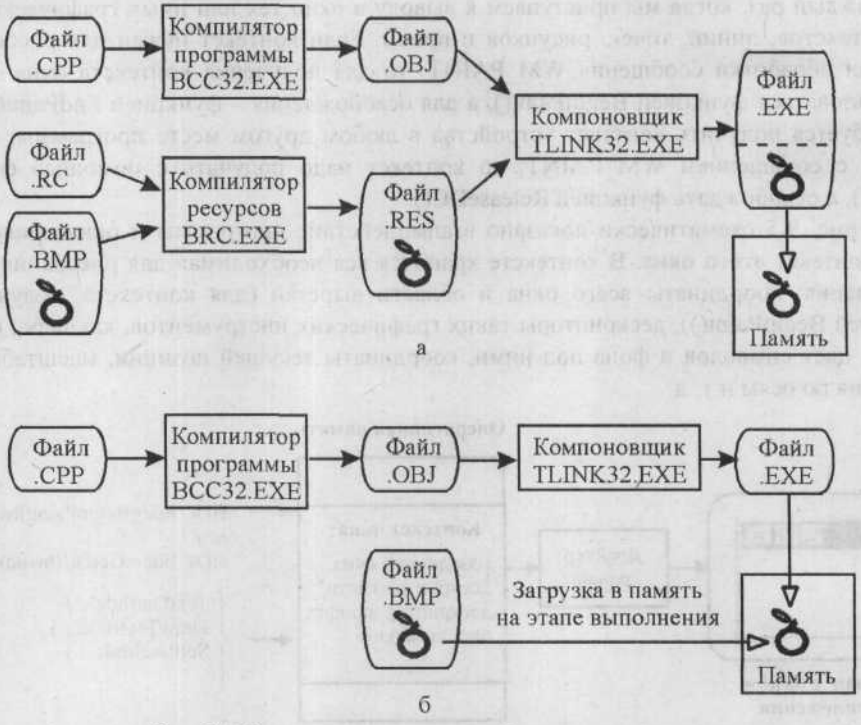


Рис. 9.4. Загрузка растрового изображения в память: а – из выполнимого файла посредством файла ресурсов, б – непосредственно из файла .BMP

Область памяти, куда загружается изображение, должна иметь особую организацию, совместимую с представлением на экране окна приложения. Такая память называется совместимой (compatible), и функция LoadImage(), загружая изображение из файла, одновременно создает область совместимой памяти соответствующего изображению размера. После этого нам остается только скопировать изображение из совместимой памяти в окно приложения, для чего предусмотрена функция BitBlt() (от Bit block transfer, блочная передача битов). Эта функция отличается высокой эффективностью, обеспечивая быстрый вывод на экран даже больших изображений. Как правило, копирование изображения из совместимой памяти в окно осуществляется в ответ на сообщение WM_PAINT, поскольку только в этом случае обеспечивается правильное восстановление содержимого окна после его частичного или полного затирания.

ние как в графической форме с помощью графических инструментов (перья, ластики, кисти и пр.), так и в форме таблицы чисел, что, впрочем, вряд ли целесообразно.

Небольшие изображения можно описывать непосредственно в памяти в виде таблиц целых чисел (уже не в символьной, а в цифровой форме). Для рассматриваемого примера эта таблица будет иметь следующий вид:

```
BYTE Check[]={0xFF,0xFE,0x7C,0x79,0xB3,0xA7,0xCF,0xDF};
```

Можно заметить, что это те же самые числа, которые образуют вторую часть таблицы ресурсов (или файла CHECK.BMP), но только записанные в сжатом виде и в обратном порядке (лучше сказать, в обратном порядке они записываются в таблице файла ресурсов). Смысл чисел, описывающих изображение, легко понять, рассмотрев рис. 9.3, где представлено в увеличенном виде наше изображение вместе с кодами, относящимися к каждой его строке. В этих кодах единичное значение бита соответствует белой точке, а нулевое – черной. Кодирование цветного изображения, естественно, сложнее.

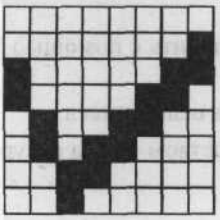


Рис. 9.3. Растровое изображение галочки в увеличенном виде

Вывод растровых изображений

Загрузка растрового изображения

Для того чтобы изображение можно было вывести на экран, оно должно находиться не в файле, а в памяти. Если исходное изображение хранится в файле .BMP, то загрузить его в память можно двумя способами. Первый требует использования файла ресурсов, в котором должно быть предложение, описывающее файл с изображением:

```
pict BITMAP "apple.bmp"
```

На имя этого ресурса ("pict" в данном примере) программа будет ссылаться при загрузке изображения в память. Как известно, при компиляции проекта, в который входит файл ресурсов, фактически выполняются две операции компиляции: исходного текста программы с образованием объектного файла с расширением .OBJ, и файла ресурсов с образованием второго промежуточного файла с расширением .RES. Этот файл содержит изображение практически в том же виде, что и исходный файл .BMP. Далее файлы .OBJ и .RES обрабатываются компоновщиком, который объединяет файлы .OBJ и .RES и создает загрузочный файл .EXE, готовый к выполнению. Ресурсы (не только изображения, но и описание диалогов, меню, текстовых строк и проч.) сохраняются в файле .EXE в специальном формате, понятном редактору ресурсов. Это дает возможность редактировать ресурсы в файле .EXE, не имея исходного текста программы.

Загрузка изображения-ресурса из файла .EXE в память осуществляется в программе вызовом функции LoadImage(). После этого программа может приступить к процедуре вывода изображения в окно. Отметим, что в этом случае для выполнения программы файл .BMP уже не нужен; изображение хранится в выполнимом файле .EXE, который в силу этого может иметь значительный размер.

```
DeleteBitmap(hBitmap);//Освободим совместимую память
SelectBitmap(hdcMem,hOldBitmap);//Восстановим контекст памяти
DeleteDC(hdcMem);//Удалим контекст памяти
PostQuitMessage(0);
}
```

Хранение растровых изображений

Имеется несколько возможностей хранения растровых изображений:

- в файлах с расширением .BMP, создаваемых обычно с помощью того или иного графического редактора, например пакета CorelDraw от Corel Corporation, встроенной в Windows программы Paint (Paintbrush) или одного из универсальных редакторов ресурсов, входящих в пакеты сред программирования Borland C++ или Visual C++; файл можно также получить путем сканирования фотографии с переводом результата в формат .BMP;
- в файле ресурсов в виде таблицы чисел, которую можно получить с помощью того же редактора ресурсов;
- непосредственно в полях данных приложения в виде массива целых чисел.

Если связь файла .BMP с приложением осуществляется посредством файла ресурсов, то в нем должна быть строка, описывающая файл с изображением:

```
Check BITMAP "check.bmp"
```

На имя этого ресурса ("Check") программа будет ссылаться при загрузке изображения в память.

Ссылка на файл .BMP в файле ресурсов может быть заменено непосредственным описанием в файле ресурсов самого изображения в виде таблицы шестнадцатеричных чисел, преобразованных в символьную форму:

```
Check BITMAP {
 '42 4D 5E 00 00 00 00 00 00 00 3E 00 00 00 28 00'
 '00 00 08 00 00 00 08 00 00 00 01 00 01 00 00 00'
 '00 00 20 00 00 00 00 00 00 00 00 00 00 00 00 00'
 '00 00 02 00 00 00 00 00 00 00 FF FF FF 00 DF 00'
 '00 00 CF 00 00 00 A7 00 00 00 B3 00 00 00 79 00'
 '00 00 7C 00 00 00 FE 00 00 00 FF 00 00 00'
 }
```

Здесь представлено описание маленького черно-белого изображения галочки размером всего 8x8 пикселов (рис. 9.2). В начале таблицы расположена служебная информация: ширина и высота изображения (8x8 пикселов), количество цветовых плоскостей (1 для монохромного изображения), число байтов в одной строке изображения (2) и др. За служебной информацией идут данные, описывающие собственно изображение, которое было сначала нарисовано вручную с помощью редактора ресурсов, а затем преобразовано в формат текстовой таблицы.

Рис. 9.2. Пример простого растрового изображения

Между прочим, если сохранить изображение галочки в виде графического файла (например, CHECK.BMP), то его содержимое будет в точности совпадать с приведенной выше таблицей. Программы редакторов ресурсов позволяют выполнять преобразование файла .BMP в символьную таблицу чисел и наоборот, а также редактировать изображе-

```
#include <windowsx.h>
#include "9-1.h"
/*Глобальные переменные*/
HINSTANCE hI;//Дескриптор экземпляра приложения
HBITMAP hBitmap,hOldBitmap;//Дескрипторы изображений в памяти
HDC hdcMem//Дескриптор совместимого контекста
BITMAP bm;//Структура для характеристик растрового изображения
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  char szClassName[]="MainWindow";
  char szTitle[]="Программа 9-1";
  MSG msg;
  WNDCLASS wc;
  hI=hInst;//Сохраним дескриптор приложения
  ZeroMemory (&wc,sizeof(wc));
  wc.lpszClassName=szClassName;
  wc.hInstance=hInst;
  wc.lpfnWndProc=WndProc;
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
  RegisterClass(&wc);
  HWND hwnd=CreateWindow(szClassName,szTitle,WS_OVERLAPPEDWINDOW,
                  0,0,300,150,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_SHOWNORMAL);
  while(GetMessage(&msg,NULL,0,0))
    DispatchMessage(&msg);
  return 0;
  }
/*Оконная процедура главного окна WndProc*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_CREATE,OnCreate);
    HANDLE_MSG(hwnd,WM_PAINT,OnPaint);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
/*Функция обработки сообщения WM_CREATE*/
BOOL OnCreate(HWND,LPCREATESTRUCT){
  hBitmap=(HBITMAP)LoadImage(hI,"Car",IMAGE_BITMAP,0,0,0);//Загрузка
                                                          //изображения
  GetObject(hBitmap,sizeof(BITMAP),&bm);//Получим его характеристики
  HDC hdc=GetDC(hwnd);//Получим контекст устройства
  hdcMem=CreateCompatibleDC(hdc);//Создаем совместимый контекст памяти
  hOldBitmap=SelectBitmap(hdcMem,hBitmap);//Выбор растрового
                        //изображения в совместимый контекст памяти
  ReleaseDC(hwnd,hdc);//Освободим контекст устройства
  return TRUE;
 }
/*Функция обработки сообщения WM_PAINT*/
void OnPaint(HWND hwnd){
  PAINTSTRUCT ps;
  HDC hdc=BeginPaint(hwnd,&ps);//Получим контекст устройства
  BitBlt(hdc,25,20,bm.bmWidth,bm.bmHeight,hdcMem,0,0,SRCCOPY);//Копирование
                        //изображения из памяти на устройство
  EndPaint(hwnd,&ps);
  }
/*Функция обработки сообщения WM_DESTROY*/
void OnDestroy(HWND){
```

Глава 9
Ресурсы: растровые изображения

Важнейшим изобразительным средством графической операционной системы являются цветные изображения разнообразных объектов. Рисунки используются для повышения наглядности курсоров, кнопок, пиктограмм, заставок и других элементов интерфейса; служат иллюстративным материалом для интерактивных учебников и справочников; входят органической частью в состав приложений, предназначенных для управления производственными процессами или механизмами.

Поскольку светящаяся поверхность видеомонитора представляет собой набор точек (точечный растр), любое изображение в конечном счете является точечным. Поэтому приобретают особую важность принципы создания, хранения и вывода на экран точечных, или растровых, изображений. В простейшем случае, для монохромной видеосистемы, каждая точка экрана описывается одним битом посылаемых на экран данных: если бит установлен, точка светится, если сброшен – погашена. Для современных цветных видеосистем это уже не так, но старое название растровых изображений – битовые матрицы или, иногда, битовые карты (от англ. bitmap) осталось в обиходе.

Программа, выводящая растровое изображение

Приведем пример простой программы, в которой некоторое изображение, хранящееся в файле с расширением .BMP, выводится в главное окно. Такое изображение может быть подготовлено с помощью какого-либо графического редактора, в частности программы Paint, входящей в состав системы Windows. На рис. 9.1 в главное окно приложения выведена картинка, используемая в одном из примеров, включенных в пакет IDE Borland C++. В последующих подразделах будет подробно рассмотрены детали организации и функционирования этой программы.

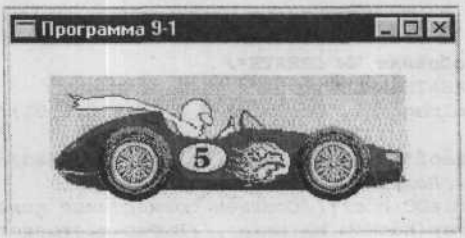


Рис. 9.1. Вывод в главное окно приложения растрового изображения

```
/*Программа 9-1. Вывод растрового изображения*/
/*Файл 9-1.h*/
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
BOOL OnCreate(HWND,CREATESTRUCT FAR*);
void OnPaint(HWND);
void OnDestroy(HWND);
/*Файл 9-1.rc*/
Car BITMAP "race.bmp"
/*Файл 9-1.cpp*/
/*Операторы препроцессора*/
#include <windows.h>
```

Выразив ωt через x, получим из первого уравнения

$\omega t=\arccos(x)$.

Подставив выражение для ωt во второе уравнение, получим связь между x и y:

$y=\cos(k*\arccos(x)+\varphi)$,

причем для получения всех значений y величина $k*\arccos(x)+\varphi$ должна пробегать все значения от 0 до 2π или, что то же самое, от $-\pi$ до $+\pi$. При заданном значении координаты x величина y будет в большинстве случаев принимать два возможных значения (см. рис. 8.7), соответствующие положительному и отрицательному значению функции $\arccos(x)$. Для получения этих значений в программе следует выполнить вычисления величины y отдельно для положительного и отрицательного значений функции arccos(x):

$y1=\cos(k*\arccos(x)+\varphi)$; (1)

$y2=\cos(-k*\arccos(x)+\varphi)$. (2)

Вычисления по формулам (1) и (2) выполняются в функции GraphOnPaint(). Сначала с помощью функции MoveToEx() текущая графическая позиция переносится в первую точку выводимой на экран кривой, а затем двумя циклами проходятся все x-координаты сначала от –100 до +100, а затем в обратном направлении и с помощью функции LineTo() проводятся линии от текущей графической позиции к следующей точке кривой. Для того чтобы кривая имела большой размер и располагалась в середине окна графика (размер которого был задан 220*220 пикселов), ординаты точек умножаются на 100 и смещаются вниз на 110 пикселов.

Коэффициент k, характеризующий отношение частот двух колебаний, может использоваться при вычислениях по формулам (1) и (2) непосредственно, так как значения позиции ползунка составляют целые числа от 1 до 10. Однако значения позиции бегунка на линейке прокрутки необходимо нормировать так, чтобы они располагались в диапазоне 0...π. Поскольку максимальное значение позиции бегунка у нас составляет 180, в формулу для вычисления y1 и y2 подставляются значения Phi/180*M_PI (символическая константа M_PI обозначает число π). Однако здесь нас подстерегает неприятность: при делении *целого числа* Phi на 180 будут отброшены все цифры после десятичной точки и в результате для всех Phi (кроме его максимального значения, равного 180) будет получено значение 0. Для исправления результата переменная Phi перед делением преобразовывается в число с плавающей точкой.

мента управления, пославшего данное сообщение. Если сообщение WM_HSCROLL приходит от линейки с ползунком (дескриптор hwndTrack), то дальнейшие действия выполняются лишь при условии, что код уведомления, поступающий в функцию через ее параметр code, равен символической константе TB_ENDTRACK. Такое значение кода уведомления говорит о том, что пользователь окончил перетаскивание ползунка или отпустил клавишу мыши после щелчка по линейке. В этом случае элементу управления вызовом функции SendMessage() посылается сообщение TBM_GETPOS с требованием вернуть в программу значение новой позиции ползунка. Значение позиции запоминается в глобальной переменной nFreq для дальнейшего использования в функции GraphOn Paint().

Если сообщение WM_HSCROLL приходит от линейки прокрутки (дескриптор hwndScroll), то выполняется обработка пяти различных кодов уведомления от всех перечисленных выше возможных действий пользователя (перемещение бегунка вправо или влево на одну позицию или на целую страницу, а также его перетаскивание мышью). Все блоки обработки уведомляющих кодов идентичны и состоят из трех операций:

- соответствующего действию пользователя изменения текущей позиции бегунка в элементе nPos структуры SCROLLINFO;
- анализа новой позиции на предмет ее выхода за установленный диапазон и возврата ее в границы диапазона; если, например, при крайнем левом положении бегунка пользователь тем не менее щелкнул по левой стрелке, декремент переменной nPos присвоит ей бессмысленное (для нашего примера) значение –1, которое следует изменить на 0;
- сообщения самому элементу управления о новой позиции бегунка путем вызова функции SetScrollInfo() с указанием в качестве третьего параметра адреса переменной sinfo и заданием флагу перерисовки (последний параметр) значения TRUE, что заставит Windows перерисовать бегунок в новой позиции.

Получив новое значение позиции бегунка для линейки прокрутки, мы вызовом функции wsprintf() преобразовываем его в символьную форму и посылаем в статический элемент управления с дескриптором hwndStatic для отображения в главном окне под линейкой прокрутки. Помимо этого числовое значение новой позиции бегунка пересылается в переменную Phi, чтобы потом использовать ее в алгоритме вычисления точек кривой Лиссажу. Никакой необходимости в этой переменной нет, она введена исключительно для наглядности.

Последняя операция в функции OnHScroll() заключается в инициировании с помощью функции InvalidateRect() посылки сообщения WM_PAINT в окно графика (дескриптор hwndGraph) с целью вычисления и вывода в окно новой формы фигуры Лиссажу. Эта последняя операция выполняется при изменении положения любого из двух имеющихся в главном окне элементов управления.

Фигуры Лиссажу, демонстрирующие сложение двух гармонических колебаний с различающимися частотами и фазами, получаются при изменении x- и y-координат точки по следующему закону:

$x = \cos(\omega t)$;
$y = \cos(k\omega t+\varphi)$,

где коэффициент k характеризует отношение частот двух колебаний, а прибавляемая к частоте величина φ – сдвиг фаз между ними. Аргумент ωt может изменяться в пределах $0...2\pi$.

Поясняющие надписи, относящиеся к линейке с ползунком, выполняются в виде статических элементов управления. Как уже отмечалось ранее, это позволяет нам избежать необходимости обрабатывать сообщение WM_PAINT для главного окна (для окна графика, в котором изображение изменяется в зависимости от настройки элементов управления, сообщение WM_PAINT обязательно должно обрабатываться).

При манипуляциях пользователя с ползунком (щелчками мышью справа или слева от него, а также перетаскиванием ползунка мышью в новую позицию) от ползунка будут поступать сообщения WM_HSCROLL, которые мы должны обрабатывать с целью получения значения новой позиции ползунка. Этот вопрос будет рассмотрен ниже.

Далее создается второй элемент управления – линейка прокрутки, принадлежащая к предопределенному классу с именем "SCROLLBAR". Для задания числовых характеристик этого элемента используется глобальная структурная переменная sinfo типа SCROLLINFO, элементы которой инициализируются требуемым образом. Назначение элементов этой структуры указано в комментариях к соответствующим программным предложениям.

Вызовом функции SetScrollInfo() с указанием в качестве первого параметра дескриптора окна линейки прокрутки, а в качестве третьего – адреса структурной переменной sinfo заданные нами числовые характеристики придаются конкретному элементу управления. Второй параметр функции SetScrollInfo(), константа SB_CTL, указывает, что данный элемент является автономным (в отличие от линеек прокрутки, располагаемых по правому или нижнему краям главного или другого окна). Третий параметр – флаг перерисовки – в нашем случае значения не имеет.

Поясняющие надписи для линейки прокрутки, как и для линейки с ползунком, создаются функцией CreateWindow() как статические элементы управления предопределенного класса "STATIC". Для того чтобы в одном из этих элементов, указывающем текущую настройку сдвига фаз, можно было изменять текст, возвращаемый функцией CreateWindow() дескриптор этого элемента сохраняется в глобальной переменной hwndStatic.

При использовании линейки прокрутки необходимо иметь в виду, что перемещение бегунка линейки может осуществляться тремя способами:

- щелчок мыши по правой или левой стрелке на концах линейки должен переместить бегунок на одну позицию, т. е. на одну единицу установленного диапазона (в нашем случае диапазон составляет 180 единиц);
- щелчок мыши по линейке прокрутки справа или слева от бегунка должен переместить бегунок на одну "страницу", величина которой произвольно задается в соответствующем элементе структурной переменной типа SCROLLINFO; мы установили размер страницы, равный 1/10 от всего диапазона, т. е. 18°;
- перетаскивание бегунка мышью по линейке прокрутки должно перемещать его в любое место линейки.

При перемещении бегунка он, во-первых, не фиксируется на новом месте и, во-вторых, значение новой позиции не устанавливается автоматически. Оба этих действия приходится выполнять вручную в ответ на сообщение WM_HSCROLL, поступающее (в нашем случае) в главное окно приложения при любых манипуляциях пользователя с линейкой прокрутки. Как уже отмечалось выше, то же сообщение поступает в главное окно и от другого элемента управления – линейки в ползунком.

В функции OnHScroll() обработки сообщений WM_HSCROLL мы прежде всего анализируем второй параметр этой функции, в который Windows передает дескриптор эле-

```
lpmmi->ptMaxTrackSize.x=XSIZE;//размеры главного
lpmmi->ptMaxTrackSize.y=YSIZE;//окна совпадают
}
```

В главной функции приложения регистрируется класс главного окна со светло-серым фоном. Вслед за этим сразу же регистрируется класс дочернего окна для графика. Как видно из текста программы, регистрацию классов всех окон удобно выполнять в одном месте, друг за другом, так как в этом случае можно воспользоваться одной структурной переменной типа WNDCLASS и сэкономить несколько программных строк. После создания и показа главного окна программа входит в бесконечный цикл обработки сообщений.

В прикладных программах такого рода с изображением в главном окне чертежей, схем или наборов инструментов случайное изменение размеров окна или перемещение его внутренних элементов разрушит композицию. Поэтому размер главного окна фиксируется с помощью обработки сообщения WM_GETMINMAXINFO, как это уже делалось ранее.

В ответ на сообщение WM_CREATE о создании главного окна выполняется значительный объем работы по созданию вложенных окон и их настройке. Окно для графика создается обычной функцией CreateWindow(). Возвращаемое этой функцией значение дескриптора окна сохраняется в глобальной переменной hwndGraph, поскольку оно нам понадобится в дальнейшем. При формировании константы стиля окна графика указывается, что это окно дочернее (WS_CHILD), с рамкой (WS_BORDER) и видимое (WS_VISIBLE). Было бы заманчиво придать окну графика строку с заголовком, однако в этом случае окно получило бы право перемещаться по экрану, а этого мы хотим избежать.

Линейка с ползунком, как и другие *стандартные* (common) элементы управления, к которым мы еще вернемся в гл. 10, создается функцией CreateWindowEx() с указанием в качестве класса предопределенного обозначения TRACKBAR_CLASS (впрочем, можно было воспользоваться и обычной, не расширенной функцией CreateWindow()). Это окно также объявляется дочерним и видимым и, кроме того, имеет стиль TBS_AUTOTICKS, который приводит к появлению под линейкой масштабных меток. Обратите внимание на указание класса создаваемого окна. Обычно класс задается в форме символьной строки в кавычках или как адрес символьной строки, описанной в каком-то другом месте программы (см., например, ранее придуманный нами класс "Graph" или далее предопределенный класс "SCROLLBAR"). В данном случае обозначение TRACKBAR_CLASS используется без кавычек, так как представляет собой не символьную строку, а макрос, который определен в файле COMMCTRL.H приблизительно таким образом:

```
#define TRACKBAR_CLASS "msctls_trackbar32"
```

Собственно именем класса является символьная строка "msctls_trackbar32".

Обратите внимание на то, что, в отличие от описания элементов управления предопределенных классов в составе диалогового окна, где имена классов (STATIC, LISTBOX и др) можно указывать без кавычек, при программном создании таких элементов функцией CreateWindow() их должны обязательно помещаться в кавычки, так как функция CreateWindow() "не знает" имен предопределенных классов.

Создав окно элемента управления, можно приступить к заданию его числовых характеристик. Для этого окну с помощью функции SendMessage() посылается набор сообщений, характерных для данного элемента управления. Сообщением TBM_ SETRANGE устанавливается диапазон изменяемой величины (в нашем случае от 0 до 10). Сообщением TBM_SETPAGESIZE задается величина перемещения ползунка при щелчке мышью справа или слева от него. В нашем случае естественно установить это значение равным единице. Наконец, с помощью сообщения TBM_SETPOS задается исходное положение ползунка (тоже 1).

```
}
/*Функция обработки сообщений WM_PAINT окна графика*/
void GraphOnPaint(HWND hwnd){
  PAINTSTRUCT ps;//Структура для функции BeginPaint()
  HDC hdc=BeginPaint(hwnd,&ps);//Получим контекст устройства
  MoveToEx(hdc,10,cos(nFreq*acos((float)-1)+
        (float)sinfo.nPos/180*M_PI)*100+110,NULL);
  for(int i=-100;i<+100;i++)
        LineTo(hdc,i+110,cos(nFreq*acos((float)i/100)+
        (float)sinfo.nPos/180*M_PI)*100+110);
  for(int i=+100;i>=-100;i--)
        LineTo(hdc,i+110,cos(-nFreq*acos((float)i/100)+
        (float)sinfo.nPos/180*M_PI)*100+110);
  EndPaint(hwnd,&ps);//Освобождение контекста устройства
  }
/*Функция обработки сообщений WM_HSCROLL от ползунка*/
void OnHScroll(HWND,HWND hwndCtl,UINT code,int pos){
  char szT[10];
  if((hwndCtl==hwndTrack)&&(code==TB_ENDTRACK)){//Если перемещение состоялось
    nFreq=SendMessage(hwndTrack,TBM_GETPOS,0,0);//Новая позиция
    }
  if(hwndCtl==hwndScroll){//Сообщение от линейки прокрутки
    switch(code){
      case SB_LINERIGHT://Сдвиг вправо на 1 позицию
        sinfo.nPos++;//Инкремент позиции в sinfo
        if(sinfo.nPos>sinfo.nMax)
          sinfo.nPos=sinfo.nMax;//Не выходить за пределы
        SetScrollInfo(hwndScroll,SB_CTL,&sinfo,TRUE);//Перемещение бегунка
        break;
      case SB_LINELEFT://Сдвиг влево на 1 позицию
        sinfo.nPos--;//Декремент позиции в sinfo
        if(sinfo.nPos<sinfo.nMin)
          sinfo.nPos=sinfo.nMin;//Не выходить за пределы
        SetScrollInfo(hwndScroll,SB_CTL,&sinfo,TRUE);//Перемещение бегунка
        break;
      case SB_PAGERIGHT://Сдвиг вправо на 1 страницу
        sinfo.nPos+=18;//Модификация позиции в sinfo
        if(sinfo.nPos>sinfo.nMax)
          sinfo.nPos=sinfo.nMax;//Не выходить за пределы
        SetScrollInfo(hwndScroll,SB_CTL,&sinfo,TRUE);//Перемещение бегунка
        break;
      case SB_PAGELEFT://Сдвиг влево на 1 позицию
        sinfo.nPos-=18;//Модификация позиции в sinfo
        if(sinfo.nPos<sinfo.nMin)
          sinfo.nPos=sinfo.nMin;//Не выходить за пределы
        SetScrollInfo(hwndScroll,SB_CTL,&sinfo,TRUE);//Перемещение бегунка
        break;
      case SB_THUMBTRACK://Перемещение бегунка мышью
        sinfo.nPos=pos;//Модификация позиции в sinfo
        if(sinfo.nPos<sinfo.nMin)
          sinfo.nPos=sinfo.nMin;//Не выходить за пределы
        SetScrollInfo(hwndScroll,SB_CTL,&sinfo,TRUE);//Перемещение бегунка
        break;
      }
    wsprintf(szT,"%d",sinfo.nPos);//Преобразуем позицию в число
    SetWindowText(hwndStatic,szT);//Выведем в статический элемент
    }
  InvalidateRect(hwndGraph,NULL,TRUE);//Перерисовать график
  }
/*Функция обработки сообщения WM_GETMINMAXINFO главного окна*/
void OnGetMinMaxInfo(HWND,LPMINMAXINFO lpmmi){
  lpmmi->ptMinTrackSize.x=XSIZE;//Минимальный
  lpmmi->ptMinTrackSize.y=YSIZE;//и максимальный
```

```
//Оконная процедура WndProc главного окна
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg)  {
    HANDLE_MSG(hwnd,WM_CREATE,OnCreate);
    HANDLE_MSG(hwnd,WM_HSCROLL,OnHScroll);
    HANDLE_MSG(hwnd,WM_GETMINMAXINFO,OnGetMinMaxInfo);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
//Функция OnCreate обработки сообщений WM_CREATE
BOOL OnCreate(HWND hwnd,LPCREATESTRUCT){
/*Создадим и покажем окно графика*/
  hwndGraph=CreateWindow("Graph",NULL,WS_CHILD|WS_VISIBLE|WS_BORDER,
                         10,10,220,220,hwnd,NULL,hInstance,NULL);
/*Создадим и настроим линейку с ползунком*/
  hwndTrack=CreateWindowEx(0,TRACKBAR_CLASS,NULL,
      WS_CHILD|WS_VISIBLE|TBS_AUTOTICKS,
      250,55,208,30,hwnd,NULL,hI,NULL);
  SendMessage(hwndTrack,TBM_SETRANGE,TRUE,MAKELPARAM(1,10));//Диапазон
  SendMessage(hwndTrack,TBM_SETPAGESIZE,0,1);//Шаг перемещения
  SendMessage(hwndTrack,TBM_SETPOS,TRUE,1);//Текущая позиция
/*Создадим поясняющие надписи для  линейки с ползунком*/
  CreateWindow("STATIC","Отношение частот по осям",
              WS_CHILD|WS_VISIBLE|SS_LEFT,
              260,25,220,22,hwnd,NULL,hInstance,NULL);
  CreateWindow("STATIC","1   2   3   4   5   6   7   8   9  10",
              WS_CHILD|WS_VISIBLE|SS_LEFT,
              260,85,220,22,hwnd,NULL,hInstance,NULL);
/*Создадим и настроим линейку прокрутки*/
  sinfo.cbSize=sizeof(SCROLLINFO);
  sinfo.fMask=SIF_ALL;//Будем работать со всеми элементами структуры
  sinfo.nMin=0;//Минимальная позиция
  sinfo.nMax=180;//Максимальная позиция
  sinfo.nPage=18;//Размер страницы
  sinfo.nPos=0;//Начальная позиция
  hwndScroll=CreateWindow("SCROLLBAR",NULL,WS_CHILD|WS_VISIBLE,
              260,160,200,20,hwnd,NULL,hI,NULL);
  SetScrollInfo(hwndScroll,SB_CTL,&sinfo,TRUE);
/*Создадим поясняющие надписи для линейки прокрутки*/
  CreateWindow("STATIC","Сдвиг фаз в градусах",WS_CHILD|WS_VISIBLE|SS_LEFT,
              280,130,200,22,hwnd,NULL,hInstance,NULL);
  CreateWindow("STATIC","0",WS_CHILD|WS_VISIBLE|SS_LEFT,
              280,180,22,22,hwnd,NULL,hInstance,NULL);
  CreateWindow("STATIC","180",WS_CHILD|WS_VISIBLE|SS_LEFT,
              430,180,40,22,hwnd,NULL,hInstance,NULL);
/*Создадим статический элемент для вывода сдвига фаз*/
  hwndStatic=CreateWindow("STATIC","00",WS_CHILD|WS_VISIBLE|SS_LEFT,
              355,200,50,20,hwnd,NULL,hInstance,NULL);
  return TRUE;
  }
/*Функция обработки сообщения WM_DESTROY*/
void OnDestroy(HWND){
  PostQuitMessage(0);
  }
//*Оконная процедура окна графика*/
LRESULT CALLBACK GraphWndProc(HWND hwnd,UINT msg,WPARAM wParam,
                                                 LPARAM lParam){
  switch(msg)  {
    HANDLE_MSG(hwnd,WM_PAINT,GraphOnPaint);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
```

Рис. 8.7. Работа программы 8-3

```
/*Файл 8-3.cpp*/
/*Операторы препроцессора*/
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include "8-3.h"
#include <math.h>//Ради cos, acos
/*Глобальные переменные, доступные всем функциям*/
int nPhase;//Сдвиг фаз
int nFreq=1;//Отношение частот
HINSTANCE hInstance;//Дескриптор экземпляра приложения
HWND hwndGraph,hwndTrack,hwndScroll,hwndStatic;//Дескрипторы окон
SCROLLINFO sinfo;//Структура поддержки линейки прокрутки
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  hInstance=hInst;
  InitCommonControls();
  char szClassName[]="MainWindow";
  char szTitle[]="Программа 8-3";
  MSG Msg;
  WNDCLASS wc;
/*Зарегистрируем класс главного окна*/
  memset(&wc,0,sizeof(wc));
  wc.lpszClassName=szClassName;
  wc.hInstance=hInst;
  wc.lpfnWndProc=WndProc;
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hbrBackground=GetStockBrush(LTGRAY_BRUSH);
  RegisterClass(&wc);
/*Зарегистрируем класс окна графика*/
  wc.lpszClassName="Graph";
  wc.lpfnWndProc=GraphWndProc;
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
  RegisterClass(&wc);
/*Создадим и покажем главное окно*/
  HWND hwnd=CreateWindow(szClassName,szTitle,
      WS_OVERLAPPEDWINDOW,10,10,0,0,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_SHOWNORMAL);
/*Войдем в цикл обработки сообщений*/
  while(GetMessage(&Msg,NULL,0,0))
    DispatchMessage(&Msg);
  return 0;
  }
```

Организация виртуального измерительного прибора

В предыдущем примере мы продемонстрировали включение в главное окно приложения некоторых элементов управления, конкретно – окна редактирования, кнопки и элемента класса STATIC для вывода текста. Рассмотрим еще один пример, где в качестве элементов управления будут использованы линейка прокрутки и линейка с ползунком.

С линейками прокрутки, располагаемыми вдоль вертикальной или горизонтальной границ окна, мы уже сталкивались. Однако линейки прокрутки (scrollbar) могут служить и самостоятельным элементом диалогового или обычного окна, задавая исходные значения входных величин, воспринимаемых далее программой (рис. 8.5).

Рис. 8.5. Линейка прокрутки

Элемент управления "линейка с ползунком" (trackbar или slider control) (рис. 8.6) схож с линейкой прокрутки. С его помощью можно также устанавливать исходные значения данных, обрабатываемых программой. Особенно эффектно выглядит использование ползунков в программах, управляющих реальной аппаратурой, поскольку по внешнему виду они напоминают аппаратные ползунки, с помощью которых устанавливаются настройки в различных электронных приборах.

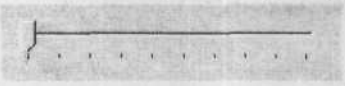


Рис. 8.6. Линейка с ползунком

Рассмотрим программу, в которой в главном окне приложения создается квадратное дочернее окно-панель с белым фоном, в которое выводятся так называемые фигуры Лиссажу, получаемые при одновременном изменении x- и y-координат точки по синусоидальному закону. Если x- и y-координаты изменяются с одинаковой частотой, а сдвиг фаз между ними отсутствует, то фигура Лиссажу вырождается в прямую линию, наклоненную к осям под углом 45°. При сдвиге фаз между колебаниями по осям, равном $\pi/2=90°$, кривая представляет собой правильную окружность. Если же x- и y-частоты не совпадают, да еще между ними имеется сдвиг фаз, то образуются типичные кривые разнообразной формы, знакомые любому школьнику.

Для изменения соотношения частот колебаний по осям x и y мы используем стандартный элемент управления – линейку с ползунком, который в конкретном примере позволяет изменять соотношение частот от 1 до 10, а для задания сдвига фаз между колебаниями – линейку прокрутки, задающую сдвиг фаз от 0 до 2π с шагом $1/180\ \pi$, т. е. через 1°. Устанавливаемые с помощью этих элементов управления значения отношения частот и сдвига фаз отображаются в главном окне под соответствующими элементами.

На рис. 8.7 изображен вид окна программы 8-3 с примером фигуры Лиссажу.

```
/*Программа 8-3. Линейка с ползунком и линейка прокрутки*/
/*Файл 8-3.h*/
#define XSIZE 500
#define YSIZE 265
BOOL OnCreate(HWND,LPCREATESTRUCT);
void OnDestroy(HWND);
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
LRESULT CALLBACK GraphWndProc(HWND,UINT,WPARAM,LPARAM);
void GraphOnPaint(HWND);
void OnHScroll(HWND,HWND,UINT,int);
void OnGetMinMaxInfo(HWND,LPMINMAXINFO);
```

ваемый фрагмент. В статье MessageBeep() интерактивного справочника можно найти перечень допустимых констант с указанием обозначаемых ими стандартных системных действий. Закрепление же за этими действиями конкретных звуков осуществляется с помощью диалогового окна "Звук", которое можно вызвать с помощью команды Пуск➢Настройка➢Панель Управления➢Звук. В этом диалоговом окне можно за любым действием (восстановление окна из значка, открытие программы, закрытие программы и др.) закрепить один из встроенных в Windows файлов звуковых фрагментов, а также выполнить пробное проигрывание этого файла. В различные варианты системы Windows включается более или менее обширный список звуковых файлов для обозначения системных действий: CHIMES.WAV, CHORD.WAV, TADA.WAV и др. В некоторых вариантах Windows их число может достигать нескольких десятков.

В табл. 8.1 приведены обозначения возможных значений параметра функции MessageBeep() с указанием связанных с ними системных действий (из диалогового окна "Звук").

Таблица 8.1. Символические константы функции MessageBeep()

Константа	*Звук*	*Обозначение в диалоговом окне "Звук"*
0xFFFFFFFF	Сигнал динамика компьютера	–
MB_ICONASTERISK	SystemAsterisk	Звездочка
MB_ICONEXCLAMATION	SystemExclamation	Восклицание
MB_ICONHAND	SystemHand	Критическая ошибка
MB_ICONQUESTION	SystemQuestion	Вопрос
MB_OK	SystemDefault	Стандартный звук

Таким образом, если мы хотим в некоторой точке программы вызвать звуковое сопровождение, в это место программы можно включить предложение такого вида:

```
MessageBeep(MB_ICONASTERISK); //Дадим звуковой сигнал
```

При этом надо не забыть, во-первых, с помощью диалогового окна "Звук" закрепить за использованным действием какой-либо звук и, во-вторых, включить звуковую систему.

Больше возможностей дает использование функции sndPlaySound(), в качестве первого параметра которой можно указать имя звукового файла (разумеется, не обязательно из системных ресурсов). Именно такой способ реализован в программе 8-2, где по истечении заданного временно́го интервала проигрывается звуковой файл TADA.WAV:

```
if(nTime==0){//Если заданное время истекло
  sndPlaySound("tada.wav",SND_ASYNC);//Дадим звуковой сигнал
  KillTimer(hwnd,1);//Уничтожим таймер
  }
```

Второй параметр функции sndPlaySound() задает способ проигрывания звукового фрагмента. Константа SND_ASYNC определяет асинхронный режим выполнения операции, когда после запуска проигрывания файла управление сразу же передается в программу, где последующие указанные в ней действия выполняются параллельно с проигрыванием. Константа SND_SYNC задает синхронный режим работы – функция sndPlaySound() возвращает управление в программу лишь после завершения проигрывания звукового фрагмента. Сочетание констант SND_ASYNC | SND_LOOP позволяет организовать циклическое проигрывание заданного фрагмента до повторного выполнения функции sndPlaySound() с указанием в качестве первого параметра значения NULL.

вания во вспомогательную строку szLine, а затем функцией C++ atoi() преобразуется в число, сохраняемое в переменной nTime. Далее с помощью функции wsprintf() содержимое переменной nTime преобразуется в десятичное число в символьной форме с указанием жесткого формата и сохраняется в переменной szResidue, откуда оно уже будет выводиться в окно остатка времени:

```
wsprintf(szResidue,"%03d",nTime);
```

Здесь в спецификации формата символ 0 задает включение в строку лидирующих нулей, символ 3 определяет минимальную длину результирующей строки, а символ d говорит о том, что в строку будет занесено десятичное представление преобразуемого числа.

Нажатие кнопки "Пуск" должно запустить таймер для отсчета времени экспозиции. Как уже говорилось выше, сообщения от таймера можно обрабатывать двумя способами: через цикл обработки сообщений и оконную процедуру с передачей управления на прикладную функцию обработки сообщений WM_TIMER, а также путем непосредственного вызова прикладной функции, адрес которой был при установке таймера передан Windows. В настоящем примере демонстрируется второй способ.

В вызове функции SetTimer()

```
SetTimer(hwndResidue,1,1000,ResidueTimerProc);
```

первый параметр говорит о том, что таймер устанавливается для окна с дескриптором hwndResidue. Вопрос о том, для какого окна устанавливать таймер, особого значения не имеет – таймер будет исправно отсчитывать время, для какого бы окна его ни установить, однако в функцию ResidueTimerProc(), которую Windows вызывает по сигналам таймера, передается дескриптор этого окна, что облегчает организацию программных действий, выполняемых в ответ на сигналы таймера.

Второй параметр функции SetTimer() задает номер таймера, третий – период срабатывания в миллисекундах, а последний параметр, если он не NULL, определяет функцию обратного вызова, на которую Windows будет передавать управление по сигналам таймера.

Если рассматриваемая программа используется для управления реальной установкой, то, установив программный таймер, следует послать в установку сигнал начала измерений.

Последним предложением функции OnCommand() вызывается функция Windows InvalidateRect() для окна остатка времени, что приводит к перерисовыванию этого окна и отображению в нем заданного времени из переменной szResidue.

Функция ResidueTimerProc() обработки сообщений от таймера выполняет декремент оставшегося времени в переменной nTime, преобразование этой величины в символьную форму в переменной szResidue и инициирование перерисовывания окна остатка времени с помощью функции InvalidateRect(). Если заданное время истекло, установленный ранее таймер уничтожается. Именно здесь следовало бы в реальном случае послать в измерительную установку сигнал окончания измерений.

Проигрывание звуковых фрагментов

Если наша программа используется для управления каким-либо реальным объектом, то окончание заданного временного интервала полезно как-то обозначить: вывести в окно приложения яркий значок или подать звуковой сигнал. Вывод изображений будет рассмотрен в следующей главе; здесь же мы покажем, как можно сопровождать выполнение приложения проигрыванием музыкальных фрагментов.

Простейшей функцией, позволяющей воспроизводить встроенные в систему звуковые сигналы (системные звуки), является функция MessageBeep(). В качестве ее единственного параметра указывается символическая константа, характеризующая проигры-

Рис. 8.4. Окно класса "STATIC" с характеристиками по умолчанию

Обрабатывая сообщение WM_PAINT, мы могли бы произвольно изменить любые характеристики выводимого текста: цвет самих символов, цвет фона под символами, режим прозрачности фона и др. Для того чтобы подобные операции можно было выполнить над текстом элемента управления, в Windows предусмотрен ряд специальных сообщений, которые посылаются элементами управления (в диалоговое окно или, в нашем случае, в главное окно приложения) в тот момент, когда Windows приступает к рисованию данного элемента на экране (см. гл.6). Для статического элемента управления следует обрабатывать сообщение WM_CTLCOLORSTATIC. Для этого в оконную процедуру главного окна включен макрос

```
HANDLE_MSG(hwnd,WM_CTLCOLORSTATIC,OnCtlColorStatic);
```

а в состав приложения – функция OnCtlColorStatic(), в которой мы вызовом функции SetBkMode() назначаем символам прозрачный фон.

Однако дело этим не ограничивается. Функция обработки сообщения WM_CTLCOLORSTATIC возвращает в Windows (она ведь вызывается из Windows в качестве функции обратного вызова и ее возвращаемое значение тоже идет в Windows) дескриптор кисти элемента управления. Если не предусмотреть возврат правильного дескриптора, по умолчанию будет возвращен дескриптор белой кисти и текст станет еще более безобразным, чем это получилось на рис. 8.4. Поскольку мы фону главного окна назначили серый цвет (константа GRAY_BRUSH), тот же цвет следует придать и окну статического элемента. Для этого мы берем со склада серую кисть и используем ее дескриптор в качестве возвращаемого значения функции OnCtlColorStaic():

```
return GetStockBrush(GRAY_BRUSH);//Кисть фона элемента управления
```

Ход выполнения программы

Взаимодействие оператора с программой включает два действия: занесение в окно редактирования требуемого значения времени экспозиции и нажатие кнопки "Пуск" для запуска измерений. Для ввода числа в окно редактирования достаточно щелкнуть по нему мышью (чтобы передать ему фокус ввода, т. е. начать направлять в это окно коды нажимаемых клавиш клавиатуры), после чего можно вводить требуемую строку символов. Указав в стиле окна редактирования константу ES_NUMBER, мы исключили возможность ввода символов, отличающихся от цифр. Отображение введенной строки Windows берет на себя, однако для получения ее из окна редактирования элементу управления следует послать сообщение WM_GETTEXT или воспользоваться для этого функцией GetWindowText().

Действия по "запуску измерений" выполняются при поступлении в оконную процедуру главного окна сообщения WM_COMMAND при условии, что оно пришло от кнопки с идентификатором ID_GO. Здесь вызовом функции GetWindowText() с указанием дескриптора окна редактирования hwndEdit введенная строка копируется из окна редактиро-

ных классов и то и другое обеспечивается Windows. В приложении (в его главном окне) достаточно создать требуемые окна; запросы от этих окон будут поступать в главное окно приложения в форме сообщений WM_COMMAND. Однако окно остатка времени мы создаем, так сказать, своими руками (оно не принадлежит к окнам стандартных, предопределенных классов). Для таких дочерних окон, как уже говорилось выше, необходимо:

- зарегистрировать класс дочернего окна;
- создать дочернее окно;
- организовать оконную процедуру дочернего окна.

Все действия, связанные с созданием дочерних окон, довольно логично выполняются в функции OnCreate() обработки сообщения WM_CREATE главного окна приложения. При регистрации окна класса "ResidueWindow" ему назначается светло-серый цвет фона, отличный от цвета фона главного окна (которому придан серый цвет) и оконная процедура ResidueWindowProc(). Эта процедура выглядит обычным образом: набор макросов, определяющий состав обрабатываемых сообщений и вызов для всех остальных сообщений функции DefWindowProc(), осуществляющей их обработку по умолчанию. В нашем примере оконная процедура окна остатка времени обрабатывает единственное сообщение WM_PAINT.

Создание окна остатка времени выполняется вызовом функции CreateWindow() с указанием констант стиля WS_CHILD (дочернее), WS_VISIBLE (видимое) и WS_DLGFRAME (с толстой рамкой). Заметьте, что при определении стиля окна ему не придано ни каких-либо кнопок изменения размеров, ни системного меню, ни заголовка. В этом случае окно нельзя будет перетаскивать по экрану и изменять его размер, что в данном случае и требуется.

Кстати, размер главного окна приложения мы также зафиксировали с помощью соответствующей обработки сообщения WM_GETMINMAXINFO в точности так же, как это было сделано в предыдущем примере).

Далее в функции OnCreate() создаются три окна стандартных классов. Для окна редактирования класса EDIT в глобальной переменной hwndEdit сохраняется возвращаемый системой дескриптор; для двух других окон дескрипторы нам не потребуются. Однако окну-кнопке мы назначаем идентификатор ID_GO, чтобы потом можно было выделять сообщения, поступающие от этого элемента управления.

На заключительном этапе выполнения функции OnCreate() создается шрифт большого размера для вывода в окно остатка времени оставшегося числа секунд.

Задание изобразительных характеристик элементов управления

Для вывода в главное окно строки текста "Введите экспозицию:" мы создали статический элемент управления, "заголовок" которого соответствует выводимому тексту. Можно было поступить по-другому: вывести текст функцией TextOut() в функции обработки сообщения WM_PAINT главного окна. Использование статического элемента управления заметно сократило текст программы при тех же результатах. Однако здесь нас подстерегает одна неприятность. Окно класса STATIC имеет предопределенный цвет (конкретно – светло-серый) и предопределенный режим вывода текста (черные символы по тому же светло-серому фону). Поскольку для фона главного окна мы выбрали серый цвет, статическое окно будет неприятно выделяться (рис. 8.4).

```
  lpmmi->ptMinTrackSize.x=XSIZE;//Минимальный
  lpmmi->ptMinTrackSize.y=YSIZE;//и максимальный
  lpmmi->ptMaxTrackSize.x=XSIZE;//размеры главного
  lpmmi->ptMaxTrackSize.y=YSIZE;//окна совпадают
  }
/*Функция обработки сообщения WM_DESTROY главного окна*/
void OnDestroy(HWND){
  DeleteFont(hFont);//Удалим созданный шрифт
  PostQuitMessage(0);
  }
/*Оконная процедура дочернего окна остатка времени*/
LRESULT CALLBACK ResidueWindowProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM
lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_PAINT,ResidueWindowOnPaint);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
/*Функция обработки сообщений от  таймера окна остатка времени*/
void CALLBACK ResidueTimerProc(HWND hwnd,UINT,UINT,DWORD){
  nTime--;//Декремент числа секунд
  wsprintf(szResidue,"%03d",nTime);//Преобразуем в символы в жестком формате
  InvalidateRect(hwnd,NULL,TRUE);//Инициируем перерисовывание окна остатка
  if(nTime==0){//Если заданное время истекло
    sndPlaySound("tada.wav",SND_ASYNC);//Дадим звуковой сигнал
    KillTimer(hwnd,1);//Уничтожим таймер
    }
  }
/*Функция обработки сообщения WM_PAINT окна остатка времени*/
void ResidueWindowOnPaint(HWND hwnd){
  PAINTSTRUCT ps;
  HDC hdc=BeginPaint(hwnd,&ps);
  SetBkMode(hdc,TRANSPARENT);
  TextOut(hdc,3,2,szResidueTitle,strlen(szResidueTitle));
  HFONT hOldFont=SelectFont(hdc,hFont);
  TextOut(hdc,10,12,szResidue,2);
  SelectFont(hdc,hOldFont);
  EndPaint(hwnd,&ps);
  }
/*Функция обработки сообщения о рисовании статического элемента*/
HBRUSH OnCtlColorStatic(HWND,HDC hdc,HWND,INT){
  SetBkMode(hdc,TRANSPARENT);//Делаем фон символов прозрачным
  return GetStockBrush(GRAY_BRUSH);//Кисть фона элемента управления
  }
```

Структура программы

В главном окне приложения создаются 4 вложенных, конкретно – дочерних (стиль WS_CHILD), окна. Большое почти квадратное окно с произвольным именем класса "ResidueWindow" и дескриптором hwndResidue предназначено для вывода времени, оставшегося до конца измерений. Остальные 3 окна имеют стандартные имена классов EDIT, BUTTON и STATIC и принадлежат, таким образом, к элементам управления. Окно класса EDIT служит для ввода значения времени измерения, окно класса BUTTON реализует кнопку пуска, а окно класса STATIC предназначено для вывода строки текста "Введите экспозицию:".

Каждое дочернее окно, создаваемое в рамках главного окна приложения, должно иметь свой зарегистрированный в Windows класс и свою оконную процедуру для обработки сообщений, поступающих в это окно или исходящих из него. Для окон стандарт-

```
    TranslateMessage(&msg);
    DispatchMessage(&msg);
    }
  return 0;
  }
/*Оконная процедура главного окна*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_CREATE,OnCreate);
    HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);
    HANDLE_MSG(hwnd,WM_GETMINMAXINFO,OnGetMinMaxInfo);
    HANDLE_MSG(hwnd,WM_CTLCOLORSTATIC,OnCtlColorStatic);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
/*Функция обработки сообщения WM_CREATE о создании главного окна*/
BOOL OnCreate(HWND hwnd,LPCREATESTRUCT){
  WNDCLASS wc;
  memset(&wc,0,sizeof(wc));
  wc.lpszClassName=szResidueClass;//Имя дочернего окна
  wc.lpfnWndProc=ResidueWindowProc;//Оконная функция дочернего окна
  wc.hInstance=hI;//Дескриптор экземпляра приложения
  wc.hbrBackground=GetStockBrush(LTGRAY_BRUSH);
  RegisterClass(&wc);//Регистрируем дочернее окно (остатка времени)
/*Создадим дочернее окно*/
  hwndResidue=CreateWindow(szResidueClass,NULL,
        WS_CHILD|WS_DLGFRAME|WS_VISIBLE, 220,10,130,100,hwnd,NULL,hI,NULL);
/*Создадим окно редактирования*/
  hwndEdit=CreateWindow("EDIT",NULL,WS_CHILD|WS_VISIBLE|WS_BORDER|ES_NUMBER,
                        162,10,45,20,hwnd,NULL,hI,NULL);
/*Создадим кнопку пуска*/
  CreateWindow("BUTTON","Пуск",WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,
               162,40,45,35,hwnd,(HMENU)ID_GO,hI,NULL);
/*Создадим статический элемент управления для текста*/
  CreateWindow("STATIC",szExpo,WS_CHILD|WS_VISIBLE|SS_LEFT,10,10,150,20,
               hwnd,NULL,hI,NULL);
/*Создадим большой шрифт*/
  char szFace[]="Times New Roman Cyr";
  LOGFONT lf;
  ZeroMemory(&lf,sizeof(lf));
  strcpy(lf.lfFaceName,szFace);
  lf.lfHeight=80;
  hFont=CreateFontIndirect(&lf);
  return TRUE;
  }
/*Функция обработки сообщения WM_COMMAND главного окна*/
void OnCommand(HWND,int id, HWND,UINT){
  char szLine[10];
  if(id==ID_GO){//Если сообщение от кнопки пуска
    if(GetWindowText(hwndEdit,szLine,9)){//Получим введенное, если оно есть,
      nTime=atoi(szLine);//преобразуем в число
      wsprintf(szResidue,"%03d",nTime);//Преобразуем в символы
      SetTimer(hwndResidue,1,1000,ResidueTimerProc);//Установим таймер
      InvalidateRect(hwndResidue,NULL,TRUE);//Перерисуем окно остатка
      }//Конец if(GetWindowText)
    }//Конец if(id==ID_GO)
  }//Конец OnCommand()
/*Функция обработки сообщения WM_GETMINMAXINFO главного окна*/
void OnGetMinMaxInfo(HWND,LPMINMAXINFO lpmmi){
```

Рис. 8.3. Вывод программы, управляющей измерениями

```
/*Программа 8-2. Дочерние окна и таймеры Windows*/
/*Файл 8-2.h*/
#define ID_GO 101
#define XSIZE 375
#define YSIZE 150
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
BOOL OnCreate(HWND,LPCREATESTRUCT);
void OnCommand(HWND,int,HWND,UINT);
void OnGetMinMaxInfo(HWND,LPMINMAXINFO);
void OnDestroy(HWND);
LRESULT CALLBACK ResidueWindowProc(HWND,UINT,WPARAM,LPARAM);
void ResidueWindowOnPaint(HWND hwnd);
void CALLBACK ResidueTimerProc(HWND hwnd,UINT,UINT,DWORD);
HBRUSH OnCtlColorStatic(HWND,HDC,HWND,INT);
//Файл 8-2.cpp
/*Операторы препроцессора*/
#include <windows.h>
#include <windowsx.h>
#include "8-2.h"
/*Глобальные переменные*/
HINSTANCE hI;//Дескриптор экземпляра приложения
char szResidueClass[]="ResidueWindow";//Имя класса окна остатка времени
char szExpo[]="Введите экспозицию:";
char szResidue[]="000";
char szResidueTitle[]="Осталось секунд";
HWND hwndResidue;//Дескриптор окна остатка времени
HWND hwndEdit;//Дескриптор окна редактирования
int nTime;//Время экспозиции
HFONT hFont;//Дескриптор большого шрифта
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  char szClassName[]="MainWindow";//Имя класса главного окна
  char szTitle[]="Программа 8-2";//Заголовок главного окна
  MSG msg;
  WNDCLASS wc;
  hI=hInst;//Сохраним дескриптор экземпляра приложения
  ZeroMemory(&wc,sizeof(wc));
  wc.lpszClassName=szClassName;
  wc.hInstance=hInst;
  wc.lpfnWndProc=WndProc;
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hbrBackground=GetStockBrush(GRAY_BRUSH);
  RegisterClass(&wc);//Регистрируем главное окно
  HWND hwnd=CreateWindow(szClassName,szTitle,WS_OVERLAPPEDWINDOW,
    10,10,XSIZE,YSIZE,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_SHOWNORMAL);
  while(GetMessage(&msg,NULL,0,0)){//Цикл обработки сообщений
```

За высоту пика принимается разность nBuf[nMax] – nBuf[nMin], а за полувысоту nHalfHight – половина этого значения. Для нахождения полуширины пика выполняется еще один пустой цикл по точкам с номерами, большими nMax, т. е. справа от максимума, пока ордината точки не станет меньше высоты срединной линии пика. Номер этой точки и принимается за правую границу пика на полувысоте. Полуширина пика nHalfWidth определяется далее как разность правой границы пика на полувысоте и положения его максимума.

Стоит обратить внимание на особенность использования цикла с оператором while. Шаги цикла повторяются, пока поставленное условие выполняется; однако внутри цикла мы выполняем постинкремент переменной цикла. В результате после выхода из цикла переменная цикла будет всегда иметь значение на единицу больше, чем при последнем анализе условия. Поэтому, выйдя из цикла, мы уменьшаем найденное значение j на единицу.

Окна предопределенных классов в главном окне приложения

До сих пор мы сталкивались с такими конструкциями приложений Windows:

- приложение с главным окном;
- приложение без главного окна на основе модального диалога;
- приложение с главным окном и вложенным диалоговым окном (модальным или немодальным);
- приложение с главным и вложенным дочерним окном.

В действительности в рамках главного окна приложения можно иметь не только вложенные дочерние или всплывающие окна, созданные в программе, но и любые окна встроенных в Windows классов – кнопки, списки, статические элементы с текстом и др. Таким образом, рассмотренные нами ранее органы управления не обязательно должны быть элементами диалога; с таким же успехом их можно использовать в составе главного окна приложения или любых вложенных окон. В этом и следующем разделах мы рассмотрим некоторые вопросы организации и обслуживания вложенных окон предопределенных классов.

Программа для управления измерениями в реальном времени

Пусть нам надо разработать программу, предназначенную для управления процессом измерения числа некоторых случайных событий за данный интервал времени (число автомобилей, прошедших через пункт контроля или число гамма-квантов, зарегистрированных детектором излучения). Программа должна по команде оператора включить аппаратуру, осуществляющую измерения, а через заданный интервал времени выключить ее. В такой программе должно быть окно для ввода требуемого интервала времени и кнопка пуска измерений. Помимо этого целесообразно предусмотреть средства оповещения оператора о времени, оставшемся до конца текущего сеанса. Разумеется, в программе должны быть еще и средства вывода на экран числа сосчитанных событий, но мы этот вопрос опустим, так как в нашем распоряжении нет ни автомобилей, ни гамма-квантов, и подсчитывать реальные события мы не будем.

Для измерения интервала времени естественно воспользоваться таймером Windows, а для наблюдения остатка времени можно использовать дочернее окно. Компоновка главного окна приложения показана рис. 8.3.

прочитанных из файла данных nBuf[nMax] преобразуются в символьную форму (в виде десятичных значений) и выводятся в дочернее окно обычной функцией TextOut().

Сообщения WM_PAINT для дочернего окна (так же как и для главного) инициируются в программе в функции OnCommand() всякий раз после чтения нового файла и завершения процедуры нахождения характеристик кривой, построенной по прочитанным из файла данным. Для этого используются вызовы функции InvalidateRect(), в качестве первого параметра которой указывается дескриптор того окна, в которое надо послать сообщение. В функции OnCommand() обработки сообщения WM_COMMAND главного окна известен дескриптор hwnd главного окна; чтобы вызвать функцию InvalidateRect() для дочернего окна, нам пришлось при создании дочернего окна сохранить его дескриптор в глобальной переменной hDataWin.

Определение характеристик функциональной кривой

Для придания правдоподобия рассматриваемому примеру в нем определяются типичные характеристики выводимой на экран функциональной кривой: положение ее максимума, значение кривой в максимуме, а также половинное значение ширины пика. В действительности определение этих величин представляет собой весьма сложную и даже не вполне определенную задачу, учитывая, что реальные кривые могут иметь несколько, в том числе частично перекрывающихся, пиков и при реальных измерениях имеют не такую идеально гладкую форму, как у нас, а обладают случайным разбросом. Свою неопределенность вносит и форма спадающей кривой, на которую накладываются пики. В настоящем примере без всякой претензии на точность использованы весьма примитивные алгоритмы.

Прежде всего надо найти точку, соответствующую началу пика. Учитывая, что до этой точки кривая может только или спадать, или находиться на одном уровне, поиск начала пика осуществляется с помощью пустого цикла while, который выполняется до тех пор, пока ордината текущей j-й точки кривой больше или равна ординате следующей j+1-й точки:

```
while(nBuf[j+1]<=nBuf[j++]);
```

Как только ордината следующей точки оказывается выше текущей, это место считается началом пика и номер текущей точки сохраняется в переменной nMin. (рис. 8.2).

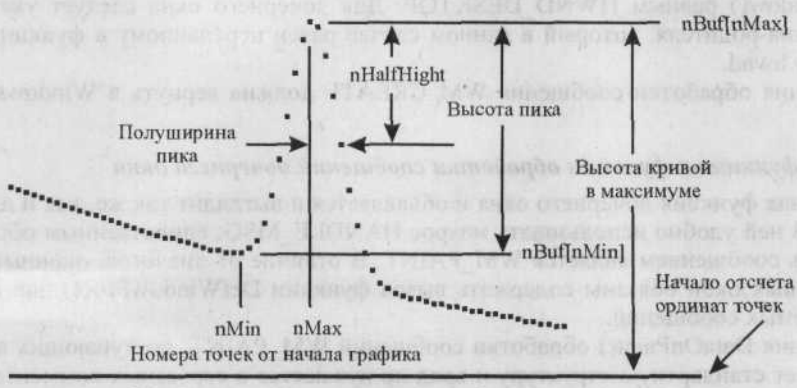


Рис. 8.2. Определение характеристик функциональной кривой

От точки с номером nMin кривая идет вверх до своего максимума. Максимум находится аналогичным способом: прокручивается пустой цикл while до тех пор, пока ордината следующей точки не станет меньше текущей. Номер текущей точки сохраняется в переменной nMax как положение максимума.

Элемент wc.hbrBackground содержит дескриптор кисти, которым закрашивается фон окна. Для разнообразия и выразительности мы используем для дочернего окна серую кисть.

В элементе wc.lpszMenuName у нас хранился адрес имени меню (главного окна) в файле ресурсов. При создании дочернего (со стилем WS_CHILD) окна это данное игнорируется, но для аккуратности мы его обнулили. Всплывающее (со стилем WS_POPUP) окно можно создать с собственным меню, при этом сценарий меню описывается в файле ресурсов, а в элемент wc.lpszMenuName помещается адрес имени соответствующего меню (которых в приложении может быть несколько).

В элемент wc.lpszClassName заносится адрес произвольного имени класса дочернего окна. Все создаваемые в программе классы должны иметь различающиеся имена; мы присвоили классу окна имя "DataWindow".

Заполнив все необходимые поля структурной переменной wc, можно повторно вызвать функцию RegisterClass(), которая выполнит регистрацию класса теперь уже дочернего окна.

После регистрации класса дочернего окна его надо создать. Эту операцию можно выполнить в любом удобном месте программы и, в частности, в ответ на выбор некоторого пункта меню; если, однако, мы хотим, чтобы дочернее окно появилось одновременно с главным, естественно создать дочернее окно в функции обработки сообщения WM_CREATE. Соответственно в оконную функцию главного окна добавилось предложение

```
HANDLE_MSG(hwnd,WM_CREATE,OnCreate);
```

а к функциям обработки сообщений главного окна прибавилась функция OnCreate(). Эта функция состоит из единственного предложения вызова функции Windows Create Window() для окна класса DataWindow. Стиль окна

```
WS_CHILD|WS_VISIBLE|WS_CAPTION
```

определяет дочернее окно с тонкой рамкой и полосой заголовка, которое создается в видимом состоянии. Кстати, обратите внимание на то, что для создаваемых окон не обязательно вызывать функцию показа окна ShowWindow(); достаточно придать окну стиль WS_VISIBLE. Это относится не только к дочерним окнам, но и к главному окну, хотя по отношению к главному окну такой способ используют редко.

Создавая главное окно, мы устанавливали восьмой по порядку параметр функции CreateWindow() равным HWND_DESKTOP. Для дочернего окна следует указать дескриптор окна-родителя, который в данном случае равен переданному в функцию Create() аргументу hwnd.

Функция обработки сообщения WM_CREATE должна вернуть в Windows значение TRUE.

Оконная функция и функции обработки сообщений дочернего окна

Оконная функция дочернего окна и объявляется и выглядит так же, как и для главного окна. В ней удобно использовать макрос HANDLE_MSG; единственным обрабатываемым здесь сообщением является WM_PAINT. В отличие от диалогов, оконные функции порожденных окон обязаны содержать вызов функции DefWindowProc() для всех необрабатываемых сообщений.

Функция DataOnPaint() обработки сообщений WM_PAINT, поступающих в дочернее окно, имеет стандартную структуру и вряд ли нуждается в серьезных комментариях. После получения контекста устройства в нем вызовом функции SetBkMode() устанавливается прозрачный фон символов, чтобы выводимые символы рисовались не на белом (по умолчанию) фоне, а на фоне окна. С помощью вызовов функции wsprintf() подготовленные заранее данные в переменных nMax и nHalfWidth, а также максимальное значение

```
        }//Конец if
    }//Конец switch(id)
  }
//Оконная функция дочернего окна
LRESULT CALLBACK DataWndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_PAINT,DataOnPaint);
    default:
      return DefWindowProc(hwnd,msg,wParam,lParam);//Обязательно для окон
    }
  }
//Функция обработки сообщения WM_PAINT для дочернего окна
void DataOnPaint(HWND hwnd){
  PAINTSTRUCT ps;
  HDC hdc=BeginPaint(hwnd,&ps);
  SetBkMode(hdc,TRANSPARENT);//Установим прозрачный фон символов
  char s[40];
  wsprintf(s,"Центр пика = %d",nMax);//Образуем строку вывода
  TextOut(hdc,10,0,s,strlen(s));//Вывод строки в дочернее окно
  wsprintf(s,"Высота пика = %d",nBuf[nMax]);//Образуем строку вывода
  TextOut(hdc,10,18,s,strlen(s));//Вывод строки в дочернее окно
  wsprintf(s,"Полуширина пика = %d",nHalfWidth);//Образуем строку вывода
  TextOut(hdc,10,36,s,strlen(s));//Вывод строки в дочернее окно
  EndPaint(hwnd,&ps);
  }
```

По сравнению с программой 6-7 в рассматриваемый вариант добавлены следующие блоки:

- несколько глобальных переменных – для имени класса дочернего окна и его дескриптора, а также набор переменных для определения характеристик кривой;
- функция обработки сообщения WM_CREATE для главного окна; в этой функции создается дочернее окно;
- оконная функция дочернего окна, в которой предусмотрена обработка лишь одного сообщения WM_PAINT;
- функция обработки сообщения WM_PAINT для дочернего окна;
- блок вычисления характеристик кривой (положения максимума и пр.), помещенный в то место функции OnCommand(), где открывается выбранный файл и, следовательно, можно вычислить и вывести в дочернее окно его характеристики.

Процедура создания дочернего окна. Сообщение WM_CREATE

Процедура создания и обслуживания дочернего, как и любого другого, окна состоит из трех этапов:

- регистрации класса дочернего окна, в процессе которой за окном закрепляется его оконная функция;
- создания окна функцией Windows CreateWindow() с указанием стиля окна и его местоположения;
- обработки сообщений, поступающих в дочернее окно.

Классы всех порожденных окон удобно регистрировать в начале функции WinMain, вслед за регистрацией класса главного окна. Для этого естественно воспользоваться той же структурной переменной wc типа WNDCLASS, которая использовалась для регистрации класса главного окна. Часть полей данных в ней уже заполнена, но некоторые надо заполнить заново.

В элемент wc.lpfnWndProc заносится адрес оконной функции, которая будет обрабатывать сообщения, поступающие в дочернее окно. Мы дали ей имя DataWndProc().

```
        SetPixel(hdc,i,r.bottom-5-nBufScaled[i],RGB(0,0,255));//Вывод точек
        break;
        }//Конец case DOTS
      case CURVE:{
        MoveToEx(hdc,0,r.bottom-5-nBufScaled[0],NULL);
        for(int i=1;i<500;i++)
          LineTo(hdc,i,r.bottom-5-nBufScaled[i]); //Вывод отрезков
        }//Конец case CURVE
      }//Конец switch(Mode)
    }//Конец if
  SelectPen(hdc,hOldPen);
  EndPaint(hwnd,&ps);
  }
/*Функция обработки сообщения WM_DESTROY*/
void OnDestroy(HWND){
  DeleteObject(hPen);
  PostQuitMessage(0);
  }
/*Оконная процедура немодального диалога*/
BOOL CALLBACK DlgProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_INITDIALOG,DlgOnInitDialog);
    HANDLE_MSG(hwnd,WM_COMMAND,DlgOnCommand);
    default:
      return FALSE;
    }
  }
/*Функция обработки сообщения WM_INITDIALOG для диалогового окна*/
BOOL DlgOnInitDialog(HWND hwnd,HWND,LPARAM){
  char S[10];//Строка для обмена данными со списком LISTBOX
  SendDlgItemMessage(hwnd,Mode,BM_SETCHECK,1,0L);//"Нажать" кнопку режима
  for (int i=0;i<8;i++){//Заполнение списка LISTBOX строками текста
    wsprintf(S,"%d%%",nScales[i]);//Пересылка в строку числа и знака %
    SendDlgItemMessage(hwnd,ID_SCALE,LB_ADDSTRING,0,(LPARAM)S);//Пересылка строки
    }//Конец цикла for
  SendDlgItemMessage(hwnd,ID_SCALE,LB_SETCURSEL,nScaleIndex,0L);
  return TRUE;
  }
/*Функция обработки сообщений WM_COMMAND для диалогового окна*/
void DlgOnCommand(HWND hwnd,int id,HWND,UINT codeNotify){
  switch(id){//Код органа управления
    case IDOK://Сообщение от кнопки "Закрыть"
    case IDCANCEL://Сообщения от системного меню диалога
      DestroyWindow(hwnd);//Уничтожение окна немодального диалога
      hSettingsBox=NULL;//Сбросим в нуль его дескриптор
      break;
    case ID_DOTS://Сообщение от кнопки "Точки"
      Mode=DOTS;//Установим режим "Точки"
      InvalidateRect(GetParent(hwnd),NULL,TRUE);//Инициируем WM_PAINT
      break;
    case ID_CURVE://Сообщение от кнопки "Огибающая"
      Mode=CURVE;//Установим режим "Огибающая"
      InvalidateRect(GetParent(hwnd),NULL,TRUE);//Инициируем WM_PAINT
      break;
    case ID_SCALE://Сообщение от списка; проанализируем извещение
      if(codeNotify==LBN_SELCHANGE){//Если изменилось выделение в списке
        nScaleIndex=(int)SendDlgItemMessage(hwnd,ID_SCALE,LB_GETCURSEL,0,0L);
        nCurrentScales=(float)nScales[nScaleIndex]/100;//Новый масштаб
        for(int i=0;i<500;i++)//Масштабирование спектра в соответствии
          nBufScaled[i]=nBuf[i]*nCurrentScales;//с новым масштабом
        InvalidateRect(GetParent(hwnd),NULL,TRUE);//Инициируем WM_PAINT
```

```
  char szFilter[]="Файлы данных(*.DAT)\0*.dat\0Все файлы(*.*)\0*.*\0";
  DWORD nCnt;//Вспомогательная переменная для файловых операций
  switch(id){
    case MI_OPEN:
      memset(&ofn,0,sizeof(OPENFILENAME));
      ofn.lStructSize=sizeof(OPENFILENAME);//Размер структуры
      ofn.hwndOwner=hwnd;//Дескриптор главного окна-владельца диалога
      ofn.lpstrFilter=szFilter;//Строка с шаблонами имен искомых файлов
      ofn.lpstrFile=szFile;//Буфер для имени файла
      ofn.nMaxFile=sizeof(szFile);//Его размер
      ofn.Flags=OFN_PATHMUSTEXIST|OFN_FILEMUSTEXIST;
      if(GetOpenFileName(&ofn)){
        HANDLE hFile=
              CreateFile(szFile,GENERIC_READ,0,0,OPEN_EXISTING,0,NULL);
        if(hFile==INVALID_HANDLE_VALUE)
          break;//Если не удалось открыть файл
        ReadFile(hFile,nBuf,2*500,&nCnt,NULL);//Чтение из файла
        CloseHandle(hFile);
        bDataOK=TRUE;//Данные имеются, можно выводить
/*Выполним начальное масштабирование спектра*/
        for(int i=0;i<500;i++)
          nBufScaled[i]=nBuf[i]*nCurrentScales;
/*Приступим к определению характеристик выводимой на экран кривой*/
/*Найдем начало пика (nMin) и центр пика (nMax)*/
        int j=0;
        while(nBuf[j+1]<=nBuf[j++]);//До начала возрастания
        nMin=j;//Координата точки в начале возрастания
        while(nBuf[j+1]>=nBuf[j++]);//До максимума
        nMax=j-1;//Номер точки с максимальным значением
/*Найдем ширину на полувысоте*/
        int nHalfHight=(nBuf[nMax]-nBuf[nMin])/2;//Половина высоты пика
        j=nMax;//Номер точки с максимальным значением
        while((nBuf[j++]>=
              (nBuf[nMax]-nHalfHight))&(j<500));//До половины максимума
        nHalfWidth=j-1-nMax;//Число точек на полувысоте
/*Инициируем WM_PAINT для главного и дочернего окон*/
        InvalidateRect(hwnd,NULL,TRUE);//Инициируем WM_PAINT в главное окно
        InvalidateRect(hDataWin,NULL,TRUE);//WM_PAINT в дочернее окно
        break;
        }
      else break;//Если файл не выбран
    case MI_EXIT:
      DestroyWindow(hwnd);
      break;
    case MI_SETTINGS:
      if(hSettingsBox==NULL)//Если диалог еще не открыт, создадим его
        hSettingsBox=CreateDialog(hInstance,"SettingsDlg",hwnd,DlgProc);
        break;
      }
  }
//Функция OnPaint обработки сообщения WM_PAINT
void OnPaint(HWND hwnd){
  RECT r;//Структура, описывающая координаты прямоугольника
  PAINTSTRUCT ps;
  HDC hdc=BeginPaint(hwnd,&ps);//Получение контекста устройства
  HPEN hOldPen=SelectPen(hdc,hPen);//Выберем синее перо в контекст
  GetClientRect(hwnd,&r);//Получение координат рабочей области
  if(bDataOK){
    switch(Mode){
      case DOTS:{
        for(int i=0;i<500;i++)
```

```
HWND hSettingsBox;//Дескриптор немодального диалога
GraphModes Mode=DOTS;//Режим наблюдения и его начальное значение
HPEN hPen;//Дескриптор синего пера
char szDataClassName[]="DataWindow";//Имя класса дочернего окна
HWND hDataWin;
int nMin,nMax,nHalfWidth;//Переменные для характеристик графика
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  MSG msg;
  hInstance=hInst;
  WNDCLASS wc;
/*Регистрируем класс главного окна*/
  ZeroMemory (&wc,sizeof(wc));
  wc.style=CS_VREDRAW;
  wc.lpfnWndProc=WndProc;//Оконная процедура главного окна
  wc.hInstance=hInst;
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
  wc.lpszMenuName="Main";//Меню главного окна
  wc.lpszClassName=szClassName;//Имя класса главного окна
  RegisterClass(&wc);//Регистрация главного окна
/*Регистрируем класс дочернего окна*/
  wc.lpfnWndProc=DataWndProc;//Оконная функция дочернего окна
  wc.hbrBackground=GetStockBrush(LTGRAY_BRUSH);//Цвет окна светло-серый
  wc.lpszMenuName=NULL;//В дочернем окне меню не бывает
  wc.lpszClassName=szDataClassName;//Имя класса дочернего окна
  RegisterClass(&wc);//Регистрация дочернего окна
/*Создаем и показываем главное окно*/
  HWND hwnd=CreateWindow(szClassName,szTitle,WS_OVERLAPPEDWINDOW,
                    0,0,600,400,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_SHOWNORMAL);
/*Цикл обработки сообщений*/
  while(GetMessage(&msg,NULL,0,0)){
    TranslateMessage(&msg);
    DispatchMessage(&msg);
    }
  return 0;//Завершение WinMain() и возврат в Windows
  }
/*Оконная процедура главного окна*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_CREATE,OnCreate);
    HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);
    HANDLE_MSG(hwnd,WM_PAINT,OnPaint);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
/*Функция обработки сообщения WM_CREATE главного окна*/
BOOL OnCreate(HWND hwnd,LPCREATESTRUCT){
  hPen=CreatePen(PS_SOLID,1,RGB(0,0,255));//Создаем синее перо для графика
  hDataWin=CreateWindow(szDataClassName,"Числовые характеристики",
           WS_CHILD|WS_VISIBLE|WS_CAPTION|WS_BORDER,0,0,250,85,
           hwnd,NULL,hInstance,NULL);//Создали дочернее окно
  return TRUE;
  }
/*Функция обработки сообщения WM_COMMAND от меню*/
void OnCommand(HWND hwnd,int id,HWND,UINT){
  OPENFILENAME ofn; //Структура для функции GetOpenFileName
```

```
/*Программа 8-1. Дочернее окно*/
/*Файл 8-1.h*/
/*Константы*/
#define MI_OPEN 101
#define MI_EXIT 102
#define MI_SETTINGS 103
#define ID_DOTS 201
#define ID_CURVE 202
#define ID_SCALE 203
/*Определение нового типа GraphModes*/
enum GraphModes{DOTS=ID_DOTS,CURVE=ID_CURVE};
/*Прототипы функций для главного окна*/
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
BOOL OnCreate(HWND,LPCREATESTRUCT);
void OnCommand(HWND,int,HWND,UINT);
void OnPaint(HWND);
void OnDestroy(HWND);
/*Прототипы функций для диалогового окна*/
BOOL CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM);
BOOL DlgOnInitDialog(HWND,HWND,LPARAM);
void DlgOnCommand(HWND,int,HWND,UINT);
/*Прототипы функций для дочернего окна*/
LRESULT CALLBACK DataWndProc(HWND,UINT,WPARAM,LPARAM);
void DataOnPaint(HWND);
/*Файл 8-1.rc*/
#include "8-1.h"
Main MENU{
  POPUP "&Файл"{
    MENUITEM "&Открыть...",MI_OPEN
    MENUITEM "&Режимы...",MI_SETTINGS
    MENUITEM SEPARATOR
    MENUITEM "В&ыход",MI_EXIT
    }
  }
SettingsDlg DIALOG 150,0,125,80
STYLE WS_VISIBLE | WS_SYSMENU | WS_POPUP
CAPTION "Режимы вывода данных"{
  CONTROL "Вид", -1, "BUTTON", BS_GROUPBOX, 10, 10, 65, 45
  CONTROL "Точки", ID_DOTS, "BUTTON", BS_AUTORADIOBUTTON, 15, 20, 45, 14
  CONTROL "Огибающая", ID_CURVE,"BUTTON", BS_AUTORADIOBUTTON, 15, 35, 55, 14
  CONTROL "", ID_SCALE,"LISTBOX",LBS_NOTIFY|WS_VSCROLL|WS_BORDER,85,15,30,50
  CONTROL "Масштаб", -1, "STATIC", SS_LEFT, 85, 3, 40, 10
  CONTROL "Закрыть", IDOK, "BUTTON", BS_PUSHBUTTON, 25, 60, 40, 14
  }
/*Файл 8-1.cpp*/
/*Операторы препроцессора*/
#include <windows.h>
#include <windowsx.h>
#include "8-1.h"
/*Глобальные переменные*/
char szClassName[]="MainWindow";//Имя класса главного окна
char szTitle[]="Программа 8-1";//Заголовок главного окна
char szFile[128]; //Для спецификации выбранного файла
short nBuf[500]; //Буфер для чтения массива из файла
BOOL bDataOK=FALSE;//Если TRUE, данные прочитаны в буфер nBuf
HINSTANCE hInstance;//Дескриптор экземпляра приложения
short nBufScaled[500]; //Буфер для масштабированного массива
int nScales[8]={200,175,150,125,100,75,50,25};//Массив масштабов
int nScaleIndex=4;//Текущее (и начальное) значение индекса в массиве масштабов
float nCurrentScales=1;//Текущее (и начальное) значение масштаба
```

ния окном, вписывает текстовые строки в соответствующие поля, выбирает требуемые режимы и т. д. *Дочерние* окна часто используются как информационные, а также для изменения каких-то характеристик отдельных областей главного окна. Выделив, например, ключевые слова справки в отдельные окна, можно с этим классом окон связать курсор характерной формы (обычную стрелку заменить на изображение ладони), задать в контексте устройства другой цвет символов и вдобавок определить функцию обработки сообщений WM_LBUTTONDOWN (или WM_LBUTTONDBLCLK), чтобы при щелчке мышью на выделенном слове вызывался поясняющий это слово текст.

Диалоговые окна относятся к окнам предопределенных классов, описанных в Windows. Это ограничивает их возможности, но упрощает обслуживание органов управления диалогового окна, так как в Windows предусмотрены стандартные средства их взаимодействия.

В тех случаях, когда порожденное окно должно обеспечивать нетипичное поведение, целесообразно создать в программе не диалоговое, а обычное порожденное окно и наделить его требуемыми характеристиками. При этом если форма диалоговых окон описывается в файле ресурсов, а создание осуществляется функциями DialogBox() или CreateDialog(), то порожденные (дочерние или всплывающие) окна описываются непосредственно в программе, а создаются универсальными функциями CreateWindow() или CreateWindowEx().

Пример использования дочернего окна для вывода результатов вычислений

Описываемая ниже программа 8-1 является развитием примера 6-7, в который добавлены средства создания и обслуживания дочернего окна. Это окно размещается в левом верхнем углу главного и предназначено для представления характеристик выводимой на экран кривой – центра пика, его высоты, а также ширины на половине высоты (так называемой полуширины) (рис. 8.1). При этом предполагается, что отображаемая функциональная зависимость имеет вид относительного узкого пика, наложенного на медленно спадающую кривую. Такая форма функциональной зависимости характерна для результатов измерений некоторых физических величин, например энергий элементарных частиц, регистрируемых специальными приборами – детекторами ядерных излучений.

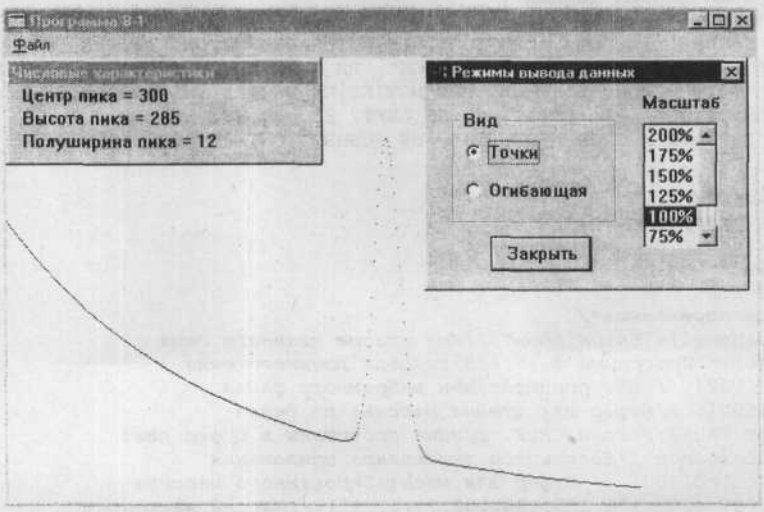


Рис. 8.1. Вид главного окна программы 8-1 с дочерним окном и диалогом

Глава 8
Окна Windows

Организация дочерних окон

Любое реальное приложение Windows содержит большое количество окон. Хотя многие графические элементы (текстовые строки, геометрические фигуры, растровые изображения) можно вывести прямо в главное окно, такой способ формирования экранного кадра обычно оказывается неудобным, главным образом из-за невозможности взаимодействия (с помощью мыши или клавиатуры) с этими элементами. Организация же системы вложенных, или порожденных, окон позволяет для каждого такого окна или группы окон иметь, например, свою форму курсора или свой цвет фона; самое же главное достоинство порожденных окон заключается в возможности придания каждому окну индивидуальной оконной функции. Это позволяет отдельно и по-разному обрабатывать для каждого окна поступающие в него сообщения Windows (и кстати, посылать в окна программно сформированные сообщения).

Реально почти все изображения, которые мы видим на экране, – разнообразные кнопки, полосы прокрутки, ползунки, поля для ввода текста, изображения различных приборов и индикаторов – представляют собой окна с теми или иными характеристиками. Обычно эти окна образуют иерархическую систему. Например, любой диалог прежде всего представляет собой окно; однако в нем обычно имеются еще и внутренние окна – кнопки, списки, полосы прокрутки и т. д.

Диалоговые окна, использованные нами в программах предыдущей главы, относились к числу всплывающих окон, что определялось указанием при описании их стиля константы WS_POPUP. Модальные диалоговые окна могут быть только всплывающими. Вообще же порожденные окна, в частности немодальные диалоги, могут быть как всплывающими, так и дочерними; в последнем случае в описании их стиля присутствует константа WS_CHILD. Как уже упоминалось в гл. 6 (применительно к немодальным диалоговым окнам), всплывающие и дочерние окна различаются целым рядом характеристик, из которых важнейшими являются следующие:

- если в главном окне приложения одновременно порождены и всплывающие, и дочерние окна, то всплывающие окна будут изображаться поверх дочерних, будут "всплывать" над ними, что и определило их название;
- дочерние окна могут перемещаться только в пределах родительского окна, в то время как положение всплывающих окон не ограничено какими-либо границами; всплывающее окно можно вытащить за пределы главного окна приложения и поместить в любом месте экрана, для чего необходимо, чтобы у всплывающего окна была не только рамка, но и полоса заголовка (см. рис. 6.27 и 6.28);
- координаты дочерних окон задаются относительно границ рабочей области родительского окна; координаты всплывающих окон задаются относительно границ экрана;
- всплывающее окно может содержать собственное меню; в дочерних окнах меню не бывает.

Указанные различия делают *всплывающие* окна более удобными для организации активного диалога с пользователем, в процессе которого он работает с органами управле-

На выполнение прикладной функции OnUser(), которой через ее аргументы wParam и lParam передаются (при необходимости) наши параметры parm1 и parm2, уже не накладывается никаких ограничений; в ней можно выполнять любые действия и, в частности, вызывать любые функции Windows. Следует только иметь в виду, что функция OnUser() не вызывается непосредственно прерыванием от таймера; сообщение WM_USER поступает в очередь сообщений приложения, а функция OnUser() будет вызвана, лишь когда дойдет очередь до обработки этого сообщения. Описанный здесь механизм иногда называют отсроченной или отложенной обработкой.

После того, как необходимость в периодических действиях отпала, мультимедийный таймер необходимо уничтожить:

```
timeEndPeriod(1);//Указывается заданное ранее разрешение
timeKillEvent(mmr);//Указывается номер данного таймера
```

Задание однократного интервала времени

При необходимости отработки однократного интервала времени необходимо выполнить все описанные выше действия, только в качестве режима установки временного события указывается константа TIME_ONESHOT:

```
timeBeginPeriod(1);//Установим максимальное разрешение
mmr=timeSetEvent(1000, 1, TimeProc, 0, TIME_ONESHOT);
...//Продолжение программы

void CALLBACK TimeProc(UINT, UINT, DWORD, DWORD, DWORD){
  PostMessage(hwnd, WM_USER, (WPARAM)parm1, (LPARAM)parm2);
  time EndPeriod(1);//Отмена установленного ранее разрешения
..timeKillEvent(mmr);//В предположении, что mmr - глобальная переменная
```

В данном варианте сообщение WM_USER посылается лишь один раз по истечении точно 1с (с погрешностью в 1 мс). В прикладной функции TimeProc обработки этого сообщения, помимо активизации содержательных действий (посредством посылки сообщения WM_USER), необходимо отменить установленное ранее временное разрешение, как это и показано в приведенных выше строках.

значении времени разрешения или при установке нескольких таймеров системе может не хватить ресурсов, что приведет к ее аварийной остановке.

Следующим этапом является установка временного события, которая выполняется с помощью функции timeSetEvent(). В качестве параметров этой функции указывается, в частности, временной интервал срабатывания таймера, а также адрес той прикладной функции обратного вызова, которая будет активизироваться при каждом его срабатывании.

Уничтожение мультимедийного таймера требует вызова двух функций: timeEndPeriod(), отменяющей установленное ранее разрешение таймера, и timeKillEvent(), которая прекращает действие всех системных процессов, связанных с работой мультимедийного таймера.

Фрагмент программы, в которой устанавливается мультимедийный таймер, может выглядеть таким образом:

```
timeBeginPeriod(1);//Установим максимальное разрешение
MMRESULT mmr=timeSetEvent(10,1,TimeProc,0,TIME_PERIODIC);
...//Продолжение программы
```

В качестве параметров функции timeSetEvent() указывается период его срабатывания (10 мс в примере), значение установленного ранее разрешения (1 мс), имя прикладной функции обработки прерываний от таймера (например, TimeProc), произвольное данное пользователя, которое будет передано в эту функцию (у нас 0), а также символическая константа, задающая режим работы таймера. Функция установки таймера возвращает (в переменную типа MMRESULT) его номер, назначаемый системой и используемый затем нами при уничтожении данного таймера.

Прикладная функция TimeProc(), вызываемая в данном примере каждые 10 мс, предназначена для выполнения требуемых периодических действий. Сложность, однако, заключается в том, что в этой функции запрещен вызов каких-либо функций Windows, кроме мультимедийных, а также функции PostQuitMessage(). В результате типичный текст функции TimeProc() выглядит следующим образом:

```
void CALLBACK TimeProc(UINT, UINT, DWORD, DWORD, DWORD){
  PostMessage(hwnd, WM_USER, (WPARAM)parm1, (LPARAM)parm2);
  }
```

Вызов функции PostMessage() приводит к посылке в окно hwnd нашего приложения сообщения пользователя с кодом WM_USER, в состав которого входят два произвольных параметра parm1 и parm2. Для Windows код WM_USER (он равен 0x400), при использовании его в рамках окон прикладных классов, ничего не значит, так как стандартные сообщения Windows имеют коды от 0 до WM_USER–1, однако мы можем обрабатывать сообщение WM_USER в нашей оконной функции наравне с остальными (системными) сообщениями:

```
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,
                         WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_CREATE,OnCreate);
    HANDLE_MSG(hwnd,WM_PAINT,OnPaint);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
... //Обработка других сообщений Windows
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    case WM_USER:
      OnUser(wParam,lParam);//Вызов прикладной функции обработки сообщений
                            //от мультимедийного таймера
    }
  }
```

грамме 7-2) адрес строки параметров запуска. Если некоторое приложение запускается командой Пуск➢Выполнить с указанием параметра, например,

```
APP.EXE D:\\MYFYLE.001
```

то параметр lpstr будет указывать на строку со следующим содержимым:

```
"D:\\MYFYLE.001",0
```

В нашем случае параметр должен представлять собой число, однако в строке lpstr он будет храниться в символьной форме. Для преобразования символьной строки в число удобно использовать функцию C++ atoi(). Если указывать в качестве параметра число секунд, то для функции SetTimer() это число надо умножить на 1000, чтобы перевести в количество миллисекунд.

Мультимедийные таймеры

Основным недостатком обычных таймеров Windows является невысокая предельная частота (18,2 Гц) и, как следствие этого, низкое временное разрешение (55 мс). Если воспользоваться таким таймером для плавного перемещения изображения какого-либо объекта по экрану, то при разрешении экрана 800×600 пикселов объект затратит на прохождение всего экрана по горизонтали более 40 с, т. е. движение его будет чрезвычайно медленным. Для получения более высоких скоростей перемещения и для отсчета интервалов времени с более высокой точностью можно использовать мультимедийные таймеры, предельное разрешение которых составляет 1 мс, что соответствует частоте 1 кГц.

Для работы с мультимедийными таймерами предусмотрена небольшая группа специальных функций, имена которых начинаются со слова time (начинающегося со строчной буквы, что нетипично для имен функций Windows).

Рассмотрим несколько вариантов установки и использования мультимедийного таймера.

Измерение интервалов времени

В процессе оптимизации программ может возникнуть необходимость измерить время выполнения того или иного фрагмента программы. Для этого можно использовать мультимедийную функцию timeGetTime(), которая возвращает время в миллисекундах, истекшее от последней загрузки операционной системы:

```
DWORD t1,t2,t3;//Переменные для записи времени
t1=timeGetTime();
...//Контролируемый фрагмент программы
t2=timeGetTime();
t3=t2-t1;//Время выполнения контролируемого фрагмента в мс
```

Далее не составит труда с помощью функции wsprintf() преобразовать полученное время в символьную строку и вывести либо в окно сообщения, либо непосредственно в главное или какое-то другое окно приложения.

Организация периодического процесса

Установка и использование мультимедийного таймера требует выполнения целого ряда действий. Прежде всего с помощью функции timeBeginPeriod() задается требуемое временное разрешение устанавливаемого таймера в миллисекундах. Хотя минимальное значение параметра этой функции составляет 1 мс, однако следует иметь в виду, что установленный таймер активно использует ресурсы операционной системы и при малом

Однако в программе присутствуют:

- цикл обнаружения сообщений;
- функция обратного вызова обработки сообщений от таймера;
- стандартное завершение приложения вызовом функции PostQuitMessage().

Как уже отмечалось в начале этой главы, обрабатывать сообщения от таймера можно двумя способами: стандартным образом через оконную функцию с передачей управления на прикладную программу обработки сообщений WM_TIMER и в обход оконной функции путем непосредственного вызова соответствующей CALLBACK-процедуры. В настоящем примере естественным образом используется второй способ, так как у нас нет ни оконной функции, ни самого окна.

Однако отсутствие оконной функции не избавляет нас от необходимости организации цикла обнаружения сообщений. Дело в том, что таймер по-прежнему посылает в наше приложение сообщения WM_TIMER. И по-прежнему передачу управления на прикладную функцию обработки этих сообщений (у нас она названа TimerProc()) осуществляет функция DispatchMessage(), только не посредством оконной функции, а непосредственно. Поэтому приложения, использующие таймеры Windows, обязаны иметь цикл обработки сообщений.

Этот цикл нужен еще и для правильного завершения приложения. В конце функции TimerProc(), в которой осуществляется отмена установленного ранее таймера, вызывается функция PostQuitMessage(), которая, как это подробно описывалось в гл. 4, разрывает цикл обработки сообщений, что и приводит к завершению всего приложения.

Рассмотрим некоторые принципиальные детали приведенного выше примера. Функция установки таймера SetTimer() требует в качестве первого параметра дескриптор окна, с которым этот таймер связывается. У нас никаких окон нет, и на месте этого параметра приходится указывать NULL. Однако тогда возникает другая проблема – как сообщить функции отмены таймера KillTimer(), о каком именно таймере идет речь. На месте первого параметра функции KillTimer() вместо дескриптора конкретного окна мы также указываем NULL, а идентификатор конкретного таймера поступает в функцию TimerProc() из Windows через ее третий параметр, idEvent. Его-то мы и передаем функции KillTimer() в качестве идентификатора таймера.

В случае установки в программе нескольких таймеров (например, с различными периодами работы) для них разумно установить отдельные CALLBACK-функции, каждая из которых будет обслуживать сообщения от таймера по своему алгоритму, а при необходимости отменить таймер сможет это сделать, получив через тот же параметр idEvent идентификатор своего таймера. В принципе можно ограничиться и одной CALLBACK-функцией, если при установке таймеров запомнить в глобальных переменных их идентификаторы

```
UINT id1,id2;
...
id1=SetTimer(NULL,0,Time1,TimerProc);
id2=SetTimer(NULL,0,Time2,TimerProc);
```

а в единственной функции TimerProc() обработки сообщений от них анализировать параметр idEvent, сравнивая его значение с переменными id1 и id2.

Рассмотрим теперь вопрос о передаче в приложение параметра, указываемого пользователем в строке запуска приложения. При вызове главной функции приложения WinMain() Windows передает через ее третий параметр (локальная переменная lpstr в про-

ние программы состоит в измерении заданного ей интервала времени и сообщении пользователю об окончании этого интервала выводом на экран предупреждающего сообщения. Предусмотрим для нашей программы несколько нестандартную, хотя и чрезвычайно удобную для всяких экспериментов конструкцию. Пусть в ней вообще не будет никаких окон, кроме сообщения об окончании интервала; интервал же мы будем задавать при запуске программы (с помощью команды Пуск➢Выполнить) через параметр командной строки. Тогда запуск программы в форме

```
7-2.exe 5
```

приведет к настройке программы на интервал, равный 5 с. Конечно, запускать приложения Windows через команду "Выполнить", да еще с параметром, довольно неудобно, однако если решать эту задачу более изящно, нам пришлось бы для задания параметра организовывать в программе диалоговое (или простое) окно с окном редактирования внутри. В нашем же варианте программа оказывается существенно более простой.

```
/*Программа 7-2. Однократный таймер*/
/*Файл 7-2.cpp*/
#include <windows.h>
VOID CALLBACK TimerProc(HWND,UINT,UINT,DWORD);
int nTime;//Переменная для параметра командной строки
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR lpstr,int){
/*Прочитаем параметр командной строки и установим таймер*/
  int nTime=atoi(lpstr);//Преобразуем параметр в число
  SetTimer(NULL,1,nTime*1000,TimerProc);//Установим таймер на nTime секунд
  MSG Msg;
  while(GetMessage(&Msg,NULL,0,0))//Организуем цикл
    DispatchMessage(&Msg);//обнаружения сообщений (необходим!)
  return 0;//Завершение приложения
  }
/*Функция обратного вызова обработки сообщений от таймера*/
VOID CALLBACK TimerProc(HWND,UINT,UINT idEvent,DWORD){
  char szText[100];
  KillTimer(NULL,idEvent);//Выключим таймер
  wsprintf(szText, "Установленное время\n%d с истекло!",nTime);
  MessageBox(NULL,szText,"Предупреждение",MB_ICONHAND);
  PostQuitMessage(0);//Завершим приложение стандартным образом
  }
```

Результат работы программы приведен на рис. 7.2.

Рис. 7.2. Сообщение из программы 7-2

Вся программа не содержит и двадцати предложений языка. Чтобы подчеркнуть нестандартность программы, перечислим *отсутствующие* в ней элементы. В программе нет:

- действий по регистрации, созданию и показу главного окна и, следовательно, самого окна;
- оконной процедуры;
- функций обработки каких-либо сообщений.

данный компьютер. Обе эти функции заполняют структуру типа SYSTEMTIME, определенную в файле WINBASE.H и имеющую следующий состав:

```
typedef struct _SYSTEMTIME {
  WORD wYear; //Год
  WORD wMonth; //Месяц
  WORD wDayOfWeek; //День недели
  WORD wDay; //День
  WORD wHour; //Час
  WORD wMinute; //Минута
  WORD wSecond; //Секунда
  WORD wMilliseconds; //Миллисекунда
  } SYSTEMTIME;
```

Вся информация возвращается в эту структуру в числовом виде. Например, в элементе структуры wDayOfWeek число 0 обозначает воскресенье, 1 – понедельник и т. д. Для преобразования этих чисел в привычные нам обозначения в программе предусмотрены массивы строк szDay[7] и szMonth[13] с сокращенными названиями дней недели и месяцев. Строка szCurrentTime, выводимая в окно приложения, формируется многократным вызовом функции C++ strcat() сцепления символьных строк, причем в качестве операнда-источника указывается соответствующий по номеру элемент массива строк. Между выводимыми элементами в строку szCurrentTime вставляются пробелы.

Обратите внимание на то, что массив названий месяцев содержит не 12, а 13 элементов, причем первому элементу назначен адрес пустой символьной строки. Так пришлось сделать из-за того, что в элементе wMonth структуры SYSTEMTIME январь обозначается числом 1, февраль – числом 2 и т. д. до декабря, которому назначен номер 12. Однако индексы любого массива начинаются от 0, которому в данном случае не соответствует никакой месяц, поэтому в массив szMonth[13] пришлось ввести этот "лишний" элемент, присвоив ему адрес пустой строки.

Для формирования числовой информации (даты и времени) извлекаемые из структурной переменной SystemTime числа преобразуются в символьную форму функцией wsprintf() и также сцепляются с предыдущей частью итоговой строки.

После полного формирования выводимой в окно строки вызывается функция InvalidateRect(), которая объявляет окно приложения поврежденным, что приводит к посылке в приложение сообщения WM_PAINT. Функция обработки этого сообщения OnPaint() выглядит обычным образом: в ней в функциональных скобках BeginPaint()-EndPaint() устанавливается режим прозрачного фона для шрифта и функцией TextOut() строка szCurrentTime выводится в окно. Таким образом, в нашем случае функция OnPaint() активизируется по сигналам от таймера один раз в секунду.

В функции обработки сообщения WM_DESTROY функцией KillTimer() уничтожается созданный нами таймер, хотя в данном случае это излишняя предосторожность, поскольку при завершении приложения Windows освобождает все занятые приложением ресурсы.

Измерение однократных интервалов

В предыдущем примере мы использовали таймер для *периодического* вывода на экран некоторой информации, конкретно – текущего времени. Изображение в окне обновлялось каждую секунду; нетрудно было для уменьшения отвлекающего от работы мелькания установить период таймера, скажем, 5 с или 1 мин. Рассмотрим теперь другую задачу – использование таймера для измерения однократного интервала времени. Пусть единственное назначе-

будет посылать в главное окно приложения сообщение WM_TIMER. В функции OnTimer() обработки этого сообщения вызывается подпрограмма OutDateTime(), которая запрашивает у системы текущие значения даты и времени и организует их обновление в главном окне приложения.

Фиксация размеров главного окна

В стиле окна, задаваемым четвертым параметром функции CreateWindowEx(), указаны лишь константы, определяющие наличие заголовка и системного меню (чтобы можно было закрыть приложение обычным путем). Таким образом, в строке заголовка окна отсутствуют кнопки изменения его размеров. Однако в обычном случае размер окна можно было бы изменить вручную, перетаскиванием его границ мышью. Чтобы устранить эту возможность, в программе использован стандартный прием – обработка сообщения WM_GETMINMAXINFO. Это сообщение посылается в приложение при создании окна, когда Windows должна определить его начальные размеры и положение на экране, а также при каждой попытке пользователя изменить его размеры.

В функцию обработки сообщения WM_GETMINMAXINFO из Windows поступает указатель (произвольно названный в программе lpmmi) на структуру типа MINMAXINFO, определенную в файле WINUSER.H следующим образом:

```
typedef struct tagMINMAXINFO {
  POINT ptReserved;//Зарезервировано
  POINT ptMaxSize;//Максимальный размер окна в развернутом состоянии
  POINT ptMaxPosition;//Координаты начала окна в развернутом состоянии
  POINT ptMinTrackSize;//Минимально допустимый размер окна
  POINT ptMaxTrackSize;//Максимально допустимый размер окна
  } MINMAXINFO;
```

Структуры POINT, используемые в качестве членов структуры MINMAXINFO, определены как пары точек, задающие координаты или размеры:

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

Таким образом, предложения

```
lpmmi->ptMinTrackSize.x=XSIZE;
lpmmi->ptMinTrackSize.y=YSIZE;
```

задают минимально возможные размеры окна по горизонтали и вертикали. Максимальные размеры задаются аналогично. Если максимальные размеры не совпадают с минимальными, окно в процессе выполнения приложения можно будет изменять только в заданных пределах. В рассматриваемой программе минимальные и максимальные размеры окна совпадают, что исключает возможность их изменения.

Поскольку размеры окна задаются в функции обработки сообщения WM_GETMINMAXINFO, значения размеров, указываемых в качестве параметров функции Create Window(), уже не имеют значения и на их месте можно, например, указать нули.

Определение и вывод текущего времени

Для получения системного времени в Windows имеется несколько функций. Функция GetSystemTyme() возвращает время по Гринвичу, которое носит название универсального скоординированного времени (Universal Coordinated Time, UTC). Мы воспользовались другой функцией, GetLocalTime(), возвращающей местное время, на которое настроен

```
  PostQuitMessage(0);
  }
/*Функция обработки сообщений WM_TIMER*/
void OnTimer(HWND,UINT){
  OutDateTime();//Периодический вывод текущего времени
  }
/*Функция получения текущего времени и преобразования его в символы*/
void OutDateTime(){
  char* szDay[7]={"Вск","Пнд","Втр","Ср","Чтв","Птн","Суб");
  char* szMonth[13]={"","Янв","Февр","Март","Апр","Май","Июнь",
                     "Июль","Авг","Сент","Окт","Нояб","Дек");
  char szT[20];
  SYSTEMTIME SystemTime;//Структура для получения системного времени
  GetLocalTime(&SystemTime);//Получим текущее местное время
  strcpy(szCurrentTime,szDay[SystemTime.wDayOfWeek]);//День недели
  strcat(szCurrentTime," ");//Разделяющий пробел
  strcat(szCurrentTime,szMonth[SystemTime.wMonth]);//Месяц
  strcat(szCurrentTime," ");//Разделяющий пробел
  wsprintf(szT,"%d",SystemTime.wDay);//Дата в символы
  strcat(szCurrentTime,szT);//Добавим ее
  strcat(szCurrentTime," ");//Разделяющий пробел
  wsprintf(szT,"%d",SystemTime.wYear);//Год в символы
  strcat(szCurrentTime,szT);//Добавим его
  strcat(szCurrentTime,"---");//Разделяющие символы
  wsprintf(szT,"%d",SystemTime.wHour);//Часы в символы
  strcat(szCurrentTime,szT);//Добавим их
  strcat(szCurrentTime,":");//Разделяющее двоеточие
  wsprintf(szT,"%d",SystemTime.wMinute);//Минуты в символы
  strcat(szCurrentTime,szT);//Добавим их
  strcat(szCurrentTime,":");//Разделяющее двоеточие
  wsprintf(szT,"%d",SystemTime.wSecond);//Секунды в символы
  strcat(szCurrentTime,szT);//Добавим их
  InvalidateRect(hwndMain,NULL,TRUE);//Перерисовка окна
  }
```

Главное окно приложения предназначено для отображения текущих даты и времени. Для его создания используется расширенная функция CreateWindowEx(), в качестве первого параметра которой можно указать набор расширенных стилей окна. Например, стиль WS_EX_CONTEXTHELP (контекстная справка) включает в заголовок окна знак вопроса. Щелчок по этому знаку изменяет форму курсора мыши, который приобретает вид стрелки с вопросительным знаком. При щелчке таким курсором по какому-либо дочернему окну Windows посылает в это окно сообщение WM_HELP, обработка которого позволяет активизировать интерактивный справочник, связанный с приложением и вывести окно сообщения со справкой по данному вопросу. Мы в нашем примере используем стиль WS_EX_TOPMOST, который заставляет окно приложения всегда оставаться на переднем плане, даже если оно неактивно. Такое окно нельзя заслонить окном другого приложения. Для окна-часов это довольно разумно.

В функции OnCreate() вызовом подпрограммы OutDateTime() определяются и выводятся в окно текущие значения даты и времени, после чего устанавливается единственный в этом приложении таймер на интервал 1с. Таймер будет вызывать ту же подпрограмму OutDateTime() каждую секунду и модифицировать текущее время. "Внеочередной" вызов этой подпрограммы в функции OnCreate() устраняет нахождение на экране пустого окна нашего приложения в течение первой секунды, до первого срабатывания таймера.

При установке таймера в качестве последнего параметра функции SetTimer() указано значение NULL. В этом случае таймер при каждом своем срабатывании (1 раз в секунду)

```
#include "7-1.h"
/*Глобальные переменные*/
char szCurrentTime[40];//Для формирования выводимой строки
HWND hwndMain;
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  char szClassName[]="MainWindow";
  char szTitle[]="Текущее время";
  MSG msg;
  WNDCLASS wc;
  ZeroMemory(&wc,sizeof(wc));
  wc.lpfnWndProc=WndProc;
  wc.hInstance=hInst;
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hbrBackground=GetStockBrush(LTGRAY_BRUSH);//Светло-серый фон окна
  wc.lpszClassName=szClassName;
  RegisterClass(&wc);//Регистрируем класс главного окна
  hwndMain=CreateWindowEx(WS_EX_TOPMOST,szClassName,szTitle,//Создаем окно
    WS_CAPTION|WS_SYSMENU,0,0,XSIZE,YSIZE,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwndMain,SW_SHOWNORMAL);//Показываем окно
  while(GetMessage(&msg,NULL,0,0))//Цикл
    DispatchMessage(&msg);//обработки сообщений
  return 0;
  }
/*Оконная процедура главного окна*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_CREATE,OnCreate);
    HANDLE_MSG(hwnd,WM_TIMER,OnTimer);
    HANDLE_MSG(hwnd,WM_PAINT,OnPaint);
    HANDLE_MSG(hwnd,WM_GETMINMAXINFO,OnGetMinMaxInfo);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
/*Функция обработки сообщения о создании главного окна*/
BOOL OnCreate(HWND hwnd,LPCREATESTRUCT){
  OutDateTime();//Первый немедленный вывод текущего времени
  SetTimer(hwnd,1,1000,NULL);//Установим таймер с периодом 1с
  return TRUE;
  }
/*Функция обработки сообщения WM_PAINT главного окна*/
void OnPaint(HWND hwnd){
  PAINTSTRUCT ps;
  HDC hdc=BeginPaint(hwnd,&ps);
  SetBkMode(hdc,TRANSPARENT);//Сделаем фон символов прозрачным
  TextOut(hdc,5,0,szCurrentTime,strlen(szCurrentTime));//Вывод строки
  EndPaint(hwnd,&ps);
  }
/*Функция обработки сообщения WM_GETMINMAXINFO главного окна*/
void OnGetMinMaxInfo(HWND,LPMINMAXINFO lpmmi){
  lpmmi->ptMinTrackSize.x=XSIZE;//Минимальный
  lpmmi->ptMinTrackSize.y=YSIZE;//и максимальный
  lpmmi->ptMaxTrackSize.x=XSIZE;//размеры главного
  lpmmi->ptMaxTrackSize.y=YSIZE;//окна совпадают
  }
/*Функция обработки сообщения WM_DESTROY главного окна*/
void OnDestroy(HWND hwnd){
  KillTimer(hwnd,1);//Перед завершением уничтожим таймер
```

должна быть определена в программе и содержать процедуру прикладной обработки прерывания от таймера. Если этот параметр указан, то при каждом срабатывании таймера система Windows будет непосредственно, в обход оконной функции, вызывать функцию tmprc(). Если учесть, что сообщения WM_TIMER посылаются функцией Dispatch Message() в оконную функцию в последнюю очередь, когда в очереди сообщений приложения нет других сообщений, а CALLBACK-функция вызывается непосредственно, то ясно, что второй способ должен обеспечить более оперативную обработку сигналов таймера. Если сообщения от таймера предполагается обрабатывать обычным образом, посредством оконной функции, то на месте последнего параметра функции SetTimer() указывается NULL.

Функция SetTimer(), в случае своего успешного выполнения (нормальной установки таймера) возвращает номер данного таймера, т. е. фактически значение второго параметра.

Как уже отмечалось, таймер после своей установки начинает периодически генерировать сообщения с заданным интервалом; если таймер надо остановить, используется функция KillTimer (hwnd, idTimer) с дескриптором окна и номером таймера в качестве параметров. В частности, можно остановить таймер в прикладной функции обработки его сообщения. В этом случае мы получим режим задания однократного временно́го интервала.

Программа календаря-часов

Для иллюстрации создания и использования таймера рассмотрим программу календаря-часов, которая выводит в маленькое окно текущие дату и время и периодически, каждую секунду, обновляет эту информацию. Разумеется, в такой программе мало смысла, так как в составе Windows уже имеется программа индикации и настройки даты-времени, однако разработанный нами алгоритм можно встроить в более сложную программу, которая помимо своей основной работы будет еще указывать в одном из принадлежащих ей окон текущее время.

Результат работы описанного ниже примера приведен на рис. 7.1.

Рис. 7.1. Окно приложения 7-1 на Рабочем столе Windows

```
/*Программа 7-1. Таймеры Windows и получение текущего времени*/
/*Файл 7-1.h*/
#define XSIZE 205//Горизонтальный размер главного окна
#define YSIZE 45//Вертикальный размер главного окна
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
BOOL OnCreate(HWND,LPCREATESTRUCT);
void OnPaint(HWND);
void OnTimer(HWND,UINT);
void OnGetMinMaxInfo(HWND,LPMINMAXINFO);
void OnDestroy(HWND);
void OutDateTime(void);

/*Файл 7-1.cpp*/
#include <windows.h>
#include <windowsx.h>
```

Глава 7
Таймеры Windows и служба времени

В прикладных задачах часто возникает необходимость временно́й синхронизации тех или иных процессов. Представим себе, например, что компьютер используется для управления экспериментальной или производственной установкой. Тогда одной из функций управляющей программы может быть периодический вывод на экран текущего состояния установки – значений действующих в ней электрических или магнитных полей, давлений, температур и т. д. Другой пример "однократной" временно́й синхронизации – запуск (или завершение) какого-либо процесса через заданный интервал времени, например, через 10 с или 15 мин. Если интервал времени необходимо выдержать с высокой точностью, то для его задания приходится использовать специальные аппаратные средства (автономные или связанные с компьютером) – измерители временны́х интервалов; если же высокой точности не требуется, то вполне можно воспользоваться машинным таймером. Непосредственный доступ к физическому машинному таймеру, как и к другим аппаратным средствам компьютера, в Windows запрещен, однако Windows предоставляет прикладному программисту функции, позволяющие установить в приложении требуемое количество *программных таймеров*, с помощью которых приложение может обеспечить временну́ю синхронизацию и задание временны́х интервалов.

Установка таймера

Приложение устанавливает, или активизирует, таймер, вызывая функцию SetTimer(). Эта функция имеет следующий прототип:

```
UINT SetTimer(
  HWND  hwnd,//Дескриптор окна, с которым связан этот таймер
  UINT  idTimer,//Идентификатор таймера
  UINT  uTimeout,//Период срабатывания таймера
  TIMERPROC  tmprc//Прикладная процедура обслуживания таймера
  );
```

Через параметр hwnd системе Windows передается дескриптор окна, для которого устанавливается данный таймер. Сама функция SetTimer() может быть вызвана в любом месте программы, однако, указав дескриптор hwnd, мы связываем таймер с конкретным окном, в оконную функцию которого будут поступать сообщения WM_TIMER.

Параметр idTimer определяет номер, который мы присваиваем данному таймеру. При установке единственного таймера этот номер не имеет значения и на его месте можно указать 0, но при наличии нескольких таймеров номер, присвоенный таймеру, позволяет определить, от какого именно таймера пришло данное сообщение WM_TIMER.

Параметр uTimeout позволяет задать период срабатывания данного таймера. Таймер после своей установки начинает периодически с интервалом uTimeout генерировать сообщения WM_TIMER, поступающие в окно hwnd. Этот временно́й интервал задается в миллисекундах, что, строго говоря, не имеет смысла, так как период отсчета времени программным таймером имеет величину около 50 мс.

Последний параметр, tmprc, дает возможность организовать обслуживание таймера несколько иначе. Параметр представляет собой имя CALLBACK-функции, которая

```
    SetBkMode(hdc,TRANSPARENT);//Прозрачный фон
    return CreateSolidBrush(RGB(180,180,255));//Покрасим список
                                              //светло-синим
    }
  return FALSE;//Остальные элементы не обрабатываем
  }
```

Воспользовавшись дескриптором контекста устройства hdc, поступающим в функции DlgOnColorxxx, мы смогли назначить для каждого списка в отдельности цвет его строк и установить режим прозрачного фона (иначе строки изображались бы на белом фоне). Чтобы покрасить списки по-разному, в программе выполняется анализ параметра hwndCtl – дескриптора конкретного элемента. Разумеется, если оба списка требуется покрасить одинаково, в анализе этого дескриптора нет необходимости, и функция перекрашивания значительно упростится:

```
HBRUSH DlgOnColorListBox(HWND,HDC hdc,HWND,int){
  SetTextColor(hdc,RGB(0,128,0));//Покрасим текст зеленым
  SetBkMode(hdc,TRANSPARENT);//Прозрачный фон
  return CreateSolidBrush(RGB(180,255,180));//Покрасим список светло-зеленым
  }
```

```
/*Нарисуем оси координат*/
  MoveToEx(hdc,0,70,NULL);
  LineTo(hdc,180,70);//Ось X
  MoveToEx(hdc,100,0,NULL);
  LineTo(hdc,100,150); //Ось Y
/*Нарисуем первую эпициклоиду*/
  for(float t=0;t<6.28;t+=0.01){
    int x=(2*cos(t)-3*cos(2*t))*15;
    int y=-((2*sin(t)-3*sin(2*t)))*15;
    Rectangle(hdc,x+100,70-y,x+102,72-y);//Рисуем точки квадратиками
    }
/*Нарисуем вторую эпициклоиду*/
  SelectPen(hdc,hPen);
  for(float t=0;t<6.28;t+=0.01){
    int x=(2*cos(t)-1.5*cos(2*t))*15;
    int y=-((2*sin(t)-1.5*sin(2*t)))*15;
    Rectangle(hdc,x+100,70-y,x+102,72-y);//Рисуем точки квадратиками
    }
  EndPaint(hwnd,&ps);
  }
```

В файле ресурсов описывается простое диалоговое окно без каких-либо элементов управления (кроме заголовка и кнопки системного меню, которые в данном случае включаются в состав диалога по умолчанию). Диалог создается, как обычно, в главной функции WinMain(), состоящей, фактически, из единственного содержательного предложения.

В оконной функции диалогового окна предусмотрена обработка трех сообщений: WM_INITDIALOG, WM_COMMAND и WM_PAINT. В функции DlgOnInitDialog() создается красное перо для вывода второго графика (первый будет нарисован пером по умолчанию, т. е. черным). В функции DlgOnCommand() осуществляется закрытие диалогового окна (и, соответственно, всего приложения) при щелчке по кнопке с крестиком.

Графики эпициклоид рисуются в функции DlgOnPaint(), причем процедура вывода изображений ничем не отличается от используемой для обычных окон. В результате выполнения функции BeginPaint() программа получает доступ к контексту устройства hdc. С помощью функций MoveToEx() и LineTo() в окно выводятся оси координат, смещенные относительно начала окна на 100 пикселов по горизонтали и 70 пикселов по вертикали. Линии осей координат рисуются пером по умолчанию – черным, толщиной 1 пиксел. Далее в цикле вычисляются последовательные значения пар координат x и y и для каждой пары координат в окно выводится квадратик размером 2 пиксела. Поскольку при заданных значениях параметров a и b (2 и 1.5) значения x и y оказываются близкими к 1, обе координаты умножаются на 15, увеличивая тем самым размер выводимого изображения. При выводе в окно (диалога или обычное) графиков следует иметь в виду, что начало отсчета в окнах Windows располагается в верхнем левом углу, поэтому графики надо перевертывать, т. е. умножать их ординаты на –1, и смещать вниз (если в них есть положительные значения). В нашем примере точки (точнее, квадратики) графиков выводятся со смещением 100 пикселов по горизонтали и 70 пикселов по вертикали, что позиционирует график правильно относительно осей координат. Значения ординат y умножаются на –1, хотя в данном случае это излишне, так как график эпициклоиды симметричен относительно оси абсцисс. После вывода первой эпициклоиды в контекст устройства выбирается красное перо и поверх первого рисуется второй график. Завершающей операцией, как и всегда, является вызов функции EndPaint(), возвращающей Windows контекст устройства.

относительная простота программы, недостатком – меньшие возможности. Рассмотрим программу, в которой в диалоговое окно, являющееся главным окном приложения, выводится график некоторой математической функции, например, эпициклоиды, уравнение которой в параметрической форме имеет вид:

$x = a\cos t - b\cos 2t$

$y = a\sin t - b\sin 2t$

Для демонстрации графических возможностей диалога выведем в него две эпициклоиды при различных значениях параметров a и b, нарисовав их перьями разного цвета (рис. 6.37).

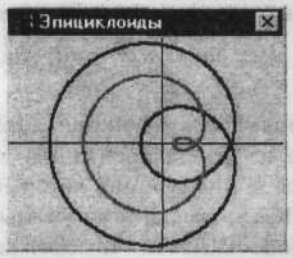


Рис. 6.37. График функции в диалоговом окне

```
/*Пример 6-8. График функции в модальном диалоге*/
/*Файл 6-8.RC*/
#include "6-8.h" Dlg DIALOG 20, 20, 90, 70
  CAPTION "Эпициклоиды"{
  }
/*Файл 6-8.CPP*/
#include <windows.h>
#include <windowsx.h>
#include <math.h>
#include "6-8.h"
HPEN hPen;
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  DialogBox(hInst,"Dlg",NULL,DlgProc);
  return 0;
  }
BOOL CALLBACK DlgProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_INITDIALOG,DlgOnInitDialog);
    HANDLE_MSG(hwnd,WM_COMMAND,DlgOnCommand);
    HANDLE_MSG(hwnd,WM_PAINT,DlgOnPaint);
    default:
      return FALSE;
    }
  }
BOOL DlgOnInitDialog(HWND,HWND,LPARAM){
  hPen=CreatePen(PS_SOLID,1,RGB(255,100,100));
  return TRUE;
  }
void DlgOnCommand(HWND hwnd,int id,HWND,UINT){
    if(id==IDCANCEL)
      EndDialog(hwnd,0);
  }
void DlgOnPaint(HWND hwnd){
  PAINTSTRUCT ps;
  HDC hdc=BeginPaint(hwnd,&ps);
```

```
InvalidateRect(GetParent(hwnd),NULL,TRUE);//Инициируем WM_PAINT
```

Однако можно было ввести, как и для дескриптора экземпляра приложения, глобальную переменную типа HWND. Сохранив в ней значение дескриптора главного окна после его создания вызовом функции CreateWindow(), далее можно было пользоваться этим сохраненным значением.

Наконец, есть и третий способ. Дескриптор родительского окна хранится в той же дополнительной области окна со смещением GWL_HWNDPARENT. Таким образом, получить его в любой точке программы можно с помощью той же функции GetWindowLong():

```
InvalidateRect((HWND)GetWindowLong(hwnd,GWL_HWNDPARENT),NULL,TRUE);
```

Здесь результат, полученный от функции GetWindowLong(), приводится к типу HWND.

Режимы вывода графика

Вывод графика осуществляется в функции OnPaint() обработки сообщений WM_PAINT при условии, что из файла прочитаны данные (переменная bDataOK = = TRUE). Вывод осуществляется в том режиме, который установлен с помощью немодального диалога. Если Mode=DOTS, выводится точечный график так же, как и в предыдущем примере. Если же Mode=CURVE, то прежде всего с помощью функции MoveToEx() выполняется перемещение текущей графической позиции в первую точку графика. По умолчанию текущая позиция устанавливается в точке с координатами (0,0), т. е. в верхнем левом углу экрана. Если ее не сместить в первую точку графика, то график всегда будет начинаться линией, идущей из верхнего левого угла. После этого в цикле из оставшихся 499 шагов функцией LineTo() проводятся прямые линии от текущей позиции к следующей точке графика. Поскольку функция LineTo() не только рисует линию, но и смещает при этом текущую позицию, не возникает необходимости в циклическом использовании функции MoveToEx().

Графика диалогового окна

Диалоговые окна широко используются для управления ходом выполнения программы или режимами ее работы. В этом случае им обычно придают стандартный вид, не заботясь особо о графическом оформлении диалогового окна. Однако в действительности возможности диалоговых окон гораздо шире. В диалоговые окна можно выводить графики и рисунки; пользуясь им, например, как средством наблюдения результатов измерений или расчетов; угрюмую серую поверхность диалогового окна можно покрасить в любой привлекательный цвет; поясняющие надписи диалогового окна также не обязательно должны быть черными. Возможности графического оформления относятся не только к самому диалоговому окну, но также и к отдельным элементам управления: спискам, кнопкам и проч. Рассмотрим некоторые из этих возможностей.

Вывод в диалоговое окно графиков

В диалоговые окна можно выводить любую графическую информацию, как и в обычное окно. При этом начальные действия, такие, как создание кистей или перьев, следует выполнять в ответ на сообщение WM_INITDIALOG, а собственно вывод в диалоговое окно – в ответ на сообщение WM_PAINT. Как и в случае обычных окон, Windows посылает это сообщение в диалоговое окно всякий раз, когда оно появляется из-под закрывающих его окон и, следовательно, требует перерисовки. Преимуществом использования модального диалога для отображения графической информации является

После этого вызовом функции InvalidateRect() инициируется посылка в приложение сообщения WM_PAINT и перерисовывание графика.

Очевидно, что перерисовывать график надо также при смене режима его вывода. Поэтому функция InvalidateRect() вызывается и при поступлении сообщений от кнопок смены режима (с идентификаторами ID_DOTS и ID_CURVE).

Определение значений дескрипторов

Многие функции Windows требуют указания в качестве одного из параметров (обычно первого) того или иного дескриптора. Так, для функции создания немодального диалога CreateDialog() первым параметром должен выступать дескриптор данного экземпляра приложения. Создание диалога обычно осуществляется в ответ на выбор той или иной команды меню, т. е. в функции обработки сообщения WM_COMMAND. Однако дескриптор приложения отсутствует среди аргументов оконной функции и, следовательно, автоматически в функции обработки сообщений не передается. Получить его можно двумя способами. Один из них заключается в объявлении в программе глобальной переменной и помещении в нее значения требуемого дескриптора в той точке программы, где он оказывается известен. В настоящей программе для этого использовалась глобальная переменная hI. Значение дескриптора экземпляра приложения помещалось в нее в самом начале функции WinMain(), перед началом действий по регистрации класса окна.

Другой метод связан с использованием функций Windows, предназначенных специально для получения тех или иных дескрипторов. Естественно, все действующие дескрипторы хранятся в Windows, и существуют функции для их извлечения. Конкретно дескриптор экземпляра приложения хранится в специальной области памяти, связанной с любым окном (так называемая дополнительная память окна). Получить информацию из этой области памяти в 32-разрядном приложении можно с помощью функции GetWindowLong(), в качестве первого параметра которой указывается дескриптор окна, а в качестве второго – символическое обозначение требуемого смещения в дополнительной памяти. Для дескриптора экземпляра приложения это смещение равно GWL_HINSTANCE.

Таким образом, для создания диалога можно было поступить так, как сделано в программе, т. е. сохранить в переменной hI дескриптор в какой-либо точке функции WinMain(), а при создании диалога воспользоваться этой переменной:

```
hSettingsBox=CreateDialog(hI,"SettingsDlg",hwnd,DlgProc);
```

но можно было обойтись без глобальной переменной, а значение дескриптора приложения получить из дополнительной области окна в тот момент, когда он оказывается нужен:

```
hSettingsBox=CreateDialog((HINSTANCE)GetWindowLong(hwnd,GWL_HINSTANCE),
                          "SettingsDlg",hwnd,DlgProc);
```

Приведение результата выполнения функции GetWindowLong() к типу HINSTANCE здесь необходимо, так как эта функция для любого смещения возвращает "безликий" результат типа LONG, а для функции CreateDialog() требуется параметр типа HINSTANCE.

Схожая ситуация имеет место и для дескрипторов окон. Правда, в функциях обработки сообщений *главного* окна проблем с этим дескриптором не возникает, так как он передается во все эти функции в качестве одного из параметров, а вот в порожденных окнах, в частности в диалогах, он неизвестен. В нашем случае дескриптор главного окна нужен для функции InvalidateRect(), которая вызывается в оконной функции диалога, но должна организовать перерисовку главного окна с графиком.

Для получения дескриптора главного окна, которое для диалога является родительским, мы воспользовались функцией Windows GetParent():

переменной Mode и, вообще говоря, не связан с внешним видом кнопок. Изменение состояния кнопок программным образом, путем посылки сообщений *в диалог*, не изменяет текущий режим. Для того чтобы изменить режим вывода графика, надо щелкнуть по кнопке мышью. В этом случае программа получит сообщение *из диалога* и, обрабатывая это сообщение, может, в частности, изменить режим.

Схожая ситуация имеет место с альтернативностью кнопок. Вполне возможно, послав одинаковые сообщения обеим кнопкам, обе их нажать или отпустить. Программное нажатие одной кнопки не отжимает автоматически другую. Альтернативность кнопок со стилем BS_AUTORADIOBUTTON проявляется только при воздействии на них мышью. Эти воздействия перехватываются Windows, которая, помимо посылки сообщения из диалога в программу, нажимает кнопку, по которой щелкнули, и отжимает другую кнопку, придавая им свойство альтернативности.

Следующее действие по инициализации диалога заключается в заполнении пустого пока списка с идентификатором ID_SCALE значениями допустимых масштабов. Для этого в элемент управления ID_SCALE посылаются сообщения LB_ADDSTRING с указанием в качестве второго параметра адреса пересылаемой в список текстовой строки S:

```
SendDlgItemMessage(hwnd,ID_SCALE,LB_ADDSTRING,0,(LPARAM)S);//Пересылка строки
```

Строка S предварительно формируется с помощью функции wsprintf(), которая преобразует числовые значения масштабов (массив nScales[8]) в символы с добавлением знака процента. Оба эти действия осуществляются в цикле из восьми шагов по всем элементам массива nScales[ ].

Наконец, последнее сообщение LB_SETCURSEL посылается в список для выделения текущего масштаба изображения. Как уже отмечалось, для удобства в программе хранится не только текущее значение масштаба (в переменной CurrentScales), но и индекс nScaleIndex текущего значения в массиве масштабов. Массив масштабов, ради удобства вывода этих чисел в список, заполнен значениями масштабов в процентах; в переменной же CurrentScales хранится коэффициент преобразования исходных данных в виде доли единицы (естественно, в переменной с плавающей точкой). При запуске приложения обе эти переменные получают начальные значения, характеризующие масштаб 100 %.

Второе место программы, где в диалоговое окно посылается сообщение, – это функция DlgOnCommand(), конкретно фрагмент этой функции, обрабатывающий сообщение от списка с идентификатором ID_SCALE. Весь этот фрагмент представляет собой условный блок, выполняемый в том случае, если код уведомления, поступающий в программу через последний параметр codeNotify функции DlgOnCommand(), равен константе LBN_SELCHANGE. Такой код уведомления посылается при изменении пользователем номера выбранной строки списка. Другие коды уведомления посылаются, например, при получении списком фокуса ввода (при управлении списком от клавиатуры) или при потере фокуса.

Сообщение, посылаемое в программу при смене строки списка, не содержит сведений о том, какая строка выбрана. Для того чтобы получить номер новой выбранной строки, программа должна с помощью функции SendDlgItemMessage() послать в список сообщение-запрос LB_GETCURSEL. Результат, возвращаемый в этом случае функцией SendDlgItemMessage(), одновременно является номером выделенной строки списка и индексом массива масштабов nScales[8].

Программа извлекает из массива масштабов соответствующее этому индексу значение масштаба, преобразует его в дробное значение (1% = 0.01) и выполняет заново масштабирование массива данных nBuf[ ], записывая новые значения в массив nBufScaled[ ].

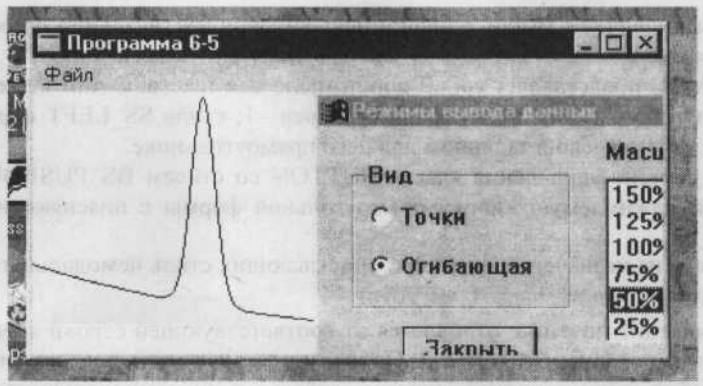


Рис. 6.36. Поведение дочернего (CHILD) окна

Взаимодействие с немодальным диалогом

Создание немодального диалога осуществляется в функции OnCommand() при выборе пользователем команды меню Файл➢Режимы, за которой закреплен идентификатор MI_SETTINGS. Диалог создается функций CreateDialog(), имеющей те же параметры, которые имеет и функция создания модального диалога DialogBox(), однако, в отличие от последней, возвращающей дескриптор созданного диалогового окна. Этот дескриптор (hSettingsBox) используется в программе в качестве флага существования диалога для защиты от его повторного создания. Перед вызовом функции CreateDialog() проверяется флаг hSettingsBox; если он не равен нулю, т. е. если диалог уже создан, функция CreateDialog() не выполняется.

Полезно иметь в виду, что дескрипторы окон не обнуляются системой при уничтожении окон; поэтому в функции DlgOnCommand() в случае прихода идентификаторов IDOK или IDCANCEL не только закрывается окно диалога функцией Destroy Window(), но и обнуляется его дескриптор hSettingsBox, что дает возможность после закрытия диалога открыть его повторно.

Использование дескриптора в качестве флага не имеет в данном случае каких-то серьезных преимуществ перед организацией специального флага кроме того, что мы экономим на одном предложении его установки.

Рассмотрим программное взаимодействие с диалогом с помощью функции SendDlgItemMessage(), посылающей в диалог сообщения. Большая часть вызовов этой функции приходится на функцию DlgOnInitDialog(), выполняющую инициализацию диалога после его создания системой. Вызов

```
SendDlgItemMessage(hwnd,Mode,BM_SETCHECK,1,0L);//Нажать кнопку текущего режима
```

приводит к показу в нажатом состоянии кнопки того режима ("Точки" или "Огибающая"), который сохранен в переменной Mode. Эта переменная перечислимого типа была объявлена таким образом, что ее допустимые значения совпадают с идентификаторами двух альтернативных кнопок переключения режима; это дало нам возможность указывать ее в качестве второго параметра функции SendDlgItemMessage(). Значение четвертого параметра (1) указывает на то, что кнопка должна быть показана в нажатом состоянии; указание в качестве этого параметра нуля отжало бы кнопку.

Важно отметить, что описанные манипуляции с кнопкой относятся только к ее внешнему виду; они не определяют ни текущий режим, ни состояние второй кнопки. Текущий режим, анализируемый и используемый в функции OnPaint(), определяется значением

чувствительны к щелчкам мыши, список имеет линейку вертикальной прокрутки и окружен простой рамкой.

Класс STATIC представляет собой просто поле для текста. С ним не осуществляется взаимодействия поэтому его идентификатор равен –1; стиль SS_LEFT определяет позиционирование текста влево в заданном для него прямоугольнике.

Наконец, элемент управления класса BUTTON со стилем BS_PUSHBUTTON представляет собой нажимаемую кнопку прямоугольной формы с поясняющим текстом на ней.

Остановимся на назначении констант, описывающих стиль немодального диалога:

```
STYLE WS_VISIBLE | WS_SYSMENU | WS_POPUP
```

Это предложение почти не отличается от соответствующей строки сценария модального диалога из программ 6-1 или 6-2. Однако для модального диалога минимально необходимый набор характеристик диалога действует по умолчанию и предложение STYLE, как уже упоминалось, можно было опустить. В диалоге же немодальном это предложение должно присутствовать обязательно.

Константа WS_VISIBLE делает диалог видимым, т. е. как бы автоматически вызывает для него функцию Windows ShowWindow(). Без этой константы окно диалога не появится на экране.

Константа WS_SYSMENU определяет наличие в строке заголовка диалога кнопки его закрытия ✕. В нашем случае эта кнопка не обязательна, так как в самом диалоге мы предусмотрели кнопку "Закрыть", однако удобнее создавать диалоговые окна стандартного, привычного вида.

Константа WS_POPUP объявляет диалоговое окно "всплывающим". Вообще порожденные окна Windows, т. е. окна (как диалоговые, так и обычные), созданные внутри главного окна приложения, бывают двух типов: всплывающие, описываемые константой WS_POPUP, и дочерние (константа WS_CHILD). Между ними имеется ряд различий в поведении. Всплывающие окна при их перемещении по экрану могут выходить за рамки родительского окна (рис. 6.35). Дочерние окна всегда остаются внутри породившего их родительского окна (рис. 6.36). Имеются и другие различия, в частности заголовок всплывающего окна, если его сделать активным, выделяется ярким цветом, а у дочернего окна заголовок всегда тусклый (см. те же рисунки).

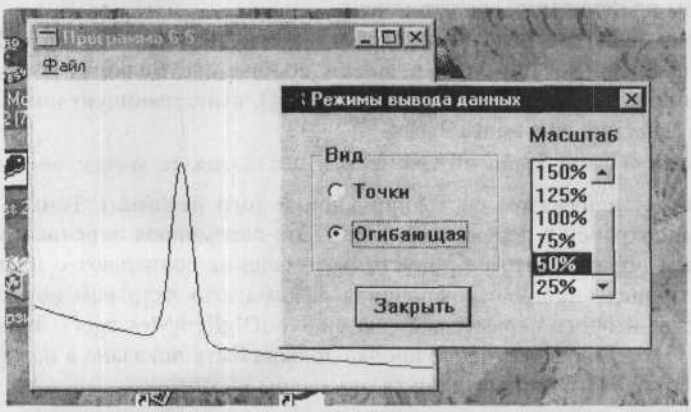


Рис. 6.35. Поведение всплывающего (POPUP) окна

Описание элементов управления в файле ресурсов

Модальные и немодальные диалоги практически ничем не различаются по своим возможностям и правилам описания, за исключением того, что модальный диалог полностью перехватывает управление и не позволяет до своего завершения работать с другими элементами окна, а немодальный не имеет этого ограничения. Придание диалогу свойства модальности или немодальности осуществляется вызовом соответствующей функции создания диалога – DialogBox() для модального диалога и CreateDialog() для немодального. Кроме того, по-разному осуществляется закрытие диалога: модальный диалог закрывается функцией EndDialog(), а немодальный – более универсальной функцией DestroyWindow(). Содержимое же диалогового окна, описываемое в файле ресурсов, и для того и для другого диалога может быть каким угодно. Таким образом, приводимые ниже правила описания элементов управления немодального диалогового окна для приложения 6-7 (рис. 6.34) можно с таким же успехом использовать и при разработке модального диалога.

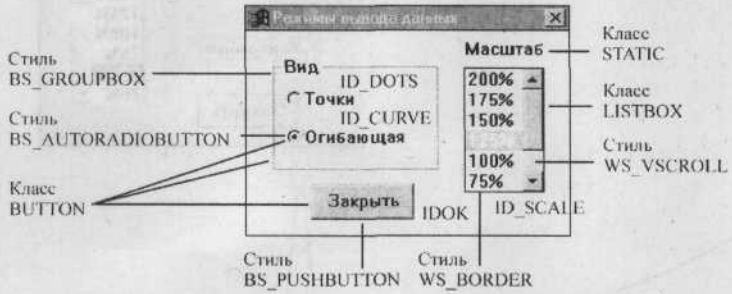


Рис. 6.34. Элементы управления в окне диалога с указанием их классов, стилей и идентификаторов

Все предложения файла ресурсов используют стандартный формат описания элементов управления. Они начинаются с ключевого слова CONTROL, за которым следуют: поясняющий текст, идентификатор (значение которого определено в файле 6-7.H), предопределенный класс элемента управления, стиль (точнее, комбинация необходимых стилей) и координаты.

Класс BUTTON с указанием стиля BS_AUTORADIOBUTTON описывает альтернативные кнопки. В группе таких кнопок в любой момент времени может быть нажата только одна, причем, если пользователь нажимает какую-то кнопку, она утапливается и подсвечивается, а нажатая ранее кнопка автоматически отжимается. В нашем диалоге всего две такие кнопки и они по умолчанию образуют группу. Если кнопок больше и они по смыслу должны образовывать несколько групп (например, группы "Вид", "Цвет", "Форма" и др.), в стиль первой кнопки из каждой группы надо добавить бит WS_GROUP.

Класс BUTTON с указанием стиля BS_GROUPBOX позволяет включить в диалоговое окно рамку с заголовком, в которую затем будут помещены альтернативные кнопки или другие элементы управления. К рамке не выполняется программных обращений, и в качестве идентификатора принимается значение –1, однако "заголовок" у нее есть – это надпись "Вид" в верхнем разрыве рамки.

Класс LISTBOX позволяет вывести на экран список некоторых данных (у нас – значений масштаба графика в процентах). Комбинация стилей LBS_NOTIFY | WS_VSCROLL | WS_BORDER определяет следующие свойства списка: элементы списка

```
  case ID_SCALE://Сообщение от списка; проанализируем извещение
    if(codeNotify==LBN_SELCHANGE){//Если изменилось выделение в списке
      nScaleIndex=(int)SendDlgItemMessage(hwnd,ID_SCALE,LB_GETCURSEL,0,0L);
      nCurrentScales=
           (float)nScales[nScaleIndex]/100;//Преобразование строки в число
      for(int i=0;i<500;i++)//Масштабирование спектра в соответствии
        nBufScaled[i]=nBuf[i]*nCurrentScales;//с новым масштабом
      InvalidateRect(GetParent(hwnd),NULL,TRUE);//Инициируем WM_PAINT
      }//Конец if
  }//Конец switch(id)
}
```

Результат работы этой программы приведен на рис. 6.33.

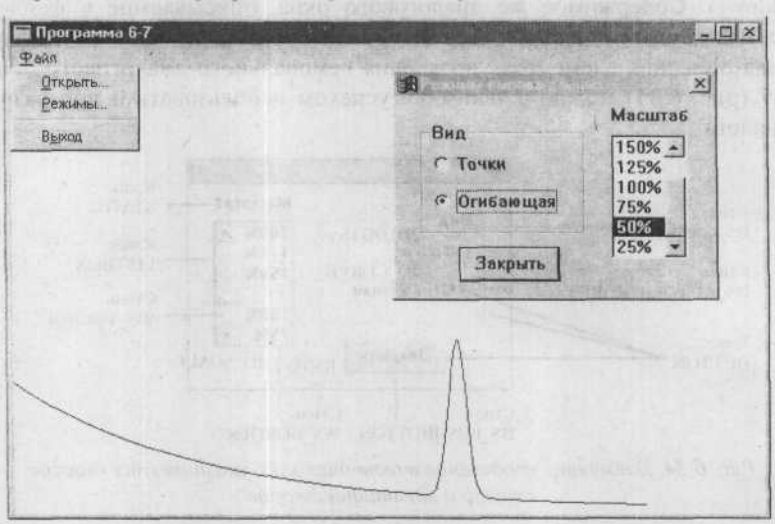


Рис. 6.33. Немодальный диалог в главном окне

Рассмотрим отличия приведенного примера от программы 6-6.

Файлы заголовков и ресурсов

Включение в приложение немодального диалогового окна с программными блоками обслуживания его элементов управления заметно усложнило программу. В заголовочный файл 6-7.H включены определения дополнительных констант – идентификаторов элементов управления диалогом. Там же определен перечислимый тип GraphModes (см. описание перечислимых типов в гл. 3). Переменные этого типа смогут принимать всего два значения. Одно из них называется DOTS и равно значению символьной константы ID_DOTS; другое называется CURVE и совпадает со значением символьной константы ID_CURVE. Переменная типа GraphModes (в программе она названа Mode) служит для хранения текущего режима изображения графика – в виде точек (DOTS) или огибающей (CURVE). Наконец, в файл 6-7.H добавлены прототипы прикладных функций обслуживания диалогового окна.

В файле ресурсов 6-7.RC расширен список команд меню "Файл" (появился пункт "Режимы" с идентификатором MI_SETTINGS для активизации немодального диалога) и задан сценарий немодального диалога SettingsDlg. Рассмотрим описание элементов управления диалоговым окном.

```
  HDC hdc=BeginPaint(hwnd,&ps);//Получение контекста устройства
  HPEN hOldPen=SelectPen(hdc,hPen);
  GetClientRect(hwnd,&r);//Получение координат рабочей области
  if(bDataOK){
    switch(Mode){
      case DOTS:{
        for(int i=0;i<500;i++)
          SetPixel(hdc,i,r.bottom-5-nBufScaled[i],RGB(0,0,255));//Вывод точек
        break;
        }//Конец case DOTS
      case CURVE:{
        MoveToEx(hdc,0,r.bottom-5-nBufScaled[0],NULL);
        for(int i=1;i<500;i++)
          LineTo(hdc,i,r.bottom-5-nBufScaled[i]);//Вывод отрезков
        }//Конец case CURVE
      }//Конец switch (Mode)
    }//Конец if(bDataOK)
  SelectPen(hdc,hOldPen);
  EndPaint(hwnd,&ps);
  }
/*Функция обработки сообщения WM_DESTROY*/
void OnDestroy(HWND){
  DeleteObject(hPen);
  PostQuitMessage(0);
  }
/*Оконная процедура немодального диалога*/
BOOL CALLBACK DlgProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_INITDIALOG,DlgOnInitDialog);
    HANDLE_MSG(hwnd,WM_COMMAND,DlgOnCommand);
    default:
      return FALSE;
    }
  }
/*Функция обработки сообщения WM_INITDIALOG для диалогового окна*/
BOOL DlgOnInitDialog(HWND hwnd,HWND,LPARAM){
  char S[10];//Строка для обмена данными со списком LISTBOX
  SendDlgItemMessage(hwnd,Mode,BM_SETCHECK,1,0L);//"Нажать" кнопку режима
  for (int i=0;i<8;i++){//Заполнение списка LISTBOX строками текста
    wsprintf(S,"%d%%",nScales[i]);//Пересылка в строку числа и знака %
    SendDlgItemMessage(hwnd,ID_SCALE,LB_ADDSTRING,0,(LPARAM)S);
    }
  SendDlgItemMessage(hwnd,ID_SCALE,LB_SETCURSEL,nScaleIndex,0L);
  return TRUE;
  }
/*Функция обработки сообщений WM_COMMAND для диалогового окна*/
void DlgOnCommand(HWND hwnd,int id,HWND,UINT codeNotify){
  switch(id){//Код элемента управления
    case IDOK://Сообщение от кнопки "Закрыть"
    case IDCANCEL://Сообщения от системного меню диалога
      DestroyWindow(hwnd);//Уничтожение окна немодального диалога
      hSettingsBox=NULL;//Сбросим в нуль его дескриптор
      break;
    case ID_DOTS://Сообщение от кнопки "Точки"
      Mode=DOTS;//Установим режим "Точки"
      InvalidateRect(GetParent(hwnd),NULL,TRUE);//Инициируем WM_PAINT
      break;
    case ID_CURVE://Сообщение от кнопки "Огибающая"
      Mode=CURVE;//Установим режим "Огибающая"
      InvalidateRect(GetParent(hwnd),NULL,TRUE);//Инициируем WM_PAINT
      break;
```

```
  wc.lpszClassName=szClassName;
  RegisterClass(&wc);
  HWND hwnd=CreateWindow(szClassName,szTitle,WS_OVERLAPPEDWINDOW,
           0,0,600,400,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_SHOWNORMAL);
  hPen=CreatePen(PS_SOLID,1,RGB(0,0,255));
  while(GetMessage(&msg,NULL,0,0)){
    TranslateMessage(&msg);
    DispatchMessage(&msg);
    }
  return 0;
  }
/*Оконная процедура главного окна*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);
    HANDLE_MSG(hwnd,WM_PAINT,OnPaint);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
/*Функция обработки сообщения WM_COMMAND от меню*/
void OnCommand(HWND hwnd,int id,HWND,UINT){
  OPENFILENAME ofn;//Структура для функции GetOpenFileName
  char szFilter[]="Файлы данных(*:DAT)\0*.dat\0Все файлы(*.*)\0*.*\0";
  DWORD nCnt;//Вспомогательная переменная для файловых операций
  switch(id){
    case MI_OPEN:
      ZeroMemoryt(&ofn, sizeof(OPENFILENAME));
      ofn.lStructSize=sizeof(OPENFILENAME);//Размер структуры
      ofn.hwndOwner=hwnd;//Дескриптор главного окна-владельца диалога
      ofn.lpstrFilter=szFilter;//Строка с шаблонами имен искомых файлов
      ofn.lpstrFile=szFile;//Буфер для имени файла
      ofn.nMaxFile=sizeof(szFile);//Его размер
      ofn.Flags=OFN_PATHMUSTEXIST|OFN_FILEMUSTEXIST;
      if(GetOpenFileName(&ofn)){
        HANDLE hFile=CreateFile(szFile,GENERIC_READ,0,0,OPEN_EXISTING,0,NULL);
        if(hFile==INVALID_HANDLE_VALUE)break;//Если не удалось открыть файл
        ReadFile(hFile,nBuf,2*500,&nCnt,NULL);//Чтение из файла
        CloseHandle(hFile);
        bDataOK=TRUE;//Данные имеются, можно выводить
        for(int i=0;i<500;i++)//Выполним начальное масштабирование спектра
          nBufScaled[i]=nBuf[i]*nCurrentScales;
        InvalidateRect(hwnd,NULL,TRUE);//Инициируем WM_PAINT
        break;
        }
      else break;//Если файл не выбран
    case MI_EXIT:
      DestroyWindow(hwnd);
      break;
    case MI_SETTINGS:
      if(hSettingsBox==NULL)//Если диалог еще не открыт, создадим его
        hSettingsBox=CreateDialog(hI,"SettingsDlg",hwnd,DlgProc);
      break;
    }
  }
//Функция обработки сообщения WM_PAINT
void OnPaint(HWND hwnd){
  RECT r;
  PAINTSTRUCT ps;
```

```
void OnCommand(HWND,int,HWND,UINT);
void OnPaint(HWND);
void OnDestroy(HWND);
/*Прототипы функций для диалогового окна*/
BOOL CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM);
BOOL DlgOnInitDialog(HWND,HWND,LPARAM);
void DlgOnCommand(HWND,int,HWND,UINT);
```

```
/*Файл 6-7.rc*/
#include "6-7.h"
Main MENU{
  POPUP "&Файл"{
    MENUITEM "&Открыть...",MI_OPEN
    MENUITEM "&Режимы...",MI_SETTINGS
    MENUITEM SEPARATOR
    MENUITEM "В&ыход",MI_EXIT
    }
  }
SettingsDlg DIALOG 150,0,125,80
STYLE WS_VISIBLE | WS_SYSMENU | WS_POPUP
CAPTION "Режимы вывода данных"{
  CONTROL "Вид", -1, "BUTTON", BS_GROUPBOX, 10, 10, 65, 45
  CONTROL "Точки", ID_DOTS, "BUTTON", BS_AUTORADIOBUTTON, 15, 20, 45, 14
  CONTROL "Огибающая", ID_CURVE, "BUTTON", BS_AUTORADIOBUTTON, 15, 35, 55, 14
  CONTROL "",ID_SCALE,"LISTBOX",BS_NOTIFY|WS_VSCROLL|WS_BORDER,85,15,30,50
  CONTROL "Масштаб", -1, "STATIC", SS_LEFT, 85, 3, 40, 10
  CONTROL "Закрыть", IDOK, "BUTTON", BS_PUSHBUTTON, 25, 60, 40, 14
  }
```

```
/*Файл 6-7.cpp*/
/*Операторы препроцессора*/
#include <windows.h>
#include <windowsx.h>
#include "6-7.h"
/*Глобальные переменные*/
char szClassName[]="MainWindow";
char szTitle[]="Программа 6-5";
char szFile[128];//Для спецификации выбранного файла
short nBuf[500];//Буфер для чтения массива из файла
BOOL bDataOK=FALSE;//Если TRUE, данные прочитаны в буфер nBuf
HINSTANCE hI;//Дескриптор приложения
short nBufScaled[500];//Буфер для масштабированного массива
int nScales[8]={200,175,150,125,100,75,50,25};//Массив масштабов
int nScaleIndex=4;//Текущее (и начальное) значение индекса в массиве масштабов
float nCurrentScales=1;//Текущее (и начальное) значение масштаба
HWND hSettingsBox;//Дескриптор немодального диалога
GraphModes Mode=DOTS;//Режим наблюдения и его начальное значение
HPEN hPen;//Дескриптор нового пера
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  MSG msg;
  WNDCLASS wc;
  hI=hInst;
  ZeroMemory (&wc,sizeof(wc));
  wc.style=CS_VREDRAW;
  wc.lpfnWndProc=WndProc;
  wc.hInstance=hInst;
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
  wc.lpszMenuName="Main";
```

Третий параметр является флагом перерисовки фона. Если его значение равно TRUE, фон окна перерисовывается, что приводит к стиранию прежнего изображения. Если флаг равен FALSE, новое изображение рисуется поверх старого. В нашем случае, когда на экран выводятся отдельные точки в произвольных местах, перерисовывать весь фон, т. е. стирать точки старого графика, необходимо. В тех случаях, однако, когда на экран выводятся прямоугольные элементы изображения, которые целиком перекрывают старое изображение (например, знакоместа с буквами или цифрами), можно обойтись без перерисовывания фона, что сэкономит процессорное время и, главное, предотвратит мерцание изображения, возникающее при перерисовке окна.

Вывод на экран графика

Вывод в главное окно приложения точечного графика осуществляется в функции OnPaint() обработки сообщения WM_PAINT. Здесь после получения с помощью функции BeginPaint() дескриптора контекста устройства вызывается функция GetClientRect(), которая возвращает в структурную переменную r типа RECT координаты рабочей области главного окна, точнее говоря, ее размеры, так как координаты начала рабочей области всегда считаются равными нулю. Размер рабочей области по вертикали нам нужен для того, чтобы выводимый на экран график строился относительно нижней границы окна (значения r.bottom), независимо от текущего размера окна.

Вывод графика осуществляется в цикле из 500 шагов при условии наличия прочитанных из файла данных (переменная bDataOK=TRUE) функцией отображения точек SetPixel(). В качестве параметров этой функции указывается дескриптор контекста устройства, x- и y-координаты выводимой точки и ее цвет (с помощью макроса RGB()). Ордината точки с x-координатой i вычисляется как разность между нижней текущей границей окна r.bottom и высотой выводимой кривой в этой точке nBuf[i]. Из полученного значения вычитается 5, чтобы поднять график на 5 пикселов над нижней границей окна.

Немодальный диалог

Как уже отмечалось ранее, немодальные диалоги отличаются от модальных тем, что при их выводе на экран остальные элементы управления окна не блокируются, что позволяет одновременно работать и с немодальным диалогом, и с другими средствами управления приложением. Для иллюстрации процедуры создания немодального диалога и работы с ним добавим к программе предыдущего примера немодальное диалоговое окно, позволяющее с помощью содержащихся в нем элементов управления выполнять следующие действия:

- масштабирование выводимого графика;
- переключение вида графика – отдельные точки или точки, соединенные линиями (огибающая).

```
/*Программа 6-7. Немодальный диалог с элементами управления*/
/*Файл 6-7.h*/
/*Константы*/
#define MI_OPEN 101
#define MI_EXIT 102
#define MI_SETTINGS 103
#define ID_DOTS 201
#define ID_CURVE 202
#define ID_SCALE 203
/*Определение нового типа GraphModes*/
enum GraphModes{DOTS=ID_DOTS,CURVE=ID_CURVE};
/*Прототипы функций для главного окна*/
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
```

менной, а такие переменные автоматически инициализируются нулями, однако и это неприятно: после запуска программы на экране появится прямая линия, соответствующая нулевым значениям всех данных.

Итак, буфер nBuf заполнен данными, которые надо вывести на экран. Однако, как было показано в гл. 5, вывод в окно любых изображений допустим только в ответ на сообщение WM_PAINT, так как лишь в этом случае при всяких естественных для Windows манипуляциях с окном (изменение размеров окна, свертывание его в пиктограмму и обратное развертывание, "вытаскивание" окна из-под других окон и т. д.) изображение в окне будет перерисовываться правильно. Если изображение в окне носит статический, неизменяемый характер, как это было в предыдущих примерах, то никаких проблем не возникает, так как при появлении на переднем плане закрытой ранее части окна Windows сама посылает в приложение сообщение WM_PAINT и приложение перерисовывает закрытое ранее окно. Если же изображение должно формироваться динамически, например в ответ на выбор команды меню, как это имеет место в рассматриваемой программе, то мы должны иметь возможность, подготовив данные для вывода изображения, *сами* инициировать сообщение WM_PAINT, в функции обработки которого изображение и будет построено. Для этой важнейшей операции в Windows предусмотрено несколько функций, самой распространенной из которых является InvalidateRect().

Инициирование сообщения WM_PAINT

Функция InvalidateRect(), с использованием которой мы уже столкнулись при рассмотрении программы 6-1, объявляет указанную в ней прямоугольную область определенного окна поврежденной (invalid – недействительный, не имеющий законной силы), что заставляет Windows послать в это окно сообщение WM_PAINT, которое приводит к перерисовке содержимого окна. Таким образом, в Windows предусмотрены два механизма инициирования сообщения WM_PAINT. Сама Windows посылает в приложение это сообщение каждый раз, когда скрытое ранее окно или его часть появляется на экране и, соответственно, требует перерисовки; кроме того, программа в любой момент может вызовом функции InvalidateRect() объявить окно поврежденным, что также приведет к посылке в приложения сообщения WM_PAINT.

Функция InvalidateRect() имеет следующий прототип:

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

Первый параметр определяет дескриптор того окна, которое требует перерисовки. При наличии в главном окне приложения внутренних, порожденных окон, мы имеем возможность инициировать перерисовку любого из них, для чего надо иметь их дескрипторы. С этой целью при создании каждого порожденного окна функцией CreateWindow() следует позаботиться о сохранении возвращаемого этой функцией дескриптора окна.

Второй параметр дает возможность указать координаты прямоугольной области, которая добавляется к так называемой области обновления, требующей перерисовки. Windows накапливает информацию обо всех областях, входящих в область обновления, и, когда приложение вызывает функцию BeginPaint(), Windows передает в программу координаты этой области. Практически, однако, вызов функции InvalidateRect() сразу приводит к посылке в окно сообщения WM_PAINT и о накапливании недействительных областей в области обновления говорить не приходится.

Если значение этого параметра равно NULL, то областью обновления считается вся рабочая область. Перерисовка занимает заметное время и к тому же может приводить к мерцанию изображения. Поэтому во всех случаях желательно определять область обновления минимального размера. В нашем случае изображение графика может занимать весь экран и приходится перерисовывать всю рабочую область.

Стандартный диалог "Открытие файла"

Вывод на экран стандартного диалога Windows "Открытие файла" осуществляется вызовом функции GetOpenFileName(), для успешного выполнения которой необходимо подготовить структуру типа OPENFILENAME (переменная ofn в программе). Эта структурная переменная объявлена в начале функции OnCommand() обработки сообщения WM_COMMAND от команд меню. Там же объявлена символьная строка-фильтр szFilter, в которую в определенном формате записываются строки, которые будут выведены в окно "Тип файлов" стандартного диалога, вместе с шаблонами спецификаций отбираемых файлов. В программе предусмотрена возможность показа файлов с расширением .DAT (шаблон *.dat), а также всех файлов (шаблон *.*). При желании список возможных типов имен файлов можно расширить. Элементы строки-фильтра разделяются двоичными нулями, которые в символьной строке обозначаются как \0. Двоичный нуль помещается также и в конец строки (которая, таким образом, завершается двумя нулями).

Структура ofn обнуляется, после чего некоторые элементы получают инициализирующие значения. Назначение большинства элементов структуры должно быть ясно из текста программы. В элемент ofn.lpstrFile записывается адрес буфера, куда функция GetOpenFileName() вернет полную спецификацию выбранного пользователем файла; принятое нами значение элемента ofn.Flags указывает, что при вводе пользователем неправильного имени каталога или файла на экран будет выведено предупреждающее сообщение вида "Такого пути нет" или "Невозможно найти файл".

Функция GetOpenFileName() вызывается в операторе if, условный блок которого выполняется, только если функция GetOpenFileName() отработала успешно и в буфере szFile появилась законная спецификация выбранного файла. Это дает возможность правильно завершить этот фрагмент программы и в том случае, когда пользователь, открыв стандартный диалог, затем закрыл его, так и не выбрав в нем файл.

Внутри условного блока выбранный файл открывается 32-разрядной функцией CreateFile() (это не опечатка – обобщенная функция CreateFile() служит и для создания и для открытия файла) только для чтения (параметр GENERIC_READ). Эта функция требует в качестве параметров указатель на адрес буфера с именем файла и в случае успешного выполнения возвращает дескриптор файла типа HANDLE, который в дальнейшем будет служить для нас идентификатором данного файла (работа с файлами в 32-разрядных приложениях Windows будет подробнее описана в гл.11). Если файл открыть не удалось, функция CreateFile() возвращает значение INVALID_HANDLE_VALUE, поэтому предложение

```
if(hFile==INVALID_HANDLE_VALUE)break;
```

позволяет обойти все последующие строки, если по какой-то причине файл не был открыт.

Тридцатидвухразрядная функция ReadFile() читает из файла в буфер nBuf заданное количество байтов (500 данных по 2 байта), после чего файл закрывается функцией CloseHandle().

После выполнения всех операций с файлом предложением

```
bDataOK=TRUE;
```

устанавливается флаг bDataOK, который говорит о том, что буфер nBuf заполнен данными и, следовательно, есть что выводить на экран. В дальнейшем, в функции OnPaint(), вывод содержимого nBuf будет осуществляться, только если этот флаг установлен. Это полезная предосторожность, которая предотвратит вывод на экран "мусора" в ответ на первое сообщение WM_PAINT, которое поступает в приложение при открытии главного окна, когда буфер nBuf еще не заполнен. Правда, мы объявили nBuf глобальной пере-

При повторном вызове стандартного диалога "Открытие файла" на экран выводится содержимое вновь выбранного файла; старый график при этом стирается.

Общая структура программы вполне очевидна. В оконной функции обрабатываются 3 сообщения Windows: WM_COMMAND, WM_PAINT и WM_DESTROY. Сообщение WM_COMMAND поступает в приложение при выборе пользователем одной из двух имеющихся команд меню. При выборе команды "Открыть" выполняются все необходимые действия по работе со стандартным диалогом "Открытие файла" и с самим файлом данных. При выборе команды "Выход" приложение закрывается обычным образом, вызовом функции DestroyWindow(). Функция обработки сообщения WM_PAINT обеспечивает вывод графика на экран.

Некоторое отличие от предыдущих примеров заключается в указании элемента стиля класса окна

```
wc.style=CS_VREDRAW;
```

Указание этой константы приводит к тому, что при каждом изменении размеров окна по вертикали Windows посылает в окно сообщение WM_PAINT, которое приводит к *полной* перерисовке всего окна (а не только области вырезки). Такой режим необходим в тех случаях, когда изображение в окне рисуется относительно его *нижней* границы, т. е. требуется, чтобы при перемещении по экрану нижней границы окна все изображение перемещалось вместе с ней. Если изображение рисуется относительно *правой* границы окна, то необходимо включать в стиль окна константу CS_HREDRAW; в этом случае окно будет перерисовываться целиком при каждом изменении его размеров по горизонтали.

Для проверки работоспособности программы 6-4 необходимо подготовить файлы данных с разумным содержимым. Ниже приводится текст программы, позволяющей это сделать. В ней вычисляются значения функции $y = f(x)$, представляющей собой сумму спадающей экспоненциальной кривой $c1*\exp(-x/c2)$ и нормальной кривой Гаусса $c3*\exp(-c4(x-c5)^2)$. Подбором констант *c1...c5* можно в широких пределах изменять форму получающейся кривой.

Разумеется, читатель может написать свою программу для вычисления значений любой другой функции. Нужно только, чтобы в файле содержалось ровно 500 чисел и чтобы их значения не превышали размера нашего окна по вертикали.

```
/*Файл spectrs.cpp. Создание тестовых файлов данных*/
#include <windows.h>
#include <math.h>//Ради exp()
char fname[]="sp1.dat";//Имя создаваемого файла
short nBuf[50];//Массив с тестовыми данными для записи в файл
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  /*Заполним массив тестовыми данными*/
  int c1,c2,c3,c5;
  float c4,x;
  c1=200; c2=150; c3=250; c4=0.01; c5=350;
  for(int i=0;i<500;i++){
    x=i;//Преобразуем счетчик цикла в число с плавающей точкой
    nBuf[i]=c1*exp(-x/c2)+c3*exp(-c4*(x-c5)*(x-c5));//Вычисляем значения
    }
/*Создадим новый файл и запишем в него тестовый массив*/
  HANDLE hFile=CreateFile(fname,GENERIC_WRITE,0,0,CREATE_ALWAYS,0,NULL);
  DWORD nCnt;//Счетчик записанного
  WriteFile(hFile,nBuf,2*500,&nCnt,NULL);//Запись в файл
  return 0;
  }
```

масштабирования данных. Возможный вариант программы для подготовки файлов данных для этой программы будет описан в конце раздела.

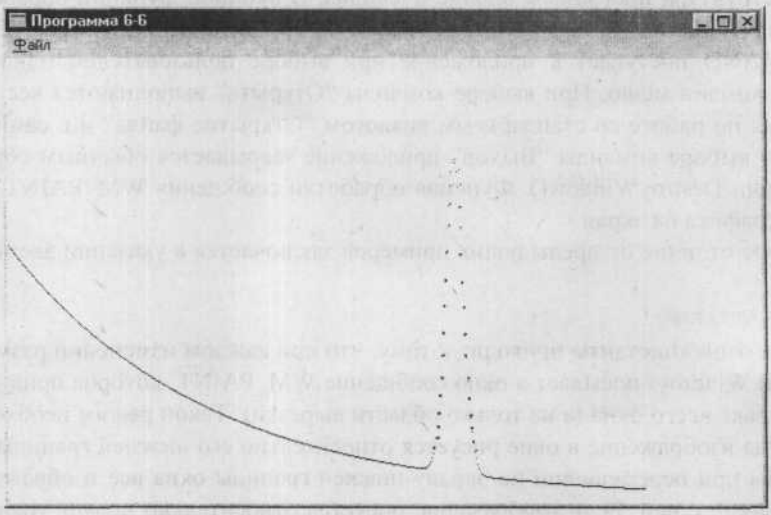


Рис. 6.31. Вывод точечного графика

При выборе пользователем пункта "Открыть" вызывается стандартный диалог Windows "Открытие файла" (рис. 6.32), обеспечивающий поиск требуемых файлов по всем дискам и каталогам. Предполагается, что пользователь заранее подготовил файлы с данными, подлежащими выводу на экран. После выбора файла он открывается, его содержимое (500 коротких целых чисел) считывается в буфер программы и вызовом функции InvalidateRect() инициируется посылка в главное окно приложения сообщения Windows WM_PAINT.

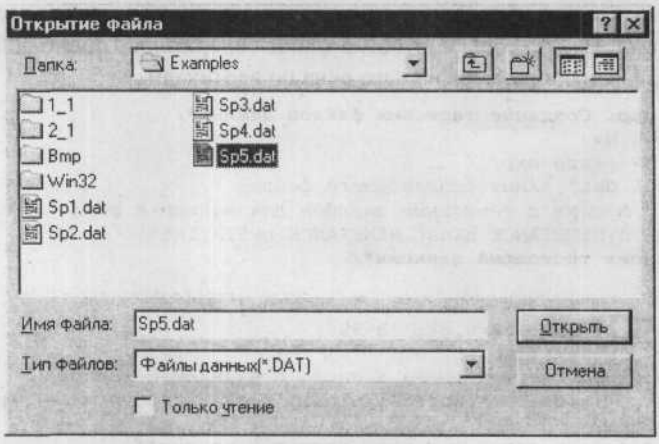


Рис. 6.32. Стандартный диалог Windows "Открытие файла"

В функции обработки сообщения WM_PAINT в цикле из 500 шагов просматривается содержимое буфера с данными и каждое данное выводится на экран в виде точки, образуя графическое представление массива данных.

```
/*Функция обработки сообщений WM_COMMAND от меню*/
void OnCommand(HWND hwnd,int id,HWND,UINT){
  OPENFILENAME ofn;//Структура для функции GetOpenFileName
  char szFilter[]="Файлы данных(*.DAT)\0*.dat\0Все файлы(*.*)\0*.*\0";
  DWORD nCnt;//Для файловых операций
  switch(id){
    case MI_OPEN://Выбран пункт "Открыть"
      ZeroMemory (&ofn, sizeof(OPENFILENAME));//Обнуляем ofn
      ofn.lStructSize=sizeof(OPENFILENAME);//Размер структуры
      ofn.hwndOwner=hwnd;//Дескриптор главного окна-владельца диалога
      ofn.lpstrFilter=szFilter;//Строка с шаблонами имен искомых файлов
      ofn.lpstrFile=szFile;//Буфер для имени файла
      ofn.nMaxFile=sizeof(szFile);//Его размер
      ofn.Flags=OFN_PATHMUSTEXIST|OFN_FILEMUSTEXIST;
      if(GetOpenFileName(&ofn)){//Если имя файла выбрано
        HANDLE hFile=
              CreateFile(szFile,GENERIC_READ,0,0,OPEN_EXISTING,0,NULL);
        if(hFile==INVALID_HANDLE_VALUE)//Если не удалось открыть файл,
          break;//не выполнять дальнейшие операции
        ReadFile(hFile,nBuf,2*500,&nCnt,NULL);//Чтение из файла
        CloseHandle(hFile);//Закроем файл
        bDataOK=TRUE;//Данные имеются, можно выводить
        InvalidateRect(hwnd,NULL,TRUE);//Инициируем сообщение WM_PAINT
        break;//Конец case MI_OPEN
        }//Конец if(GetOpenFileName)
      else break;//Если файл не выбран
    case MI_EXIT:
      DestroyWindow(hwnd);
    }
  }
/*Функция OnPaint обработки сообщения WM_PAINT*/
void OnPaint(HWND hwnd){
  RECT r; //Структура, описывающая координаты прямоугольника
  PAINTSTRUCT ps;//Структура с характеристиками рабочей области
  HDC hdc=BeginPaint(hwnd,&ps);//Получение контекста устройства
  GetClientRect(hwnd,&r);//Получение координат рабочей области
  int i=0;//Номер читаемого слова в буфере и x-координата на экране
  while((i<500)&bDataOK) //Берем из буфера по одному слову до конца
    SetPixel(hdc,i,r.bottom-5-nBuf[i++],RGB(0,0,255));//Вывод точки
  EndPaint(hwnd,&ps);
  }
/*Функция обработки сообщения WM_DESTROY*/
void OnDestroy(HWND){
  PostQuitMessage(0);
  }
```

Исходные тексты приложения включают стандартный набор: заголовочный файл 6-6.H с объявлениями символических констант и прототипов функций, файл ресурсов 6-6.RC с описанием простой линейки меню (единственное меню "Файл" с командами "Открыть" и "Выход") и, наконец, файл 6-6.CPP с исходным текстом программы.

Программа 6-4 выводит на экран содержимое указанного ей файла данных в виде точечного графика, в котором каждое данное отображается синей точкой, расстояние которой от низа окна равно значению конкретного данного (рис. 6.31). Файл данных должен содержать набор из 500 целых коротких (двухбайтовых) чисел, не превышающих высоты главного окна в пикселах, чтобы график не вышел за пределы экрана. Для упрощения программы в ней не предусмотрено никаких средств анализа размера файла данных или

```
#define MI_EXIT 102
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
void OnCommand(HWND,int,HWND,UINT);
void OnPaint(HWND);
void OnDestroy(HWND);
```

```
/*Файл 6-6.rc*/
#include "6-6.h"
Main MENU{
  POPUP "&Файл"{
    MENUITEM "&Открыть...",MI_OPEN
    MENUITEM SEPARATOR
    MENUITEM "В&ыход",MI_EXIT
    }
  }
```

```
/*Файл 6-6.cpp*.
/*Операторы препроцессора*/
#include <windows.h>
#include <windowsx.h>
#include "6-6.h"
/*Глобальные переменные */
char szClassName[]="MainWindow";
char szTitle[]="Программа 6-4";
char szFile[128];//Для спецификации выбранного файла
short nBuf[500];//Буфер для чтения из файла 2-байтовых данных
BOOL bDataOK=FALSE;//Если TRUE, данные прочитаны в буфер nBuf
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  MSG msg;
  WNDCLASS wc;
  ZeroMemory (&wc, sizeof(wc));
  wc.style=CS_VREDRAW; //Вывод относительно нижней границы рабочей области
  wc.lpfnWndProc=WndProc;
  wc.hInstance=hInst;
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
  wc.lpszMenuName="Main";
  wc.lpszClassName=szClassName;
  RegisterClass(&wc);
  HWND hwnd=CreateWindow(szClassName,szTitle,WS_OVERLAPPEDWINDOW,
                         0,0,600,400, HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_SHOWNORMAL);
  while(GetMessage(&msg,NULL,0,0))  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
    }
  return 0;
  }
/*Оконная процедура главного окна*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);
    HANDLE_MSG(hwnd,WM_PAINT,OnPaint);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
```

даваемого элемента управления. Выделив элемент щелчком левой клавиши мыши, можно перетаскиванием изменить его размер и положение на поле диалога.

После чернового заполнения диалогового окна элементами управления можно приступить к подгонке их размещения. Для этого используется меню Layout (размещение), которое вместе с открывающимися меню второго уровня предоставляет богатые возможности по выравниванию размеров и положения выделенных групп элементов. На рис. 6.30 приведено раскрытое меню Layout вместе с подменю Align (выравнивание) в процессе подгонки положения двух нажимаемых кнопок.

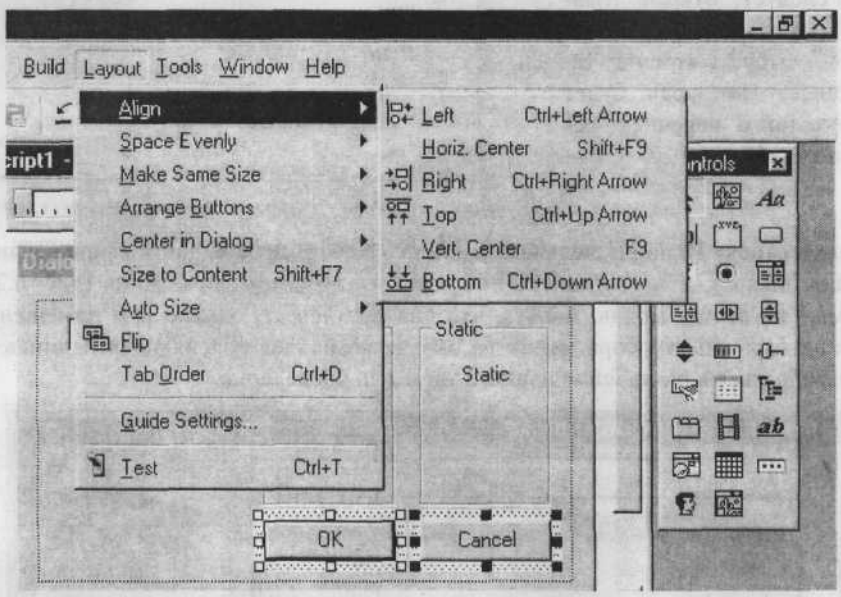


Рис. 6.30. Раскрытое меню Layout с подменю Align

После завершения работы над диалоговым окном полученный сценарий необходимо сохранить в файле с расширением .RC с тем, чтобы в дальнейшем включить его в проект отлаживаемого приложения.

Стандартные модальные диалоги Windows

В системе Windows имеется группа модальных диалогов, предназначенных для обслуживания стандартных ситуаций: поиска открываемых или сохраняемых файлов, поиска и замены отдельных слов и др. Все эти диалоги хранятся в файле COMMDLG.DLL и характеризуются схожими правилами вызова и обслуживания. Каждый диалог создается выделенной для этого функцией Windows, перед вызовом которой необходимо заполнить соответствующую этому диалогу структуру данных. В настоящем разделе будет проиллюстрировано использование стандартного диалога, служащего для поиска открываемого файла. Рассмотрим программу, которая читает с диска файл с данными и выводит их на экран в виде точечного графика.

```
/*Программа 6-6. Стандартные модальные диалоги Windows*/
/*Файл 6-6.h*/
#define MI_OPEN 101
```

Работа с редактором ресурсов Visual C++ 6.0

Общие принципы визуального составления сценария ресурсов в Visual C++ и Borland C++ одинаковы, хотя в технике работы имеются значительные различия. Для создания нового файла ресурсов следует, открыв главное окно среды разработки Visual C++ 6.0, выбрать команду Insert➢Resource. На экран будет выведено окно с перечнем ресурсов (рис. 6.28).

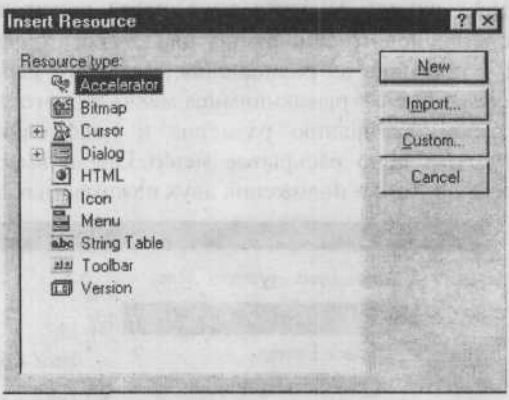


Рис. 6.28. Окно с перечнем допустимых ресурсов

Выделим строку Dialog и щелкнем по кнопке New. В правой части кадра появится заготовка для диалога, а также панель пиктограмм элементов управления (рис. 6.29). По пиктограмме не всегда можно понять, что она обозначает, однако при помещении над значком пиктограммы курсора мыши на экране возникает поясняющая надпись, помогающая разобраться в назначении данного элемента управления.

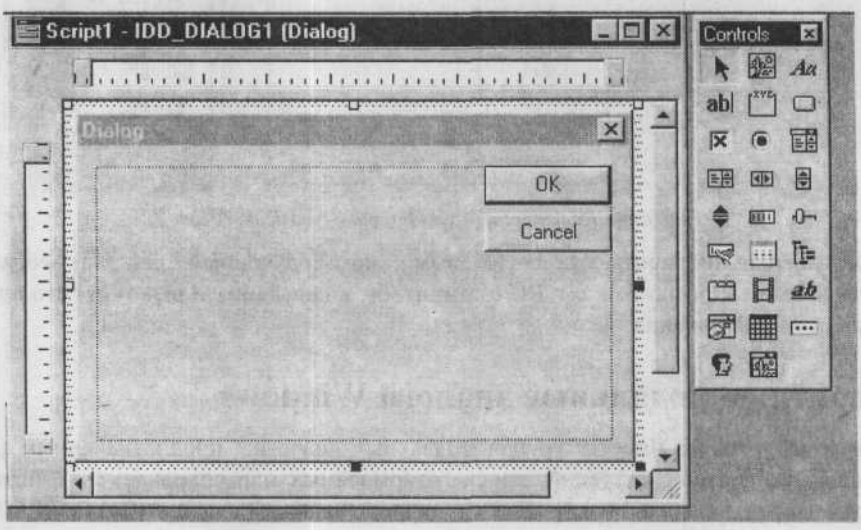


Рис. 6.29. Рабочее окно редактора ресурсов с заготовкой диалога

Дальнейшие действия по заполнению диалогового окна необходимыми элементами управления осуществляются в общем так же, как и в редакторе ресурсов Borland C++. Для помещения требуемого элемента управления в диалоговое окно следует щелкнуть по его пиктограмме на панели элементов. Курсор примет форму креста, который можно перенести на поле диалогового окна и при нажатой левой клавише мыши "нарисовать" там прямоугольник, положение и размеры которого будут соответствовать размещению соз-

Следующее действие – подгонка размеров и положения элементов управления. Для этого, выделив, например, обе групповые рамки (щелчками левой клавиши мыши при нажатой клавише Shift), выберем в меню команду Dialog➢Size и в появившемся окне (рис. 6.25) отметим альтернативные кнопки Grow to largest, если мы хотим выровнять размеры рамок по наибольшей из них. Как видно из рис. 6.25, имеется возможность и численной установки ширины (Width) и высоты (Height) элементов, хотя это не очень удобно.

Выбрав затем команду меню Dialog➢Align➢Controls, можно выровнять выделенные элементы требуемым образом: по левой или правой границе, по центру диалогового окна и другими предусмотренными способами. Можно также перемещать элементы управления (поодиночке или целыми группами) вручную, как мышью, так и с помощью клавиш со стрелками на клавиатуре. Применив команды выравнивания к двум нажимаемым кнопкам, получим в результате прилично выглядящее диалоговое окно (рис. 6.26).

Для просмотра характеристик элементов управления следует воспользоваться командой меню Dialog➢Show➢➢Property Inspector, которая выводит на экран окно свойств выделенного элемента (рис. 6.27). В этом окне можно назначить каждому элементу как содержательный заголовок, так и более наглядное символическое обозначение идентификатора. Впрочем, редактирование заголовков и идентификаторов удобнее выполнить позже в обычном текстовом редакторе.

Теперь полученный сценарий необходимо сохранить в файле с расширением .RC (командой меню File➢Save as), после чего его можно подсоединить к проекту и использовать в разрабатываемом приложении.

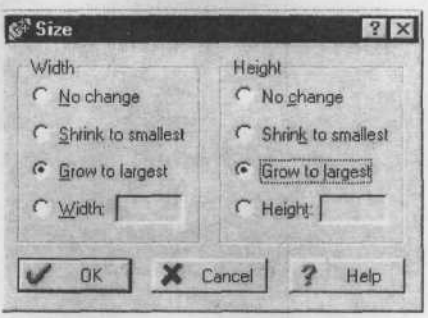


Рис. 6.25. Окно установки размеров выделенных элементов управления

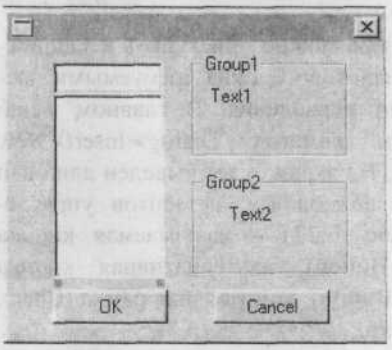


Рис. 6.26. Окончательное расположение элементов управления в диалоговом окне

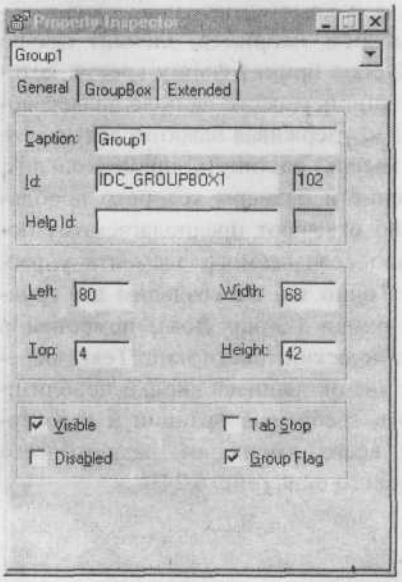


Рис. 6.27. Окно свойств

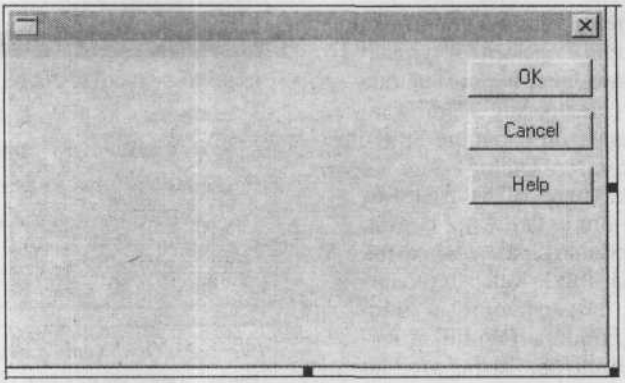


Рис. 6.22. Заготовка диалогового окна

Теперь можно приступить к заполнению диалогового окна требуемыми элементами управления. В главном меню выберем команду Dialog➢Insert➢New Control. На экран будет выведен длинный список возможных элементов управления (рис. 6.23) – нажимаемая кнопка (Push Button), альтернативная кнопка (Radio Button), контрольная рамка (Check Box) и др.

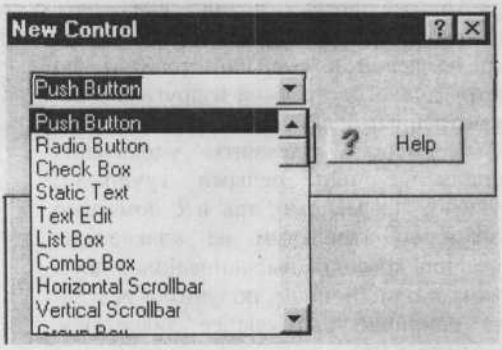


Рис. 6.23. Список элементов управления

Выберем, например, элемент Combo Box. Курсор примет форму креста, который можно перенести на поле диалогового окна и, удерживая нажатой левую клавишу мыши, растянуть прямоугольник, положение и размеры которого приблизительно отвечают предполагаемому положению создаваемого элемента управления. Точно так же создадим две групповые рамки (Group Box), поместим в них статический текст (Static Text), перетащим две оставшиеся кнопки приблизительно в требуемые позиции и в довершение всего уменьшим размер всего диалогового окна (рис. 6.24).

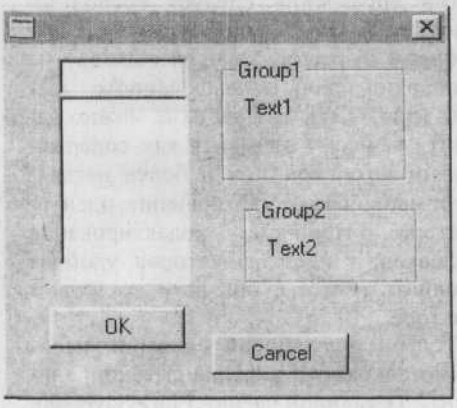


Рис. 6.24. Элементы управления вчерне перенесены в диалоговое окно

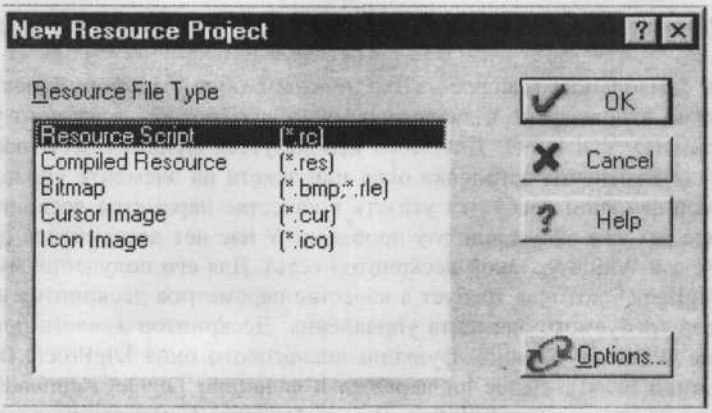


Рис. 6.19. Начальный кадр редактора ресурсов

Выберем пункт Resource Script (файл сценария ресурсов с расширением .RC). На экран будет выведено рабочее окно редактора ресурсов (рис. 6.20).

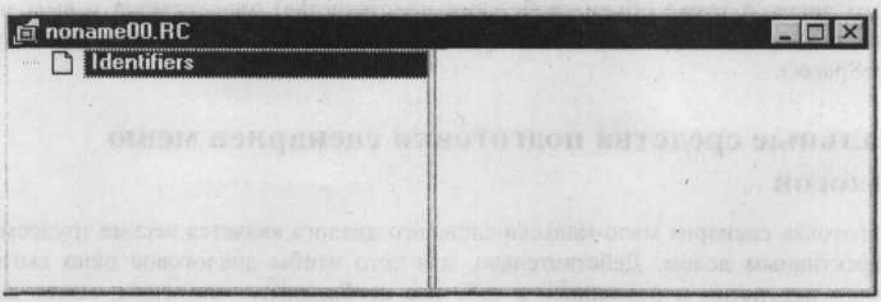


Рис. 6.20. Рабочее окно редактора ресурсов

Теперь надо задать тип создаваемого ресурса. Для этого в главном меню выберем команду Resource➢New. На экран будет выведен обширный список возможных ресурсов, которые можно включить в файл ресурсов – растровое изображение, курсор, диалог, меню и пр. (рис. 6.21). Мы хотим создать диалоговое окно, поэтому выберем пункт DIALOG.

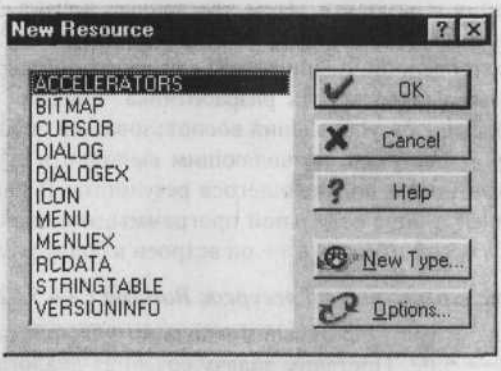


Рис. 6.21. Перечень ресурсов

На экране появится действующая по умолчанию заготовка – диалоговое окно с тремя кнопками (рис. 6.22). Поскольку мы хотим иметь в диалоговом окне только две кнопки, удалим лишнюю, выделив ее щелчком левой клавиши мыши и нажав после этого клавишу Delete.

включения в строку символа \n к ней добавляется последнее данное – имя файловой системы.

Получив в символьном массиве szText нужным образом сформированную строку с данными, можно отправить ее в диалоговое окно, конкретно – в элемент управления с идентификатором ID_VOLINFO. Для этого используется функция SetWindowText(), которая служит для изменения заголовка окна или текста на элементе управления диалогом. Однако этой функции требуется указать в качестве параметра дескриптор адресуемого окна. Ранее мы уже обсуждали эту проблему. У нас нет дескриптора области с текстом (у нас нет, а в Windows такой дескриптор есть). Для его получения мы используем функцию GetDlgItem(), которая требует в качестве параметров дескриптор всего диалога и идентификатор требуемого элемента управления. Дескриптор диалога поступил к нам еще при вызове Windows оконной функции диалогового окна DlgProc() (первый параметр этой функции hwnd). Далее он перешел в функцию DlgOnCommand() и из нее – в нашу функцию-подпрограмму FilInfo(). Вызов функции GetDlgItem() в лучших традициях C++ включен прямо в качестве первого параметра вызова функции SetWindowText().

Физическая информации о выбранном диске (размеры сектора, кластера и всего логического диска, а также объем свободного пространства) определяется и выводится в окно диалога схожим образом. Для получения этих данных используется функция GetDiskFreeSpace().

Визуальные средства подготовки сценариев меню и диалогов

Подготовка сценария мало-мальски сложного диалога является весьма трудоемким и малоэффективным делом. Действительно, для того чтобы диалоговое окно выглядело достаточно аккуратно и размещало в себе все необходимые элементы, макет диалога приходится рисовать на миллиметровке, измерять линейкой получившиеся размеры и смещения и заносить затем эти данные в файл .RC. После первого прогона программы оказывается, что какие-то размеры выбраны неудачно, и сценарий приходится подправлять с повторной компиляцией после каждого исправления до тех пор, пока результаты не будут удовлетворять разработчика. Гораздо удобнее для размещения в диалоговом окне элементов управления воспользоваться специальным программным средством – редактором ресурсов, позволяющим выполнить эту работу наглядным образом с визуальным контролем получающегося результата. В пакете Borland C++ 4.5 редактор ресурсов выполнен в виде отдельной программы, носящей имя Resource Workshop; в пакетах Borland C++ 5.02 и Visual C++ он встроен в среду разработки.

Работа с редактором ресурсов Borland C++ 5.02

Рассмотрим основные правила визуальной подготовки сценария диалога в IDE Borland C++ 5.02. Поставим задачу создания диалога, схожего с использованным в программе 6-5 и содержащего комбинированный список, две групповые рамки с текстами и две нажимаемые кнопки. С помощью меню среды разработки Borland C++ 5.02 выберем команду File➢New➢Resource Project. На экран будет выведен начальный кадр редактора ресурсов (рис. 6.19), в котором следует выбрать тип создаваемого файла – растровое изображение (.BMP), изображение курсора (.CUR) и др.

типа LPVOID, который можно преобразовать в указатель любого конкретного типа. Это преобразование (приведение типа) мы и совершаем, указав перед вызовом функции в скобках требуемый тип LPSTR – указатель на символы (то же самое, что и char*).

Выделив память под строку с обозначениями дисков, мы вторичным вызовом функции GetLogicalDriveStrings() заполняем эту строку (указываемую в качестве второго параметра функции) конкретной информацией, относящейся к данному компьютеру.

Далее проблема заключается в том, чтобы разбить полученную строку на составляющие ее обозначения отдельных дисков и перенести эти обозначения в комбинированный список. Эта операция выполняется с помощью вспомогательного указателя szNext, который перемещается по строке в цикле while, останавливаясь в каждом шаге на очередном первом байте за нулевым, т. е. как раз на начале следующего обозначения диска. Вызовом функции SendDlgItemMessage() элементу управления с идентификатором ID_COMBO, т. е. комбинированному списку, посылается сообщение CB_ADDSTRING, добавляющее в список очередную строчку (начинающуюся с адреса szNext). По завершении цикла (обнаружением последнего нуля) вызовом функции HeapFree() выделенный блок памяти освобождается и поступает в пул кучи.

Вывод конкретных данных в информационные поля диалогового окна осуществляется в функции DlgOnCommand().

Обработка сообщений от комбинированного списка (идентификатор ID_COMBOBOX) выполняется лишь в том случае, если так называемый код уведомления, поступающий через параметр CodeNotify, равен символической константе CBN_SELCHANGE. Равенство этой константе говорит о том, что пользователь изменил выбранный элемент списка. Тем самым устраняются ненужные перерисовывания диалога при повторном выборе той же строки в списке.

При поступлении сообщения от списка прежде всего следует узнать, какая строка списка выбрана. Для этого списку сначала посылается сообщение CB_GETCURSEL; возвращаемое функцией SendDlgItemMessage() значение соответствует номеру выбранной строки. Затем списку посылается сообщение CB_GETLBTEXT с этим номером в качестве первого параметра; теперь SendDlgItemMessage() возвращает через второй параметр саму выбранную строку, т. е. обозначение диска ("A:\", "B:\" и т. д.). Имея обозначение выбранного диска, можно приступить к получению о нем необходимой информации. Действия по получению этой информации и выводу ее в диалоговое окно вынесены в функцию-подпрограмму FillInfo().

Для получения информации о логическом диске предназначена функция Windows GetVolumeInformation(). Она возвращает метку тома, его серийный номер, даваемый тому при его форматировании программой FORMAT, набор системных флагов, указывающих, в частности, разрешены ли в именах файлов строчные буквы и не является ли диск сжатым (программой типа DRIVESPACE), а также обозначение файловой системы (FAT, NTFS и др.).

Системные флаги мы для простоты не выводим; 3 остальных параметра надо объединить в одну символьную строку, включив в нее знаки перевода строки. Для формирования строки используется символьный массив szText. Первое данное (метка) заносится в переменную szText операцией копирования strcpy(). Далее с помощью функции C++ strcat(), осуществляющей сцепление (конкатенацию) строк, в конец строки szText добавляется символ перевода строки \n. Второе данное (серийный номер) сначала преобразуется в символьную форму функцией wsprintf() и заносится в промежуточный символьный массив szTemp. Затем строка szTemp сцепляется со строкой szText. Наконец, после

константу "общего назначения" WS_VCSROLL, добавляющую к списку вертикальную линейку прокрутки, а также специфическую для комбинированных списков константу CBS_DROPDOWNLIST, определяющую появление выпадающего списка и возможность вводить в индикаторное окно требуемую информацию с клавиатуры.

Главная функция WinMain() имеет всего одну содержательную строку – вызов функции DialogBox() открытия модального диалога. Все время до завершения работы программы приложение "висит" внутри этой функции; однако блокирование программы не мешает обработке сообщений, поступающих в диалоговое окно и активизирующих оконную процедуру DlgProc().

Обслуживание диалога распадается на два этапа. В процессе инициализации диалога (в ответ на сообщение WM_INITDIALOG) выполняется определение состава дисков в системе и заполнение комбинированного списка. Вывод же информации о конкретных дисках производится при поступлении сообщений WM_COMMAND, приходящих от комбинированного списка при выборе в нем того или иного диска.

Рассмотрим сначала действия, выполняемые на первом этапе.

Для получения информации об имеющихся в системе логических дисках предусмотрена функция GetLogicalDriveStrings(). При первом вызове (с нулевыми параметрами) она возвращает размер строки, в которую поместятся обозначения всех наличных дисков в формате

```
'A:\', 0, 'B:\' , 0,'C:\', 0, 'D:\',...0,0
```

Поскольку заранее размер этой строки неизвестен, так как на разных компьютерах физический жесткий диск может быть разделен на разное число логических и, кроме того, могут отсутствовать второй дисковод B: для съемных дисков или привод компакт-диска, память под эту строку следует выделить динамически, взяв ее из кучи процесса. Кучей (heap) называется свободная область памяти, предназначенная для выделения из нее программой по мере необходимости участков нужного размера. Размер кучи по умолчанию – 1 Мбайт, однако ее можно увеличить на этапе компоновки программы. Первоначально куча существует только в линейном адресном пространстве; физическая память под нее не выделяется. По мере запроса программой памяти из кучи система выполняет отображение части линейного адресного пространства на физическую память, передавая в программу информацию об отображенном участке. Такое динамическое выделение памяти является одним из краеугольных камней в функционировании операционной системы и широко используется в программах в тех случаях, когда размеры данных могут быть определены только в процессе выполнения.

Выделение памяти из кучи требует двух действий. Сначала надо вызовом функции GetProcessHeap() получить указатель (типа HANDLE) на кучу процесса. Имея этот указатель, можно вызвать функцию HeapAlloc(), которая выполняет собственно выделение памяти из состава кучи. В качестве параметров функции HeapAlloc() выступают указатель кучи, константа HEAP_ZERO_MEMORY, задающая обнуление выделяемой памяти (без этой константы память при выделении не будет очищена), и размер требуемой области в байтах. Функция возвращает указатель (линейный адрес) выделенного блока памяти.

На первый взгляд, полученный указатель должен дать нам возможность работать с выделенным блоком. Вспомним, однако, что указатели на данные разных типов сами относятся к разным типам указателей: указатель на символы – это один тип, а указатель, например, на целые числа – совсем другой. Поскольку программе может потребоваться память под любые данные, функция HeapAlloc() возвращает "обобщенный" указатель

каторы. В качестве этих идентификаторов использованы произвольные символические обозначения ID_VOLINFO, ID_DRIVETYPE и ID_DISKINFO, значения которых (тоже произвольные) определены в заголовочном файле.

Как видно из рис. 6.17, всю текстовую информацию, выводимую в диалоговое окно, можно разбить на две категории. К первой категории относятся статические строки текста ("Тип диска", "Информация о томе", "Метка тома" и др.), которые целиком описаны в файле ресурсов и не изменяются по ходу выполнения программы. Составляя сценарий диалога, следует позаботиться о том, чтобы эти строки были расположены в диалоговом окне разумным и изящным образом, что достигается назначением им соответствующих координат. В нашем примере некоторые из статических строк объединены в группы и помещены внутрь рамок (элемент управления класса BUTTON со стилем BS_GROUPBOX). Всего у нас две такие группы, одна из которых включает три строки, а другая – четыре. В файле ресурсов каждая из этих двух групп описана как один элемент управления. Для того чтобы поясняющие надписи располагались на отдельных строках, в конце каждой надписи добавлен символ перевода строки \n. Размеры прямоугольников для этих надписей выбраны так, чтобы в них поместилось требуемое число строк.

Вторую категорию текстов составляют строки, формируемые программно и выводимые в диалоговое окно с помощью функции Windows SetWindowText(), неоднократно вызываемой в подпрограмме FillnfoO. Как можно увидеть из рис. 6.17, этих строк также 7.

Для вывода всех этих семи строк в диалоговом окне предусмотрено лишь два элемента управления – относительно большие прямоугольные области: одна (с позиционированием текста влево) с идентификатором ID_VOLINFO и вторая (с позиционированием вправо) с идентификатором ID_DISKINFO (рис. 6.18). Поэтому в программе нам придется объединить полученные данные в две группы и вставить между отдельными данными символы перевода строки \n, чтобы данные изобразились столбиками, как показано на рис. 6.17. При этом числовые данные о диске, занимающие разное число десятичных разрядов, с помощью стиля SS_RIGHT выровнены вправо.

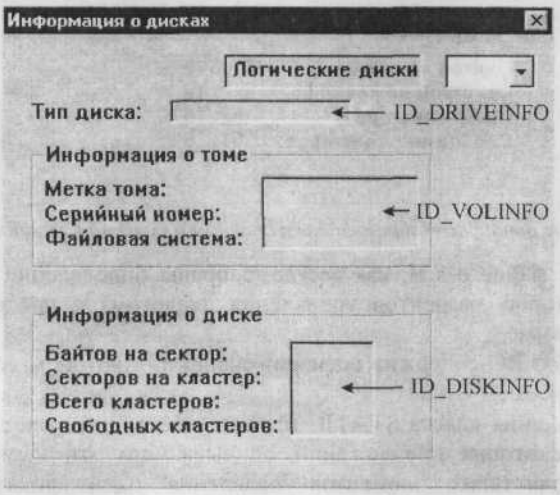


Рис. 6.18. Элементы управления в диалоговом окне программы 6-5

Элемент управления класса COMBOBOX отображается в виде комбинированного списка с определенными характеристиками. Стиль этого элемента управления включает

```
  strcat(szText,szTemp);//Добавление строки с серийным номером
  strcat(szText,"\n");//Добавление перевода строки
  strcat(szText,szFat);//Добавление имени файловой системы
  SetWindowText(GetDlgItem(hwnd,ID_VOLINFO),szText);//Вывод в диалог
/*Получим информацию о физических характеристиках диска*/
  DWORD dwSectorsPerCluster,dwBytesPerSector,dwFreeClusters,dwClusters;
  GetDiskFreeSpace(szVol,&dwSectorsPerCluster,&dwBytesPerSector,
                   &dwFreeClusters,&dwClusters);
  wsprintf(szTemp,"%d",dwBytesPerSector);//Размер сектора в символы
  strcpy(szText,szTemp); //Копирование в строку вывода szText
  strcat(szText,"\n");//Добавление перевода строки
  wsprintf(szTemp,"%d",dwSectorsPerCluster);//Размер кластера в символы
  strcat(szText,szTemp);//Добавление к строке вывода
  strcat(szText,"\n");//Добавление перевода строки
  wsprintf(szTemp,"%d",dwClusters);//Число кластеров в символы
  strcat(szText,szTemp); //Добавление к строке вывода
  strcat(szText,"\n");//Добавление перевода строки
  wsprintf(szTemp,"%d",dwFreeClusters);//Число свободных кластеров в символы
  strcat(szText,szTemp); //Добавление к строке вывода
  SetWindowText(GetDlgItem(hwnd,ID_DISKINFO),szText);//Вывод в диалог
  }
```

Результат работы программы приведен на рис. 6.17.

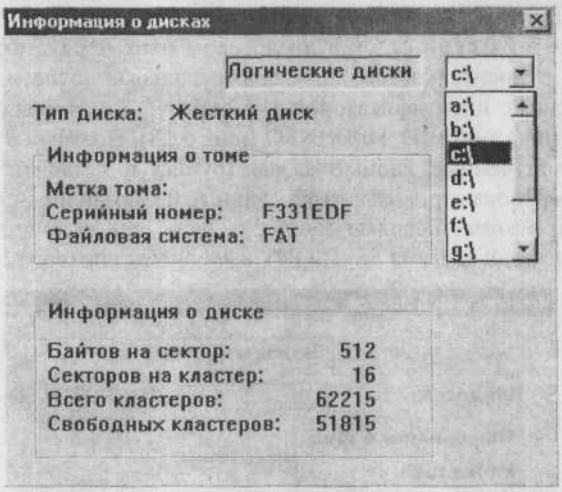


Рис. 6.17. Диалоговое окно с комбинированным списком в качестве главного окна приложения

В заголовочном файле 6-5.H, как всегда, собраны определения констант (в данном случае идентификаторов элементов управления диалогом) и прототипы функций, используемых в программе.

Файл ресурсов 6-5.RC содержит сценарий диалога, в котором есть некоторые любопытные особенности.

Элементы управления класса STATIC предназначены для вывода в диалоговое окно текстовых строк. Некоторые предложений, описывающих эти элементы, уже содержат выводимый текст; для таких "элементов управления" идентификаторы фактически не имеют смысла, и им присвоены значения –1. Другие предложения (последние 3 в файле ресурсов) имеют в качестве текста пустые строки, которые будут заполняться программно по мере получения информации о диске. Таким элементам управления, к которым будет выполняться программное обращение, необходимо назначить уникальные идентифи-

```
    default:
      return FALSE;
    }
  }
/*Функция обработки сообщения об инициализации диалога*/
BOOL DlgOnInitDialog(HWND hwnd,HWND,LPARAM){
  DWORD dwSize=GetLogicalDriveStrings(0,NULL);//Получим размер строки
  HANDLE hHeap=GetProcessHeap();//Указатель на кучу процесса
  LPSTR szDrives=(LPSTR)HeapAlloc(hHeap,HEAP_ZERO_MEMORY,dwSize);//Выделим память
  GetLogicalDriveStrings(dwSize,szDrives);//Получим все наличные диски
  LPSTR szNext=szDrives;//Текущий адрес в строке = начальному
  while(szNext[0]!=0){
    SendDlgItemMessage(hwnd,ID_COMBO,CB_ADDSTRING,0,(LPARAM)szNext);
    szNext=strchr(szNext,0)+1;//Текущий адрес = следующему
    }
  HeapFree(hHeap,0,szDrives);//Освободим память
  return TRUE;
  }
/*Функция обработки сообщений от элементов управления*/
void DlgOnCommand(HWND hwnd,int id,HWND,UINT CodeNotify){
  char szVolume[5];
  switch(id){
    case ID_COMBO://Сообщение от комбинированного списка
      if(CodeNotify==CBN_SELCHANGE){//Если изменился выбор в списке,
        int Index=SendDlgItemMessage(hwnd,ID_COMBO,CB_GETCURSEL,0,0);//получим
         SendDlgItemMessage(hwnd,ID_COMBO,//индекс и текст
             CB_GETLBTEXT,(WPARAM)Index,(LPARAM)szVolume);//из списка
          FillInfo(hwnd,szVolume);//и вызовем функцию заполнения окна диалога
         }//Конец if
       break;
    case IDCANCEL://Сообщение от кнопки закрытия
      EndDialog(hwnd,0);//Закрыть диалог и все приложение
    }//Конец switch
  }
/*Функция-подпрограмма заполнения диалогового окна*/
void FillInfo(HWND hwnd,LPSTR szVol){
  char szText[80];//Для формирования
/*Получим тип диска*/
  int nDriveType=GetDriveType(szVol);
  switch(nDriveType){
    case DRIVE_REMOVABLE:
      strcpy(szText,"Съемный диск");
      break;
    case DRIVE_FIXED:
      strcpy(szText,"Жесткий диск");
      break;
    case DRIVE_CDROM:
      strcpy(szText,"Компакт-диск");
      break;
    }
  SetWindowText(GetDlgItem(hwnd,ID_DRIVETYPE),szText);
/*Получим информацию о логических характеристиках диска*/
  char szLabel[20];//Поле для метки тома
  char szFat[10];//Поле для типа файловой системы
  char szTemp[20];//Поле для преобразования чисел в символы
  DWORD dwVolSer,dwMaxLen,dwFlags;//Для GetVolumeInformation()
  GetVolumeInformation(szVol,szLabel,sizeof(szLabel),&dwVolSer,
                       &dwMaxLen,&dwFlags,szFat,sizeof(szFat));
  strcpy(szText,szLabel);//Копирование метки в строку вывода szText
  strcat(szText,"\n");//Добавление перевода строки
  wsprintf(szTemp,"%X",dwVolSer);//Преобразование номера в символы
```

В нормальном состоянии список свернут и видно только индикаторное окно; щелчок мыши по этому окну развертывает список и дает возможность выбрать требуемую строку.

Рассмотрим программу, которая выводит в диалоговое окно информацию о выбранном логическом диске; для выбора диска из перечня возможных воспользуемся комбинированным списком.

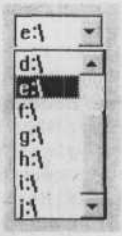


Рис. 6.16. Комбинированный список в развернутом виде

```
/*Программа 6-5. Модальный диалог с комбинированным списком*/
/*Файл 6-5.h*/
/*Определения идентификаторов элементов управления*/
#define ID_COMBO 100
#define ID_DRIVETYPE 101
#define ID_DISKINFO 102
#define ID_VOLINFO 103
/*Прототипы используемых функций*/
BOOL CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM);
void DlgOnCommand(HWND,int,HWND,UINT);
BOOL DlgOnInitDialog(HWND,HWND,LPARAM);
void FillInfo(HWND,LPSTR);
```

```
/*Файл 6-5.rc*/
#include "6-5.h"
Files DIALOG 10, 20, 179, 148
STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU | DS_MODALFRAME
CAPTION "Информация о дисках" {
  CONTROL "Логические диски",  -1, "STATIC", WS_BORDER, 72, 6, 65, 12
  CONTROL "", ID_COMBO, "COMBOBOX", CBS_DROPDOWNLIST | WS_VSCROLL,
                                                    144, 6, 30, 75
  CONTROL "Тип диска:", -1, "STATIC", SS_LEFT, 9, 21, 39, 8
  CONTROL "Информация о томе", -1,"BUTTON", BS_GROUPBOX, 9, 35, 131, 41
  CONTROL "Метка тома:\nСерийный номер:\nФайловая система:", -1, "STATIC",
                                                    SS_LEFT, 13, 46, 71, 24
  CONTROL "Информация о диске", -1, "BUTTON", BS_GROUPBOX, 9, 87, 131, 53
  CONTROL "Байтов на сектор:\nСекторов на кластер:\n"
          "Всего кластеров:\nСвободных кластеров:",
           -1,"STATIC", SS_LEFT, 13, 100, 77, 32
  CONTROL "", ID_VOLINFO, "STATIC", SS_LEFT, 84, 46, 52, 24
  CONTROL "", ID_DRIVETYPE, "STATIC", SS_LEFT, 54, 21, 60, 8
  CONTROL "", ID_DISKINFO, "STATIC",SS_RIGHT, 93, 100, 28, 32
  }
```

```
/*Файл 6-5.cpp */
#include <windows.h>
#include <windowsx.h>
#include "6-5.h"
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  DialogBox(hInst,"Files",NULL,DlgProc);
  return 0;
  }
/*Оконная функция диалогового окна*/
BOOL CALLBACK DlgProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_INITDIALOG,DlgOnInitDialog);
    HANDLE_MSG(hwnd,WM_COMMAND,DlgOnCommand);
```

Ключевое слово WINUSERAPI, с которого начинается прототип, фактически ничего не значит, так как это символическое обозначение просто объявлено известным. LONG говорит о типе возвращаемого функцией значения. Ключевое слово WINAPI определено в файле WINDEF.H как эквивалент __stdcall, и следовательно, для этой функции действует соглашение стандартного вызова (см. гл. 4).

Функция SendDlgItemMessage() в качестве своего первого параметра требует указания дескриптора того диалогового окна, в котором находится искомый элемент управления. Откуда его взять? Если бы мы создавали окно (главное или внутреннее) с помощью функции CreateWindow(), мы получили бы дескриптор создаваемого окна в качестве возвращаемого этой функцией значения. Однако окно модального диалога создается функцией DialogBox(), которая возвращает не дескриптор, а код завершения диалога. Таким образом, на этапе создания диалогового окна мы не имеем его дескриптора и практически не можем взаимодействовать с этим окном. Однако при активизации оконной функции диалога Windows передает в нее в качестве ее первого параметра дескриптор того окна, которому эта функция принадлежит. Макросы HANDLE_MSG передают далее этот параметр в функции обработки сообщений DlgOnInitDialog() и DlgOnCommand(), где им можно воспользоваться, в частности, при вызове функции SendDlgItemMessage().

Параметр nIDDlgItem представляет собой идентификатор конкретного элемента управления (в нашем примере это ID_LIST). Параметр Msg определяет код посылаемого сообщения, который указывается обычно в символической форме (например, LB_ADDSTRING). Наконец, последние два параметра позволяют передать элементу управления специфическую для него информацию (адрес передаваемой в список строки, номер выделяемой строки и т. д.).

Сообщение LB_ADDSTRING несет с собой лишь адрес передаваемой строки, который указывается в качестве последнего параметра функции SendDlgItemMessage(). Как видно из ее прототипа, этот параметр должен быть типа LPARAM, в то время как передаваемое данное имеет тип TCHAR[ ], т. е. фактически char* (имя символьного массива представляет собой указатель на символы). Приведение типа осуществляется указанием перед фактическим параметром обозначения требуемого типа в скобках.

В случае успешного нахождения первого файла организуется цикл while вызова функции FindNextFile() с отправкой в каждом шаге этого цикла очередного найденного имени файла в список диалога с помощью той же функции SendDlgItemMessage(). Цикл выполняется до тех пор, пока функция FindNextFile() возвращает значение TRUE, что говорит об обнаружении очередного файла. После исчерпания файлов, удовлетворяющих шаблону, завершается цикл поиска и вся функция DlgOnInitDialog().

Шаблон, указанный в программе 6-4, позволяет осуществить перебор файлов в текущем каталоге текущего диска. Если требуется искать файлы где-то в другом месте, в шаблон должен входить полный путь, например

```
"C:\\Мои архивы программ\\*.cpp"
```

Комбинированный список в диалоговом окне

Комбинированный список (combo box) представляет собой обычный список, к началу которого пристроено индикаторное окно для вывода выбранной в настоящий момент строки списка (рис. 6.16).

Перебор файлов в каждом каталоге осуществляется в два приема: сначала функцией FindFirstFile() отыскивается первый файл, удовлетворяющий заданному шаблону, а затем циклическим вызовом функции FindNextFile() ищутся все остальные.

При вызове функции FindFirstFile() в качестве ее первого параметра указывается шаблон поиска (в нашем примере шаблон "6-4.*" задает поиск файлов с именем 6-2 и любыми расширениями), а вторым параметром выступает адрес структурной переменной типа WIN32_FIND_DATA. Если функция поиска обнаруживает искомый файл, она заносит в элементы этой переменной характеристики файла и возвращает "дескриптор поиска" типа HANDLE, который затем следует использовать в качестве входного параметра при вызове функции FindNextFile(). Если файл не найден, функция возвращает значение INVALID_HANDLE_VALUE (равное –1 и определенное в файле WINBASE.H) и мы завершаем функцию DlgOnInitDialog() оператором return, ничего не поместив в список.

Если функция FindFirstFile() нашла первый файл, удовлетворяющий указанному шаблону, вызывается функция SendDlgItemMessage() (о ней ниже), посылающая диалоговому окну сообщение LB_ADDSTRING для занесения найденного имени в список. Обратите внимание на то, что имя найденного файл пересылается элементу управления диалоговым окном непосредственно из структурной переменной fd.

Передача сообщений элементам управления диалогового окна

До сих пор мы имели дело только с сообщениями, рождающимися в Windows и приходящими из Windows в приложение. В настоящей программе рассматривается другой, можно сказать, противоположный по смыслу вид сообщений – сообщения, посылаемые в окна приложения из самой программы. Такие сообщения позволяют программно управлять видом и содержимым окон. В Windows предусмотрен целый ряд функций для создания и посылки сообщений. Наиболее универсальной является функция SendMessage(); примеры ее использования будут приведены в последующих главах. Для диалоговых окон обычно используется функция SendDlgItemMessage(), с помощью которой обусловленные в системе сообщения посылаются непосредственно элементам управления диалогового окна. В данной программе эти сообщения используются для заполнения списка именами найденных файлов.

Сообщения, посылаемые программой элементам управления диалогового окна, могут быть двоякого рода: изменяющими вид или содержимое элемента управления или получающими информацию о его состоянии. Для каждого элемента управления (кнопка, список, контрольный переключатель и др.) предусмотрен свой, обычно довольно обширный, набор сообщений, имеющих закрепленные за ними символические обозначения. Так, посылка альтернативной кнопке сообщения BM_SETCHECK позволяет установить эту кнопку в нажатое или, наоборот, отжатое состояние; посылка списку сообщения LB_ADDSTRING позволяет добавить в список очередную строку его содержимого, а с помощью сообщения LB_SETCURSEL можно выделить цветом выбранную строку списка. В нашем случае требуется посылка сообщений LB_ADDSTRING.

Функция SendDlgItemMessage() имеет следующий прототип, описанный в файле WINUSER.H:

```
WINUSERAPI LONG WINAPI SendDlgItemMessage(
  HWND hDlg,//Дескриптор диалогового окна
  int nIDDlgItem,//Идентификатор элемента управления
  UINT Msg,//Код (символическое обозначение) посылаемого сообщения
  WPARAM wParam,//Первый параметр посылаемого сообщения
  LPARAM lParam//Второй параметр посылаемого сообщения
  );
```

```
  DialogBox(hInst,"Files",NULL,DlgProc);//Открыть диалог
  return 0;
  }
/*Оконная функция диалогового окна*/
BOOL CALLBACK DlgProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_INITDIALOG,DlgOnInitDialog);
    HANDLE_MSG(hwnd,WM_COMMAND,DlgOnCommand);
    default:
      return FALSE;
    }
  }
/*Функция обработки сообщения WM_INITDIALOG для диалога*/
BOOL DlgOnInitDialog(HWND hwnd,HWND,LPARAM){
  WIN32_FIND_DATA fd;//Структура для функций поиска файлов
  HANDLE hFile=FindFirstFile("6-4.*",&fd);//Найдем первый файл
  if(hFile==INVALID_HANDLE_VALUE)//Если не найден, завершить
    return TRUE;
  else
    SendDlgItemMessage(hwnd,ID_LIST,//Добавить имя файла из структуры fd
        LB_ADDSTRING,0,(LPARAM)fd.cFileName);//в список
  while(FindNextFile(hFile,&fd))//Поиск остальных файлов в цикле
    SendDlgItemMessage(hwnd,ID_LIST,//Добавлять имена найденных файлов
        LB_ADDSTRING,0,(LPARAM)fd.cFileName);//из структуры fd в список
  return TRUE;
  }
/*Функция обработки сообщений WM_COMMAND для диалога*/
void DlgOnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id){
    case IDCANCEL:
      EndDialog(hwnd,0);
    }
  }
```

В заголовочном файле 6-4.H определена константа ID_LIST, которая будет служить идентификатором нашего списка; далее указаны прототипы используемых в программе прикладных функций.

Сценарий диалога чрезвычайно прост. В нем кроме координат диалогового окна и его заголовка описан единственный элемент управления предопределенного класса LISTBOX, служащего для организации списков.

Главная функция WinMain() в таком варианте приложения состоит буквально из одного предложения – вызова функции DialogBox() создания модального диалога с именем Files. Как уже отмечалось, важнейшим параметром этого вызова является имя оконной функции диалогового окна DlgProc, которую Windows будет вызывать при поступлении в диалоговое окно каких-либо сообщений.

Оконная функция DlgProc() предусматривает обработку всего двух сообщений – WM_INITDIALOG, поступающего в диалог в процессе его создания, и WM_COMMAND от элементов управления. В нашем случае единственным элементом управления, способным инициировать сообщения, является кнопка  закрытия диалога, и именно ради этой кнопки в диалоговую процедуру введена обработка сообщения WM_COMMAND.

Вся содержательная часть нашей программы сосредоточена в функции DlgOnInitDialog(), вызываемой перед выводом диалога на экран. Функция начинается с объявления структурной переменной fd типа WIN32_FIND_DATA, используемой функциями Win32 поиска файлов.

Тогда фактический аргумент функции EndDialog() вернулся бы в переменную result. Таким образом создается возможность обрабатывать коды завершения диалогового окна.

Модальный диалог как главное окно приложения

Использование разнообразных элементов управления позволяет формировать весьма сложные по структуре диалоговые окна, содержащие как средства ввода в программу требуемой числовой или символьной информации, так и средства вывода результатов ее работы. Многие прикладные задачи, в частности задачи моделирования процессов, математических вычислений, управления технологическими установками или научными экспериментами и др., вполне удовлетворительно решаются с помощью программ, построенных на основе модального диалога (без главного окна). К приложениям с главным окном приходится прибегать главным образом в тех случаях, когда на экран требуется выводить рисунки (например, схемы процессов или установок) или графики. Если же результаты работы программы допустимо представлять в числовой форме, проще и удобнее использовать приложение на базе модального диалога.

Список в диалоговом окне

Рассмотрим в качестве примера программу, которая просматривает текущий каталог и выводит на экран (конкретно – в диалоговое окно) список всех файлов, удовлетворяющих некоторому шаблону (рис. 6.15).

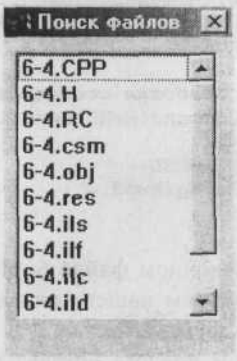


Рис. 6.15. Приложение без главного окна

```
/*Программа 6-4. Модальный диалог без главного окна*/
/*Файл 6-4.h*/
#define ID_LIST 100
BOOL CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM);
void DlgOnCommand(HWND,int,HWND,UINT);
BOOL DlgOnInitDialog(HWND,HWND,LPARAM);

//Файл 6-4.rc
#include "6-4.h"
/*Сценарий диалога*/
Files DIALOG 41, 21, 68, 97
STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU | DS_MODALFRAME
CAPTION "Поиск файлов" {
  CONTROL "",ID_LIST,"LISTBOX",WS_BORDER|WS_VSCROLL,3,4,61,88
  }

/*Файл 6-4.cpp*/
#include <windows.h>
#include <windowsx.h>
#include "6-4.h"
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
```

если они предусмотрены. В нашей программе их нет, однако перехват сообщения WM_INITDIALOG все же имеет смысл. Дело в том, что возврат в Windows в ответ на это сообщение значения TRUE приводит к специфическим действиям – установке так называемого фокуса ввода на определенный элемент управления диалогом. Обычно это элемент, описанный первым в файле ресурсов. Если элемент управления владеет фокусом ввода, то в него поступают все сообщения от клавиатуры. Если фокус ввода установлен, например, на поле для ввода текста, то вводимый с клавиатуры текст будет попадать именно в это поле; если фокус ввода установлен на кнопке, нажатие клавиши Enter приведет к "нажатию" этой кнопки. В нашем диалоге предусмотрена единственная кнопка "Закрыть", и помещение на нее фокуса ввода позволяет управлять этой кнопкой не только мышью, но и клавишей Enter.

Второе обрабатываемое в программе сообщение, WM_COMMAND, поступает в диалоговое окно в следующих случаях:

- нажата кнопка "Закрыть"; передаваемый в функцию обработки сообщения WM_COMMAND идентификатор элемента управления id равен в нашем случае ID_OK (id=100), поскольку именно эта константа закреплена за кнопкой в файле ресурсов;
- введено сочетание Alt+F4 (id=IDCANCEL=2);
- щелчок по кнопке с крестиком в правом верхнем углу диалогового окна (id=IDCANCEL).

В нашем простом диалоге любое из этих действий должно приводить к закрытию диалогового окна, что и выполняется в функции DlgOnCommand() путем вызова функции EndDialog().

Функция EndDialog() вызывает два действия: гасится диалоговое окно и завершается выполнение функции DialogBox(). Вспомним, что для вывода на экран модального диалога надо вызывать функцию DialogBox(). Вспомним также, что, пока на экране присутствует окно модального диалога, все остальные окна приложения блокированы: пользователь может работать только с элементами управления модального диалога. Именно этим модальный диалог и отличается от немодального, который не препятствует манипуляциям с любыми другими окнами и элементами управления приложения.

Все время, пока пользователь работает с модальным диалогом, выполняется функция DialogBox(). Другими словами, выполнение программы как бы остановилось в точке вызова DialogBox() в функции OnCommand(), которая является фрагментом оконной функции главного окна. Эта остановка не мешает Windows вызывать по мере необходимости оконную функцию диалога и вложенные в нее подпрограммы; однако пока программа не завершила обработку сообщения WM_COMMAND главного окна, она не может перейти к обработке следующих сообщений, поступающих в главное окно.

Вызов функции EndDialog() завершает выполнение функций DlgOnCommand() и OnCommand(), снова активизируя главное окно приложения.

Функция EndDialog() требует указания двух аргументов – типов HWND и int соответственно. Первый – это, естественно, дескриптор диалогового окна, передаваемый из Windows в оконную функцию диалога DlgProc() при ее вызове, а из нее в функцию DlgOnCommand(). Второй аргумент представляет собой целое число, которое будет служить возвращаемым значением для функции создания диалогового окна DialogBox(). Для того чтобы использовать это значение, открывать диалог следовало так:

```
int result=DialogBox(...);
```

вызове, и нет необходимости запоминать его в глобальной переменной (хотя иногда, если окон много, это все же приходится делать).

Четвертый параметр принципиально является важнейшим; он представляет собой имя оконной функции для окна диалога. Как уже отмечалось, каждое окно, входящее в состав приложения, обязано иметь свою оконную функцию. Оконная функция диалога вызывается системой Windows в процессе инициализации диалога, а также при выполнении пользователем каких-либо операций с элементами управления диалогового окна. Параметры этой функции такие же, как у оконной функции главного окна, однако возвращаемое значение должно иметь тип не LRESULT, а BOOL (между прочим, обязательно тип BOOL Windows, а не тип bool C++!). Роль функции создания диалога заключается прежде всего в том, что она привязывает к окну диалога соответствующую ему оконную функцию, состоящую из фрагментов обслуживания элементов управления диалогового окна.

Назначение оконной функции диалогового окна, как и любой другой оконной функции, состоит в обработке сообщений, поступающих в это окно. Сообщения могут поступать из системы Windows по инициативе системы (например, сообщение о создании диалога), а могут быть следствием манипуляций элементами управления в диалоговом окне, например нажатия той или иной кнопки.

Для большей части сообщений, поступающих в диалоговое окно, в файле WINDOWSX.H можно найти макросы вида HANDLE_WM_*сообщение*, поэтому структура оконной функции диалога может быть в точности такой же, как и для оконной функции главного окна:

```
BOOL CALLBACK DlgProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_INITDIALOG,DlgOnInitDialog);
    HANDLE_MSG(hwnd,WM_COMMAND,DlgOnCommand);
    default:
      return FALSE;//Возвращаем FALSE для сообщений, которые не обрабатываем
    }
  }
```

Рассмотрим отличия диалоговой функции. Для диалогов нет необходимости, более того, недопустимо вызывать функцию обработки сообщений по умолчанию DefWindowProc(). Эта функция вызывается только для окон (главного и порожденных), которые строит и обслуживает пользователь. Диалог же организует и поддерживает Windows, которая и так знает, что ей надлежит делать с большинством сообщений, поступающих в диалоговое окно.

Надо заметить, что в процессе создания диалога и работы с ним Windows посылает в него большое количество сообщений, на которые сама же и отвечает. Программа же обрабатывает лишь немногие из них. Прежде всего это сообщение WM_INITDIALOG о создании диалога и WM_COMMAND о работе пользователя с элементами управления.

Windows должна знать, какие сообщения мы обрабатываем, а какие нет. Обычно руководствуются следующим правилом: если оконная функция диалога обрабатывает сообщение, она возвращает значение TRUE, а если не обрабатывает – FALSE. Поэтому для всех сообщений, не обрабатываемых в приложении, надо просто завершить оконную функцию, вернув значение FALSE (см. предложение default в приведенном выше фрагменте).

В рассматриваемом примере обрабатываются лишь два сообщения модального диалога. Сообщение WM_INITDIALOG поступает в диалоговое окно непосредственно перед его выводом на экран и предназначено для выполнения инициализирующих действий,

Табл. 6.2. Некоторые стили элемента управления BUTTON

Стиль	*Описание*
BS_PUSHBUTTON	Нажимаемая кнопка
BS_CENTER	Позиционирование текста на кнопке по центру
BS_AUTOCHECKBOX	Кнопка-флажок
BS_GROUPBOX	Групповая рамка
BS_AUTORADIOBUTTON	Селективная кнопка

Табл. 6.3. Некоторые стили элемента управления LISTBOX

Стиль	*Описание*
LBS_NOTIFY	Щелчок мышью по строке списка посылает в родительское окно сообщение WM_COMMAND
LBS_SORT	Строки списка автоматически упорядочиваются по алфавиту
WS_VSCROLL	Наличие полосы вертикальной прокрутки
WS_BORDER	Наличие рамки вокруг списка
LBS_STANDARD	Комбинация стилей LBS_NOTIFY, LBS_SORT, WS_VSCROLL и WS_BORDER

Табл. 6.4. Некоторые стили элемента управления COMBOBOX

Стиль	*Описание*
CBS_SORT	Строки списка автоматически упорядочиваются по алфавиту
CBS_DROPDOWN	Выпадающий комбинированный список

Обслуживание модального диалога

Открытие модального диалога осуществляется при выборе команды меню с идентификатором MI_ABOUT специально предусмотренной для этого функцией DialogBox() со следующим прототипом:

```
int DialogBox(
  HANDLE  hInstance,//Дескриптор экземпляра приложения
  LPCTSTR  lpTemplate,//Имя диалога в файле ресурсов
  HWND  hWndParent,//Дескриптор родительского окна
  DLGPROC  lpDialogFunc //Адрес оконной функции диалогового окна
  );
```

Эта функция использует 4 параметра. Первый из них – это дескриптор данного экземпляра приложения. Этот дескриптор не передается автоматически в оконную функцию и далее в функции обработки сообщений. Имея в виду, что он понадобится нам для вызова функции DialogBox(), мы сохранили его в глобальной переменной hInstance еще в процессе регистрации главного окна приложения.

Вторым параметром функции DialogBox() служит указатель на имя диалога, использованное в сценарии этого диалога в файле ресурсов (или само это имя). У нас это строка "About".

Третий параметр – дескриптор окна, в котором открывается диалог. Этот дескриптор нужен многим функциям Windows, поэтому он передается в оконную функцию при ее

```
CONTROL "Групповая рамка", -1, "button", BS_GROUPBOX,
            122, 4, 108, 40
CONTROL "Селективная кнопка 1", ID_RADIO1, "button",
            BS_AUTORADIOBUTTON, 125, 16, 84, 12
CONTROL "Селективная кнопка 2", ID_RADIO2, "button",
            BS_AUTORADIOBUTTON, 125, 28, 84, 12
}
```

Внешний вид диалогового окна, соответствующего такому файлу ресурсов, изображен на рис. 6.14.

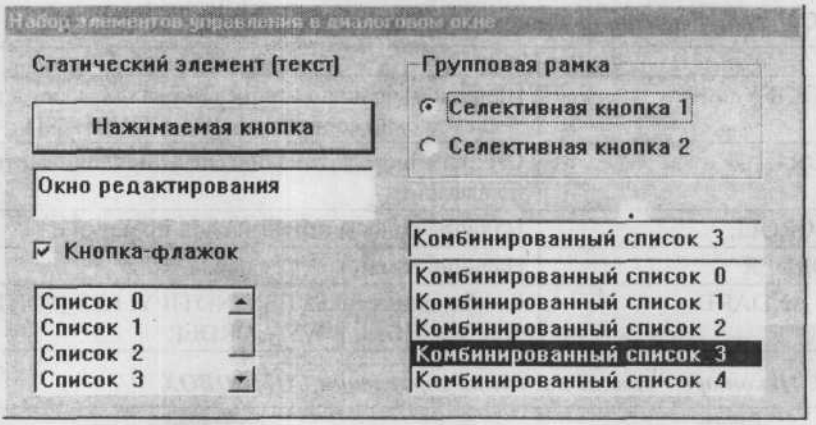


Рис. 6.14. Диалоговое окно с набором элементов управления

В приведенном варианте файла ресурсов для некоторых элементов управления указаны поясняющие надписи ("Статический элемент (текст)", "Окно редактирования", "Групповая рамка" и др.). У списка и комбинированного списка таких надписей-заголовков не предусмотрено, поэтому в файле ресурсов при их описании на соответствующих местах указаны пустые символьные строки (""). Содержимое же окон списка и комбинированного списка было помещено в эти элементы управления программно, при обработке сообщения WM_INITDIALOG посредством вызовов (в цикле) функции SendDlgItemMessage(). Процедуры динамического изменения содержимого списков и других элементов управления будут рассмотрены в последующих подразделах.

В табл. 6.1 – 6.4 приведены наиболее употребительные классы и стили различных элементов управления.

Табл. 6.1. Некоторые стили элемента управления STATIC

Стиль	*Описание*
SS_LEFT, SS_RIGHT, SS_CENTER	Позиционирование текста в прямоугольнике с указанными координатами, соответственно, влево, вправо или по центру
SS_WHITEFRAME	Пустая белая рамка
SS_BLACKFRAME	Пустая черная рамка
SS_ICON	Вывод пиктограммы, описанной в файле ресурсов
SS_BITMAP	Вывод рисунка, описанного в файле ресурсов

помещаемые в список текстовые строки; стиль WS_VSCROLL снабжает список вертикальной полосой прокрутки; стиль LBS_NOTIFY позволяет управлять списком посредством щелчков мыши. В рамках класса BUTTON можно создавать разнообразные кнопки – нажимаемые (стиль BS_PUSHBUTTON), селективные (стиль AUTORADIOBUTON), кнопки-флажки (стиль BS_AUTOCHECKBOX) и др.

Следует заметить, что ключевые слова CONTROL, "STATIC", "BUTTON" и др. могут писаться как прописными, так и строчными буквами, однако стили элементов управления (SS_CENTER и BS_PUSHBUTTON в нашем примере), являясь предопределенными константами, могут указываться только прописными буквами (как и все прочие константы Windows, например, HWND_DESKTOP или IDC_ARROW).

Последний блок параметров предложения CONTROL представляет собой четыре числа, задающие координаты данного элемента в окне диалога, а также его размеры (в диалоговых единицах).

Общая процедура заполнения диалогового окна элементами управления заключается в том, что в файле ресурсов описываются виды элементов управления и их местоположение в диалоге, а затем программным образом организуется взаимодействие их с программой путем посылки и приема сообщений. С помощью сообщений, посылаемых элементам управления диалога, можно, например, нажать или отпустить кнопку, выделить цветом строку списка, пометить или снять отметку с кнопки-флажка; с помощью же сообщений, принимаемых от элемента управления, можно определить, какая кнопка нажата или какая строка списка выбрана пользователем.

Стоит отметить, что для большинства элементов управления допускаются два формата описания, равнозначные по конечному результату. Помимо формата с обобщенным ключевым словом CONTROL, описанного выше, в котором вид элемента управления задается его классом и стилем, имеется возможность определять элементы управления с помощью набора ключевых слов, сразу описывающих и класс, и стиль. В этом случае нажимаемая кнопка описывается ключевым словом PUSHBUTTON, центрированный текст – ключевым словом CTEXT и т. д.

Так, для нажимаемой кнопки, включенной в файл 6-1.RC, возможны следующие равнозначные описания:

```
CONTROL "Закрыть",ID_OK,"BUTTON",BS_PUSHBUTTON,27,32,43,12
PUSHBUTTON "Закрыть",ID_OK, 27, 32, 43, 12
```

Ниже для справки приводится пример файла ресурсов с описаниями (в первом формате) наиболее употребительных элементов управления с указанием типичных для них стилей:

```
Controls DIALOG 31, 6, 241, 116
CAPTION "Набор элементов управления в диалоговом окне" {
  CONTROL "Нажимаемая кнопка", ID_BTN, "button",
            BS_PUSHBUTTON | BS_CENTER, 8, 20, 104, 16
  CONTROL "Кнопка-флажок", ID_CHECK, "button",
            BS_AUTOCHECKBOX, 8, 60, 64, 12
  CONTROL "Статический элемент (текст)", ID_TEXT, "static",
            SS_LEFT, 8, 4, 104, 12
  CONTROL "Окно редактирования", ID_EDIT, "edit",
            ES_LEFT | WS_BORDER, 8, 40, 104, 16
  CONTROL "",ID_LIST,"listbox",LBS_STANDARD,8,76,68,36
  CONTROL "",ID_COMBO,"combobox",CBS_SORT,122,56,108,56
```

Следующее предложение сценария диалога определяет его стиль, для чего служит ключевое слово STYLE. Стиль диалога, как и вообще любого окна, задается с помощью символических констант, описывающих отдельные характеристики окна и объединяемых в одно слово с помощью побитовой операции ИЛИ (знак |). Для модального диалога обычно используется специфический стиль DS_MODALFRAME, определяющий наличие вокруг диалогового окна толстой рамки, позволяющей перемещать окно по экрану, но не изменять его размеры, а также общие для всех окон стили WS_CAPTION и WS_SYSMENU для создания заголовка и системного меню. Стиль WS_POPUP говорит о том, что диалоговое окно будет "всплывающим". Все перечисленные выше стили назначаются диалогу по умолчанию, и предложение STYLE в нашем простом случае можно было опустить.

Далее в операторных скобках {...} определяется перечень элементов управления, включаемых в диалоговое окно. Этот перечень задается с помощью зарезервированных ключевых слов и позволяет разместить в диалоговом окне любые предусмотренные системой элементы – кнопки, списки, надписи и пр. В рассматриваемой программе используются лишь два элемента управления – область с текстом и нажимаемая кнопка.

Элементы управления описываются в файле ресурсов с помощью ключевого слова CONTROL, имеющего следующий формат:

```
CONTROL "текст", идентификатор, "класс", стиль, координаты_и_размеры
```

Параметр "*текст*" определяет тот текст, который будет выведен вместе с данным элементом управления. Для кнопок – это надпись на кнопке или рядом с ней; для области с текстом – строки, выводимые в указанную область, для групповой рамки – ее заголовок. Некоторые элементы управления не могут иметь текстового сопровождения и для них этот параметр игнорируется.

Идентификатор элемента управления, как и идентификатор команды меню, представляет собой произвольное число, обычно указываемое в виде символической константы; он позволяет программе определить, от какого элемента управления пришло сообщение или, наоборот, какому элементу сообщение отправлено. В нашей программе в качестве идентификатора кнопки используется константа ID_OK, которой в файле 6-1.H присвоено значение 100. Область с фиксированным текстом ("Автор программы...") не посылает сообщений, и ее идентификатор не играет роли. Обычно таким идентификаторам дается значение –1.

В качестве третьего параметра предложения CONTROL указывается (в кавычках или без них) класс элемента управления. В Windows имеется целый ряд предопределенных классов специфических окон, используемых в качестве элементов управления как в диалогах, так и в обычных окнах. Названия этих классов и указываются в предложениях CONTROL в файле ресурсов. Так, класс STATIC позволяет, в частности, создавать области с текстом; класс LISTBOX – списки; класс EDIT – окна редактирования, класс BUTTON – разнообразные кнопки. В Windows имеются и другие предопределенные классы.

В рамках указанного класса можно создавать элементы управления, различающиеся (иногда очень сильно) по внешнему виду и характеристикам. Конкретный вид элемента управления определяется его стилем. Так, для класса STATIC стиль SS_CENTER задает область с текстом, выровненным по центру; стиль SS_LEFT выравнивает указанный текст по левому краю; стиль SS_ICON (того же класса STATIC) не имеет отношения к текстам, а служит для получения изображения пиктограммы; стиль SS_BITMAP позволяет вывести в диалоговое окно растровое изображение. Для класса LISTBOX стиль LBS_SORT упорядочивает

```
  return 0;
  }
/*Оконная функция главного окна*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);//При выборе пунктов меню
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);//При завершении приложения
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
/*Функция обработки сообщений WM_COMMAND от пунктов меню главного окна*/
void OnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id){//id=идентификатор выбранного пункта меню
    case MI_ABOUT:
      DialogBox(hInstance,"About",hwnd,DlgProc);//Открыть диалог
      break;//Выход из оператора switch
    case MI_EXIT:
      DestroyWindow(hwnd);//Завершить приложение
    }//Конец оператора switch
  }//Конец функции OnCommand
/*Функция обработки сообщения WM_DESTROY*/
void OnDestroy(HWND){
  PostQuitMessage(0);
  }
/*Оконная функция диалогового окна*/
BOOL CALLBACK DlgProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_INITDIALOG,DlgOnInitDialog);//При инициализации диалога
    HANDLE_MSG(hwnd,WM_COMMAND,DlgOnCommand);//При нажатии кнопок диалога
    default://При любых других сообщениях, связанных с диалогом
      return FALSE;
      }//Конец оператора switch
   }//Конец функции DlgProc
/*Функция обработки сообщения WM_INITDIALOG от диалога*/
BOOL DlgOnInitDialog(HWND,HWND,LPARAM){
  return TRUE;//Устанавливает фокус клавиатуры на кнопку
  }
/*Функция обработки сообщений WM_COMMAND от диалога */
void DlgOnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id){//Код элемента управления (кнопки)
    case ID_OK://Нажата кнопка "Закрыть"
    case IDCANCEL://Дана команда закрыть диалог через его системное меню
      EndDialog(hwnd,0);//Закрытие диалогового окна
    }
  }
```

Описание диалога в файле ресурсов

Так же как и меню, модальный диалог описывается в файле ресурсов с помощью соответствующих ключевых слов. Начинается это описание с ключевого слова DIALOG, предваряемого произвольным именем этого диалога. Естественно, в файле ресурсов можно описать любое количество диалоговых окон с различающимися именами. В качестве параметров этого предложения языка ресурсов выступают 4 целых числа – координаты начала (т. е. верхнего левого угла) диалогового окна и его размеры по осям x и y. Эти величины задаются не в пикселах, а в диалоговых единицах, значение которых в пикселах можно получить с помощью функции GetDialogBaseUnits(). Обычно каждая диалоговая единица (и по оси x, и по оси y) равна 2 пикселам.

```
#define ID_OK 100
/*Прототипы функций для главного окна*/
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
void OnCommand(HWND,int,HWND,UINT);
void OnDestroy(HWND);
/*Прототипы функций для окна диалога*/
BOOL CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM);
BOOL DlgOnInitDialog(HWND,HWND,LPARAM);
void DlgOnCommand(HWND,int,HWND,UINT);
```

```
/*Файл 6-3.rc*/
#include "6-3.h"
/*Сценарий меню*/
Main MENU{
  POPUP "&Файл"  {
    MENUITEM "&О программе...",MI_ABOUT
    MENUITEM SEPARATOR
    MENUITEM "В&ыход",MI_EXIT
    }
  }
/*Сценарий диалога*/
About DIALOG 60, 15, 93, 50
STYLE WS_SYSMENU | WS_CAPTION | WS_POPUP | DS_MODALFRAME
CAPTION "О программе"{
  CONTROL "Автор программы\nИванов И.И.\n2002", -1, "STATIC", SS_CENTER,
                                                          12, 4, 72, 25
  CONTROL "Закрыть", ID_OK, "BUTTON", BS_PUSHBUTTON, 27, 32, 43, 12
  }
```

```
/*Файл 6-3.cpp */
/*Операторы препроцессора*/
#include <windows.h>
#include <windowsx.h>
#include "6-3.h"
/*Глобальная переменная*/
HINSTANCE hInstance;
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  char szClassName[]="MainWindow";
  char szTitle[]="Программа 6-3";
  MSG Msg;
  WNDCLASS wc;
  hInstance=hInst;//Сохраним дескриптор приложения
/*Зарегистрируем класс главного окна*/
  memset(&wc,0,sizeof(wc));
    wc.lpfnWndProc=WndProc;//Оконная процедура для главного окна
    wc.hInstance=hInst;
    wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
    wc.hCursor=LoadCursor(NULL,IDC_ARROW);
    wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
    wc.lpszMenuName="Main";//Имя меню в файле ресурсов
    wc.lpszClassName=szClassName;
    RegisterClass(&wc);
/*Создадим главное окно и сделаем его видимым*/
  HWND hwnd=CreateWindow(szClassName,szTitle,WS_OVERLAPPEDWINDOW,
                         10,10,350,200,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_SHOWNORMAL);
  while(GetMessage(&Msg,NULL,0,0)){//Цикл обнаружения сообщений
    TranslateMessage(&Msg);//Теперь меню управляется с помощью Alt/буква,
    DispatchMessage(&Msg);//а пункты меню - буквами без Alt
    }
```

ми и их можно создать в программе с помощью соответствующих функций Windows. Однако при объявлении некоторого окна диалоговым мы передаем Windows значительную часть работы по созданию и поддержке этого окна, упрощая тем самым программу.

Диалог, представляя собой окно со специальными свойствами, может существовать и без главного окна, образуя специфический тип "безоконного" приложения. Такие приложения, состоящие из одного диалогового окна, оказываются значительно проще обычных программ с главным окном и широко используются как в качестве законченных программных продуктов, так и для проведения разного рода программных экспериментов, исследований и демонстраций.

Диалоги бывают двух видов: модальные и немодальные (довольно странная терминология, ни в русском, ни в английском языке не имеющая ничего общего со специфическими свойствами этих окон). Модальный диалог характерен тем, что после открытия диалогового окна пользователь может работать только с *его* элементами управления, а все остальные окна приложения блокируются до тех пор, пока модальный диалог не будет закрыт. Модальные диалоги естественно использовать в тех случаях, когда без информации, вводимой через окно диалога, программа не может дальше работать. Если, например, программа на некотором этапе должна ввести данные из того или иного файла, то выбор конкретного файла естественно выполнить с помощью модального диалогового окна. Кстати, отсюда следует, что нормальное состояние модального диалога – закрытое.

Немодальный диалог не препятствует пользователю работать с любыми элементами главного окна приложения или его внутренних окон. С помощью немодальных диалогов удобно, например, задавать масштаб, цвет или какие-то другие характеристики выводимых на экран диаграмм и графиков. Нормальное состояние немодального диалога – открытое.

Простая программа с меню и диалогом

Рассмотрим сначала типичную организацию приложения с главным окном, управляющим меню и диалогом в качестве внутреннего элемента.

В приложении 6-3 предусмотрена линейка меню с единственным пунктом "Файл", состоящим из двух команд: "О программе..." и "Выход" (рис. 6.13). Выбор команды "О программе..." приводит к выводу на экран диалогового окна с информацией об авторе программы и кнопкой "Закрыть". Кнопка диалога "Закрыть" закрывает диалоговое окно, а команда меню "Выход" завершает все приложение.

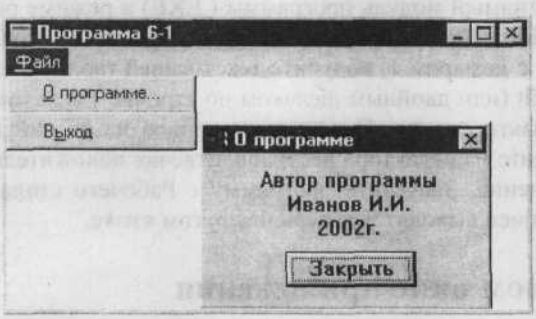


Рис. 6.13. Приложение с меню и диалоговым окном

```
/*Программа 6-3. Меню и диалог*/
/*Файл 6-3.h*/
/*Определения констант*/
#define MI_ABOUT 100
#define MI_EXIT 101
```

Ресурс - таблица строк описывается следующим образом:

```
STRINGTABLE {
  0,"первая строка"
  1,"вторая строка"
  2,"третья строка"
  ... // и т. д.
  }
```

Для использования строк таблицы в программе каждую строку следует сначала загрузить с помощью функции LoadString(), первым параметром которой служит дескриптор приложения, вторым – номер, которым помечена строка в файле ресурсов, третьим – адрес символьного массива, в который эта строка будет скопирована, а четвертым – размер этого массива. Отсюда, между прочим, следует, что при наличии в файле ресурсов нескольких таблиц строк (это вполне возможно), строки во всех таблицах должны иметь различающиеся номера, при этом порядок номеров не имеет значения. После загрузки строки ее копия в символьном массиве – приемнике может использоваться во всех местах, где допустимы символьные строки: при выводе текстов в окна приложения, в качестве заголовков окон или пунктов меню и т. д.

Для того, чтобы познакомиться с методикой локализации программных продуктов, воспользуйтесь тем вариантом программы 6-1, в которой меню создавалось программным образом. Включите в приложение файл ресурсов и оформите все выводимые в окно текстовые строки (заголовок главного окна, а также заголовки пунктов меню) в качестве ресурса – таблицы строк, как это описано в подразделе выше.

Для извлечения этих строк из секции ресурсов выполнимого модуля в программу воспользуйтесь функцией LoadString(), в качестве второго параметра которой выступают номера, присвоенные строкам в файле ресурсов. Откомпилируйте программу и убедитесь, что она работает точно так же, как и раньше.

Закройте проект и перенесите в другой каталог файлы ресурсов .RC и .RES, а также исходный файл .CPP, лишив себя, таким образом, исходных текстов программы. Выведите на рабочий стол Windows значок для быстрого запуска вашей программы и убедитесь в его правильном функционировании.

Выполните локализацию программы для англоязычной страны, т. е. перевод надписей на английский язык *при отсутствии исходного текста программы*. Откройте в среде Borland C++ выполнимый модуль программы (.EXE) в режиме редактирования ресурсов Edit Resources. Щелкнув по крестику слева от пункта STRINGTABLE, а затем по появившейся строчке с номером 1, получите текст вашей таблицы строк. Выбором пункта меню Resource➢Edit (или двойным щелчком по строчке 1 в левом окне редактора) перейдите в режим редактирования. Переведите надписи на английский или другой иностранный язык. Выйдите из редактора ресурсов, отвечая положительно на его вопросы о компиляции и сохранении. Запустите программу с Рабочего стола. Убедитесь, что все измененные вами надписи выводятся теперь на другом языке.

Диалог в главном окне приложения

Диалоговые окна, или просто диалоги, – это специальный вид окон, позволяющих вести диалог с программой, т. е. вводить в программу и получать из нее разнообразную информацию. Обмен информацией с программой и задание режимов ее работы осуществляется с помощью элементов, или органов, управления – кнопок разного вида и с разными свойствами, контрольных переключателей, списков, окон ввода, линеек прокрутки и др. В принципе все эти элементы, так же как и сама диалоговая панель, являются окна-

Функция OnNotify() будет вызываться в ответ на все сообщения WM_NOTIFY, однако нас будут интересовать лишь те сообщения, для которых код code, входящий в структуру NMHDR, равен TTN_NEEDTEXT. Эти сообщения посылаются в приложение, когда курсор мыши устанавливается над какой-либо кнопкой инструментальной панели, и, соответственно, панель нуждается в получении текста подсказки. Поэтому в функции OnNotify() следует прежде всего отобрать (с помощью предложения if) эти сообщения. Вывод подсказок осуществляется путем копирования текста подсказки в элемент szText структуры TOOLTIPTEXT. Поскольку для разных кнопок подсказки разные, предложения копирования (посредством функции C++ strcpy()) удобно включить в конструкцию switch-case; в которой в этом случае должно анализировать значение элемента idFrom структуры NMHDR, в который поступает идентификатор кнопки, находящейся под курсором мыши.

Усовершенствуем программу 6-2, создав для инструментальной панели набор всплывающих подсказок. Изменения коснуться лишь трех мест в программе.

При создании инструментальной панели в стиль окна следует включить константу TBSTYLE_TOOLTIPS, указывающую на наличие инструментальной панели со всплывающими подсказками:

```
CreateToolbarEx(hwnd,
      WS_CHILD|WS_BORDER|WS_VISIBLE|TBSTYLE_TOOLTIPS,
      0,3,hI,ID_BTNS,tbb,3,16,16,16,16,sizeof(TBBUTTON));
```

В оконную процедуру добавляется макрос для обнаружения сообщений WM_NOTIFY:

```
HANDLE_MSG(hwnd,WM_NOTIFY,OnNotify);
```

Соответственно в программе должна быть функция обработки этого сообщения, которая для нашего конкретного случая имеет следующий вид:

```
LRESULT OnNotify(HWND,int,LPNMHDR lParam){
  if(lParam->code==TTN_NEEDTEXT){//Курсор над инструментальной панелью
    LPTOOLTIPTEXT lpttt = (LPTOOLTIPTEXT) lParam;//Приведение типа параметра
    UINT idBtn=lpttt->hdr.idFrom;//Идентификатор какой кнопки под курсором?
    switch(idBtn){//Переключатель по идентификатору кнопок
      case ID_1://Первая кнопка
      strcpy(lpttt->szText,"Синий");//Копируем подсказку в поле структуры
      break;
      case ID_2://Вторая кнопка
      strcpy(lpttt->szText,"Зеленый");//Копируем подсказку в поле структуры
      break;
      case ID_3://Третья кнопка
      strcpy(lpttt->szText,"Выход");//Копируем подсказку в поле структуры
      }
    }
  return 0;
  }
```

Таблицы строк и локализация программных продуктов

Выше уже отмечалось, что в качестве ресурсов могут фигурировать самые разные объекты Windows: описания меню и диалогов, растровые изображения, пиктограммы и курсоры и пр. В частности, в виде ресурсов могут быть описаны текстовые строки. Размещение таблиц строк в файле ресурсов позволяет выполнять их редактирование непосредственно в загрузочном файле. Это дает возможность локализовать программные продукты, т. е. переводить используемые в них поясняющие надписи на национальный язык при отсутствии исходного текста программы.

открываемые автоматически, когда курсор мыши помещается на соответствующую кнопку инструментальной панели (рис. 6.11).

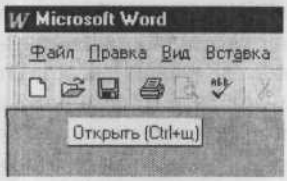


Рис. 6.11. Всплывающая подсказка в редакторе Microsoft Word

Для организации всплывающих подсказок следует прежде всего создать инструментальную панель, как это было описано в предыдущем подразделе, дополнив ее стиль, указываемый в качестве второго параметра функции CreateToolbarEx(), константой TBSTYLE_TOOLTIPS. Отслеживание положения курсора мыши и вывод на экран требуемых подсказок осуществляется в ответ на сообщение Windows WM_NOTIFY, для обработки которого оконную функцию следует дополнить соответствующим макросом HANDLE_MSG. Функция OnNotify(), вызываемая в случае прихода в приложение сообщения WM_NOTIFY, имеет следующий прототип:

```
LRESULT OnNotify (HWND hwnd, int wParam, LPNMHDR lParam);
```

Последний параметр этой функции lParam имеет тип указателя на структуру NMHDR. Фактически, однако, этот параметр указывает на бóльшую по объему структуру типа TOOLTIPTEXT, в которую структура NMHDR входит в качестве первого элемента (рис. 6.11).

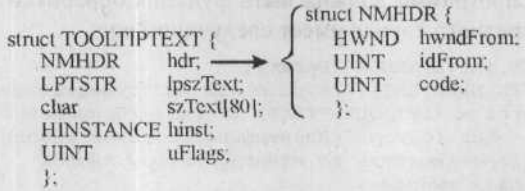


Рис. 6.12. Взаимоотношение структур TOOLTIPTEXT и NMHDR

Таким образом, если использовать параметр lParam непосредственно, он будет служить указателем на структуру NMHDR, и с его помощью можно обращаться к элементам этой структуры, однако остальные элементы структуры TOOLTIPTEXT будут недоступны. Если же преобразовать параметр lParam в тип LPTOOLTIPTEXT, то он будет указывать на всю структуру TOOLTIPTEXT, и через него можно получить доступ ко всем элементам этой структуры; через имя вложенной структуры hdr в этом случае можно получить доступ и к элементам структуры NMHDR.

Таким образом, допустимы, в частности, следующие обращения:

- lParam->code (lParam – указатель на структуру NMHDR, а code – элемент этой структуры);
- lParam->idFrom (та же ситуация);
- ((LPTOOLTIPTEXT) lParam)->szText (выражение (LPTOOLTIPTEXT) lParam является указателем на структуру TOOLTIPTEXT, а szText – элемент этой структуры);
- ((LPTOOLTIPTEXT) lParam)->hdr.idFrom (обращение к элементам вложенной структуры посредством указателя на внешнюю).

```
  tbb[1].iBitmap=1;//Порядковый номер кнопки 2
  tbb[1].idCommand=ID_2;//Идентификатор кнопки 2
  tbb[1].fsState=TBSTATE_ENABLED; //Разрешить управление от кнопки 2
  tbb[2].iBitmap=2;//Порядковый номер кнопки 3
  tbb[2].idCommand=ID_3;//Идентификатор кнопки 3
  tbb[2].fsState=TBSTATE_ENABLED; //Разрешить управление от кнопки 3
  CreateToolbarEx(hwnd,WS_CHILD|WS_BORDER|WS_VISIBLE,//Создаем инструмен-
        -1,3,hI,ID_BTNS,tbb,3,16,16,16,16,sizeof(TBBUTTON));//тальную панель
  return TRUE;
  }
void OnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id){
    case ID_1://Нажали на кнопку 1 - изменить цвет
      sw=1;
      InvalidateRect(hwnd,NULL,TRUE);
      break;
    case ID_2://Нажали на кнопку 2 - изменить цвет
      sw=2;
      InvalidateRect(hwnd,NULL,TRUE);
      break;
    case ID_3://Нажали на кнопку 3 - завершить приложение
      DestroyWindow(hwnd);
     }
  }
void OnPaint(HWND hwnd){
  PAINTSTRUCT ps;
  HDC hdc=BeginPaint(hwnd,&ps);
  if(sw==1)
    FillRect(hdc,&ps.rcPaint,hBlueBrush);
  if(sw==2){
    FillRect(hdc,&ps.rcPaint,hGreenBrush);
    }
  EndPaint(hwnd,&ps);
  }
void OnDestroy(HWND){
  PostQuitMessage(0);
  }
```

На рис. 6.10 показан вид главного окна приложения 6-2 с инструментальной панелью. Как видно из текста программы, мы предусмотрели две кнопки для управления цветом окна и третью (с символом E) – для завершения приложения.

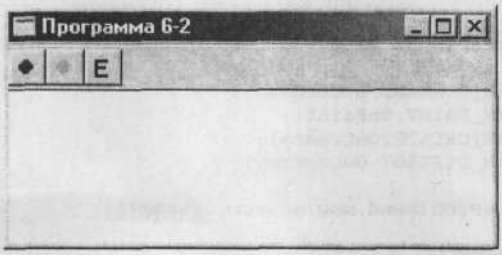


Рис. 6.10. Пример приложения с инструментальной панелью

Всплывающие подсказки

В современных программах, управляемых с помощью меню и инструментальных панелей, принято включать в состав меню всплывающие подсказки – небольшие окошки,

```
/*Файл 6-2.h*/
#define ID_1 300//Идентификатор кнопки 1
#define ID_2 301//Идентификатор кнопки 2
#define ID_3 302//Идентификатор кнопки 3
#define ID_BTNS 400//Идентификатор ресурса - файла с изображением кнопок
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
void OnCommand(HWND,int,HWND,UINT);
void OnPaint(HWND);
BOOL OnCreate(HWND,LPCREATESTRUCT);
void OnDestroy(HWND);
/*Файл 6-2.rc*/
#include "6-2.h"
ID_BTNS BITMAP "buttons.bmp"//Файл с изображением кнопок
/*Файл 6-2.cpp*/
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include "6-2.h"//Подключаем заголовочный файл
HBRUSH   hBlueBrush, hGreenBrush;//Дескрипторы кистей
int sw;//Переключатель цвета окна
HINSTANCE hI;//Дескриптор приложения
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  hI=hInst;
  char szClassName[]="MainWindow";
  char szTitle[]="Программа 6-2";
  MSG Msg;
  WNDCLASS wc;
  memset(&wc,0,sizeof(wc));
  wc.lpfnWndProc=WndProc;
  wc.hInstance=hInst;
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
  wc.lpszClassName=szClassName;
  RegisterClass(&wc);
  HWND hwnd=CreateWindow(szClassName,szTitle,
       WS_OVERLAPPEDWINDOW,10,10,300,150,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_SHOWNORMAL);
  while(GetMessage(&Msg,NULL,0,0))
    DispatchMessage(&Msg);
  return 0;
  }
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);
    HANDLE_MSG(hwnd,WM_PAINT,OnPaint);
    HANDLE_MSG(hwnd,WM_CREATE,OnCreate);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
BOOL OnCreate(HWND hwnd,LPCREATESTRUCT){
  hBlueBrush=CreateSolidBrush(RGB(100,100,255));//Создаем кисти
  hGreenBrush=CreateSolidBrush(RGB(100,255,100));//для покраски окна
  TBBUTTON tbb[3];//Массив структур TBBUTTON
  ZeroMemory(&tbb,sizeof(tbb));//Обнулим весь массив структур
  tbb[0].iBitmap=0;//Порядковый номер кнопки 1
  tbb[0].idCommand=ID_1;//Идентификатор кнопки 1
  tbb[0].fsState=TBSTATE_ENABLED;//Разрешить управление от кнопки 1
```

Здесь ID_BTNS – произвольная символическая константа, которая затем будет использоваться в качестве идентификатора данного ресурса, а BTNS.BMP – имя файла с изображением кнопок.

Действия по программной организации инструментальной панели удобно выполнить в функции OnCreate(). Здесь прежде всего создается (и обнуляется) структурная переменная – массив из трех (по числу кнопок) структур типа TBBUTTON. Эта структура описана в заголовочном файле COMMCTRL.H, который не подсоединяется автоматически, поэтому в начало программы необходимо включить директиву

```
#include <commctrl.h>
```

Структура TBBUTTON содержит шесть элементов, из которых нас будут интересовать всего три:

- iBitmap – порядковый номер кнопки (кнопки нумеруются от 0);
- idCommand – идентификатор кнопки (определенная заранее символьная константа);
- fsState – состояние кнопки. Значение TBSTATE_ENABLED разрешает управление с помощью данной кнопки.

После заполнения перечисленных выше элементов во всех трех (если кнопок три) членах нашего массива следует вызвать функцию Windows CreateToolbarEx(), указав в качестве ее параметров:

- дескриптор главного окна;
- стиль инструментальной панели (значение WS_VISIBLE);
- идентификатор всей инструментальной панели, который в программе не используется и может быть равен, например, –1;
- число изображений кнопок в файле .BMP;
- дескриптор приложения;
- идентификатор ресурса с изображением кнопок;
- адрес массива структур типа TBBUTTON;
- число кнопок;
- ширину и высоту каждой кнопки;
- ширину и высоту изображения на каждой кнопке (размеры кнопок и изображений на них в принципе могут не совпадать);
- размер структуры TBBUTTON, определяемый с помощью функции C++ sizeof().

При нажатии кнопки инструментальной панели Windows посылает в приложение сообщение WM_COMMAND, которое приводит к вызову функции OnCommand() с передачей через второй параметр этой функции значения идентификатора нажатой кнопки. Обычно в функции OnCommand() предусматривают конструкцию switch-case, с помощью которой организуют анализ значения идентификатора и выполнение запланированных действий. Если при этом в приложении имеются и обычное меню, и инструментальная панель, то при назначении пунктам меню и кнопкам инструментальной панели одинаковых идентификаторов они будут дублировать друг друга, если же идентификаторы будут разными, то с помощью меню можно управлять одними характеристиками приложения, а с помощью инструментальной панели – другими.

Модифицируем программу 6-1 так, чтобы переключение цвета окна осуществлялось выбором кнопок инструментальной панели, исключив при этом для простоты обычное меню. Программа примет такой вид:

Приведем текст функции OnRButtonDown(), которую можно включить в состав программы 6-1, чтобы получить плавающее меню. Для того чтобы из этой функции можно было обратиться к переменной Msg, ее следует объявить глобальной.

```
void OnRButtonDown(HWND hwnd,BOOL,int x,int y,UINT){
  char str1[]="Фон: Синий";
  char str2[]="Фон: Зеленый";
  char str3[]="Выход";
  RECT rect;
  HMENU hPopupMenu=CreatePopupMenu();
  AppendMenu(hPopupMenu,MF_STRING,MI_BLUE,str1);
  AppendMenu(hPopupMenu,MF_STRING,MI_GREEN,str2);
  AppendMenu(hPopupMenu,MF_SEPARATOR,0,NULL);
  AppendMenu(hPopupMenu,MF_STRING,MI_EXIT,str3);
  GetWindowRect(hwnd,&rect);
  TrackPopupMenu(hPopupMenu,TPM_LEFTALIGN|TPM_LEFTBUTTON,
        Msg.pt.x,Msg.pt.y,0,hwnd,NULL);
}
```

Инструментальная панель

Дальнейшего улучшения внешнего вида приложения можно добиться, включив в состав главного окна, наряду с линейкой меню (или вместо нее), инструментальную панель, которая представляет собой набор кнопок с рисунками, действующих точно так же, как и пункты обычного меню. Эти кнопки могут дублировать отдельные команды меню, но могут и дополнять их. На рис. 6.8 показана часть главного окна среды программирования Borland C++ со строкой заголовка, линейкой обычного меню (File, Edit и т. д.) и инструментальной панелью с пиктограммами стандартных действий (чтение и сохранение файла, компиляция, выполнение и т. д.).

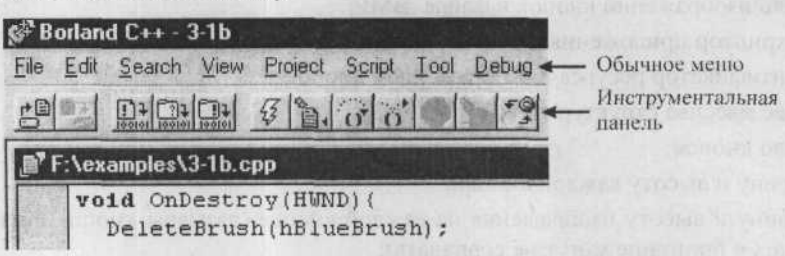


Рис. 6.8. Обычное меню и инструментальная панель

Для создания инструментальной панели необходимо прежде всего подготовить файл с растровым изображением рисунков на кнопках (в формате .BMP). Могут быть разные варианты размещения кнопок (вплотную друг к другу, с промежутками, группами и т. д.). Обычно кнопки имеют размер 16×16 пикселов, и тогда для размещения на инструментальной панели, например, трех кнопок, нужно создать растровое изображение размером 16×48 пикселов (рис. 6.9). Разумеется, рисунки на кнопках могут быть цветными.

Рис. 6.9. Изображение трех кнопок для инструментальной панели

Поскольку изображение кнопок будет выступать у нас в качестве ресурса, в файл ресурсов необходимо включить ссылку на файл с изображением кнопок, например, таким образом:

```
ID_BTNS  BITMAP  "BTNS.BMP"
```

Плавающее меню

В приложениях Windows широко используются плавающие меню (не путать с всплывающим меню, являющимся элементом обычного меню), активизируемые обычно щелчком правой клавиши мыши. Плавающее меню (рис. 6.7) появляется в том месте экрана, где в данный момент находится курсор мыши.

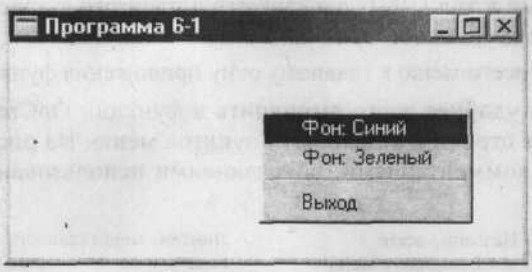


Рис. 6.7. Плавающее меню

Поскольку плавающее меню должно активизироваться нажатием правой клавиши мыши, в оконную функцию следует включить обработку сообщения WM_RBUTTONDOWN, и все действия по созданию меню выполнить в функции обработки этого сообщения (назвав ее, например, OnRButtonDown()). Установка плавающего меню почти не отличается от программного создания обычного статического меню и включает в себя следующие действия:

- создание всплывающего меню для помещения в него требуемого набора пунктов (пока пустого) функцией CreatePopupMenu() с получением дескриптора всплывающего меню (линейка меню в этом случае не создается);
- заполнение всплывающего меню конкретными пунктами функцией AppendMenu() в точности так же, как и при создании обычного меню. Поскольку линейки меню в этом случае нет, всплывающее меню ни к чему не подсоединяется;
- объявление созданного всплывающего меню плавающим функцией TrackPopupMenu(). Второй параметр этой функции задается равным 0. В качестве следующих двух параметров следует указать текущие координаты курсора мыши, а далее дескриптор окна и NULL (в качестве последнего параметра.

Последний пункт требует обсуждения. В справке по функции TrackPopupMenu() указано, что эта функция рассматривает передаваемые ей координаты как экранные, в то время как при вызове функции OnRButtonDown() в нее передаются координаты относительно левого верхнего угла рабочей области окна. Для коррекции координат можно воспользоваться функцией GetWindowRect(), которая возвращает текущие координаты левого верхнего угла окна приложения относительно начала экрана. Однако и в этом случае плавающее окно будет несколько смещено относительно положения указателя мыши, поскольку рабочая область окна не совпадает со всем пространством окна из-за наличия толстой рамки и строки заголовка. Для точного позиционирования плавающего меню относительно курсора мыши придется учесть размеры и этих элементов. Лучше всего получить координаты курсора мыши не из параметров функции OnRButtonDown(), а непосредственно из пакета сообщения (структурной переменной Msg), куда системой передаются экранные координаты мыши.

рес строки текста, представляющей собой название данного пункта меню (для разделительной линии этому параметру дается значение NULL);

- включение всплывающего меню в линейку меню и задание его имени той же функцией AppendMenu(), в качестве второго параметра которой следует указать константу MF_POPUP, в качестве третьего – дескриптор всплывающего меню, преобразованный в тип UINT, а в качестве четвертого – адрес строки с названием всего всплывающего меню;
- присоединение всего меню к главному окну приложения функцией SetMenu().

Все эти действия удобнее всего выполнить в функции OnCreate(); там же можно объявить символьные строки с названиями пунктов меню. На рис. 6.6 показано меню из программы 6-1 с комментариями, поясняющими использованную в этом разделе терминологию.

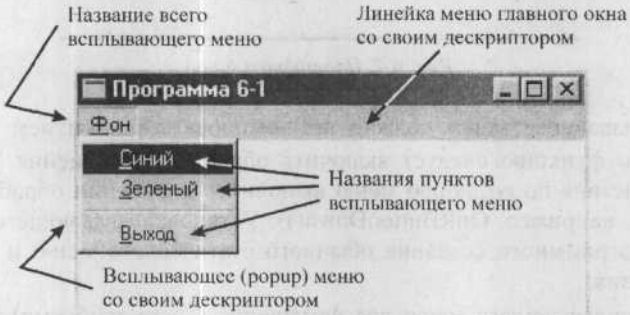


Рис. 6.6. Меню, созданное программно

Приведем модифицированный вариант функции OnCreate() из программы 6-1, в которой программным образом создается меню. Остальные части программы не изменяются за исключением того, что при регистрации класса окна член wc.lpszMenuName, в который обычно заносится имя меню в файле ресурсов, остается неинициализированным.

```
/*Функция OnCreate, создающая меню программным образом*/
BOOL OnCreate(HWND hwnd,LPCREATESTRUCT){
/*Создадим новые кисти*/
  hBlueBrush=CreateSolidBrush(RGB(100,100,255));
  hGreenBrush=CreateSolidBrush(RGB(100,255,100));
/*Создадим меню*/
  char str0[]="Фон";
  char str1[]="Синий";
  char str2[]="Зеленый";
  char str3[]="Выход";
  HMENU hMenu=CreateMenu();
  HMENU hPopupMenu=CreatePopupMenu();
  AppendMenu(hPopupMenu,MF_STRING,MI_BLUE,str1);
  AppendMenu(hPopupMenu,MF_STRING,MI_GREEN,str2);
  AppendMenu(hPopupMenu,MF_SEPARATOR,0,NULL);
  AppendMenu(hPopupMenu,MF_STRING,MI_EXIT,str3);
  AppendMenu(hMenu,MF_POPUP,(UINT)hPopupMenu,str0);
  SetMenu(hwnd,hMenu);
  return TRUE;
  }
```

Разумеется, ни внешний вид, ни функционирование меню не зависят от того, каким образом – с помощью файла ресурсов или программно – оно создается.

выборе того или иного пункта меню мы попадаем не в функцию OnPaint(), а в функцию OnCommand(). Для передачи в функцию OnPaint() информации о выбранном пункте меню и, соответственно, цвете фона, можно использовать переключатель в виде глобальной переменной (в нашем случае sw), которой в функции OnCommand() присваивается, в зависимости от выбранного пункта меню, то или иное значение (у нас 1 или 2). В функции же OnPaint() это значение анализируется в предложении if (или конструкцией switch-case) и выполняются необходимые действия по перекраске фона.

Однако выбор пункта меню не приведет к посылке в приложение сообщения WM_PAINT. Получится, что переключатель sw мы установим, но изображение в окне не изменится. Для того, чтобы после установки переключателя заставить Windows перекрасить окно, следует воспользоваться специальной функцией Windows InvalidateRect(), которая объявляет окно (или задаваемую часть окна) затертым. Windows, получив информацию о затирании окна, немедленно посылает в приложение сообщение WM_PAINT, что приводит к вызову функции OnPaint() и действительному перерисовыванию содержимого окна.

Через второй параметр функции InvalidateRect() передаются координаты той области окна, которую мы желаем перерисовать. Поскольку в данном случае требуется перерисовать все окно, в качестве второго параметра указывается NULL. Третий параметр функции InvalidateRect() задает режим стирания предыдущего изображения. Если значение этого параметра равно TRUE, старое изображение стирается, если FALSE – остается. В случае перерисовки фона окна значение этого параметра не отражается на результате, однако при управлении посредством меню вывода в окно тех или иных фигур (например, различных графиков) старое изображение, как правило, следует стирать, т. е. в качестве третьего параметра функции InvalidateRect() указывать TRUE.

Перекраска фона окна осуществляется посредством вызова функции Windows FillRect(). В качестве второго параметра этой функции следует передать информацию о координатах закрашиваемого прямоугольника. Эту информацию (в виде адреса структуры типа RECT) можно извлечь из структурной переменной ps, заполняемой функцией BeginPaint().

Программное создание меню

Как уже отмечалось, меню чаще всего описывается в файле ресурсов, так как в этом случае облегчается задача изменения его состава или поясняющих надписей. Однако в некоторых случаях может оказаться более удобным создать меню программно, без помощи файла ресурсов. Программное создание меню включает в себя следующие действия:

- создание линейки меню приложения (пока пустой) функцией CreateMenu() с получением дескриптора этого меню;
- создание всплывающего меню для помещения в него требуемого набора пунктов (но пока тоже пустого и даже без названия) функцией CreatePopupMenu() с получением дескриптора всплывающего меню;
- заполнение всплывающего меню конкретными пунктами функцией AppendMenu(). В качестве второго параметра этой функции указывается константа MF_STRING (или MF_SEPARATOR для разделительной линии), в качестве третьего – символическое обозначение идентификатора данного пункта меню, которое должно быть определено в файле .H, а в качестве последнего, четвертого параметра – ад-

клавиши, а ее виртуальный код, в котором нет информации о том, ввел ли пользователь русскую или латинскую букву. Для преобразования кодов виртуальных клавиш в коды ASCII и служит функция TranslateMessage().

Эта функция, обнаружив в очереди сообщение WM_SYSKEYDOWN, транслирует код виртуальной клавиши в код ASCII и посылает в приложение еще одно сообщение – WM_SYSCHAR, передавая с ним и этот код (в кодировке, принятой в Windows). Программы же Windows, обслуживающие меню, не реагируют на сообщения WM_SYSKEYDOWN, однако воспринимают сообщения WM_SYSCHAR, и при получении кода ASCII "горячей" клавиши открывают соответствующее меню. Между прочим, этот механизм используется только при выборе пунктов линейки меню, располагаемой по верхнему краю главного окна. Выбор пунктов открывающихся меню осуществляется нажатием обычных, а не системных алфавитно-цифровых клавиш (т. е. без клавиши Alt) и не имеет отношения к функции TranslateMessage().

Сообщение WM_COMMAND

Для того программа отзывалась на выбор пользователем тех или иных пунктов меню, необходимо включить в оконную функцию обработку сообщения WM_COMMAND. Это сообщение поступает в приложение при выборе пользователем команды меню, причем второй по порядку параметр (id), передаваемый макросом HANDLE_MSG в прикладную функцию OnCommand() обработки этого сообщения, соответствует идентификатору выбранной команды меню. Таким образом, алгоритм функции OnCommand() вполне очевиден – при поступлении идентификатора со значением MI_BLUE закрасить главное окно приложения синим цветом, при поступлении MI_GREEN – зеленым, если же id имеет значение MI_EXIT – завершить приложение.

Для завершения приложения достаточно вызвать функцию DestroyWindow(), в качестве параметра которой следует указать дескриптор главного окна, полученный в качестве параметра функции OnCommand(). Функция DestroyWindow() уничтожает окно и посылает в приложение сообщение WM_DESTROY, что приводит к завершению приложения.

В нашей простой программе перед ее завершением не предусмотрено каких-либо действий вроде уничтожения созданных объектов. Поэтому вместо вызова (с помощью функции DestroyWindow()) сообщения WM_DESTROY, которое через оконную функцию главного окна приведет к вызову функции PostQuitMessage(), можно было сразу вызвать эту функцию. В этом случае завершение приложения произойдет помимо функции обработки сообщения WM_DESTROY: функция PostQuitMessage() пошлет в очередь сообщений приложения сообщение WM_QUIT, цикл обработки сообщений разорвется и приложение будет завершено. Лучше, однако, всегда для завершения приложения использовать функцию DestroyWindow(), чтобы вся процедура завершения происходила стандартным образом, через функцию обработки сообщения WM_DESTROY.

С другой стороны, функция DestroyWindow(), как и следует из ее названия, уничтожает главное окно (если ее аргумент является дескриптором главного окна) и создается впечатление, что приложения уже нет и не надо думать о его завершении. Это не так. Уничтожение окна не приводит к завершению приложения, и, если не вызвать функцию PostQuitMessage(), приложение будет "висеть" в системе и Windows будет функционировать неправильно.

Рассмотрим теперь, как выполняется перекраска фона окна. Как уже говорилось, рисовать что-либо в окне, в частности, перекрашивать его фон, можно только в функции OnPaint(), вызываемой в ответ на приход в приложение сообщения WM_PAINT. Лишь в этом случае при затирании окна изображение в нем будет восстанавливаться. Однако при

Как уже отмечалось, меню можно вкладывать друг в друга. Для того чтобы выбор пункта меню приводил к открытию вложенного списка команд (подменю), этот пункт следует описывать ключевым словом POPUP (вместо MENUITEM). Например, приведенный ниже сценарий вызовет развертывание команды "О программе" во вложенный список из двух команд (рис. 6.5).

```
Main MENU{
  POPUP "&Файл"  {
    POPUP "&О программе"{
      MENUITEM "Об авторе...", MI_AUTHOR
      MENUITEM "О поставщике...",MI_VENDOR
      }
    MENUITEM SEPARATOR
    MENUITEM "В&ыход",MI_EXIT
    }
}
```

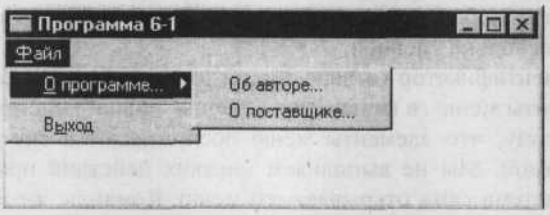


Рис. 6.5. Вложенное меню

Использованные нами символические идентификаторы MI_AUTHOR и MI_VENDOR должны быть определены в файле заголовков..

Описание сценария меню в файле ресурсов не приведет к его появлению в главном окне приложения. Для того чтобы линейка меню была включена в состав приложения как его активный элемент, необходимо присвоить члену структуры типа WNDCLASS lpszMenuName адрес строки с именем меню, под которым оно фигурирует в файле ресурсов. Как уже отмечалось, вместо адреса строки с именем, описанной где-то в другом месте, можно просто указать это имя в кавычках. Фактически сама строка будет размещена в сегменте данных приложения, а в переменную wc.lpszMenuName будет действительно занесен ее адрес.

Функция TranslateMessage()

Обратимся теперь к тексту программы и рассмотрим ее характерные особенности. Прежде всего, в этой программе несколько изменился состав цикла обработки сообщений. Раньше в него входила только одна функция DispatchMessage(), теперь к ней добавилась вторая – TranslateMessage() (функция GetMessage() не входит в тело цикла, а служит "поставщиком" условия дальнейшего выполнения цикла). Поскольку тело цикла while теперь содержит блок из двух операторов, оно заключается в операторные скобки {...}. Функция TranslateMessage() включается в цикл обработки сообщений в тех случаях, когда в приложении предусматривается ввод с клавиатуры и, в частности, если мы хотим, чтобы выбирать пункты меню можно было не только щелчком мыши, но и нажатием на клавиатуре системных клавиш. Системной называется любая алфавитно-цифровая клавиша, если она нажата при нажатой клавише Alt. Такие комбинации клавиш Windows обрабатывает особым образом, посылая в приложение вместо сообщения WM_KEYDOWN сообщение WM_SYSKEYDOWN. Однако управлять пунктами меню с помощью этого сообщения затруднительно, потому что в нем передается не код ASCII нажатой

имени элемента меню следует предварить знаком амперсанда (&). В нашей программе линейка меню имеет единственное меню "Фон", которое будет вызываться (открываться) нажатием русской буквы Ф при нажатой клавише Alt.

Вслед за предложением POPUP следует перечень пунктов, или команд этого меню, который появится на экране при его выборе. Перечень заключается в операторные скобки {...}. Каждый пункт начинается ключевым словом MENUITEM, за которым указывается название данной команды и ее идентификатор. Идентификаторы обычно имеют символическую форму. У нас это обозначения MI_BLUE, MI_GREEN и MI_EXIT (MI_ от Menu Item, пункт меню), которым были присвоены произвольные значения 100, 101 и 102. В дальнейшем при выборе той или иной команды меню система Windows передаст в приложение в сообщении WM_COMMAND идентификатор выбранной команды, что позволит программе определить, какая именно команда была выбрана пользователем.

Часто пункты меню представляют собой не просто команды, а названия вложенных меню (подменю), которые раскрываются при их выборе в перечень команд. Вложенность меню может иметь несколько уровней.

Заметьте, что идентификатор (в виде имени Main) имеет также вся линейка меню, в то время как элементы меню (в нашем случае меню "Файл") идентификатора не имеют. Это происходит потому, что элементы меню обслуживаются системой Windows, а не прикладной программой. Мы не выполняем никаких действий при щелчке мышью по элементу "Файл"; система сама открывает его меню. Команды же меню обслуживаются приложением, и, чтобы их можно было различить, им присваиваются идентификаторы.

Предложение

```
MENUITEM SEPARATOR
```

служит для проведения в меню горизонтальной черты, разделяющей группы команд, и, естественно, идентификатора не имеет.

В нашей программе линейка меню имеет всего один элемент (одно меню) "Фон". Для того чтобы увеличить число элементов меню, следует включить в сценарий дополнительные предложения POPUP. Например, сценарий

```
Main MENU{
  POPUP "&Файл"{
    ...}
  POPUP "&Правка"{
    ...}
  POPUP "&Вид"{
    ...}
  POPUP "Фор&мат"{
    ...}
  }
```

создаст линейку меню, показанную на рис. 6.4. Следует только иметь в виду, что элементы меню не могут быть пустыми, в них должно входить по меньшей мере по одной команде (с ключевым словом MENUITEM).

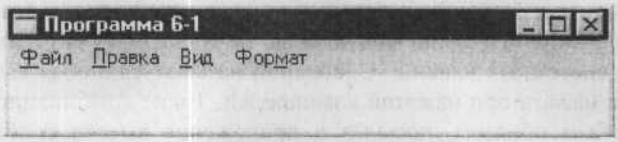


Рис. 6.4. Вид линейки меню с несколькими пунктами

Преимущества, возникающие при выделении определенных объектов в файл ресурсов, носят многоплановый характер:

- Файл ресурсов проще выглядит и имеет заметно большую наглядность, чем исходный текст программы. Редактирование состава меню и диалоговых окон или содержимого текстовых строк удобнее выполнять в отдельном файле ресурсов, чем в исходном тексте программы.
- Использование ресурсов упрощает структуру и содержание программы, так как при описании, например, меню в файле ресурсов система берет на себя значительную часть работы по организации и обслуживанию меню.
- Специальные инструментальные средства, разработанные для редактирования ресурсов, в частности входящая в состав IDE Borland C++ программа Resource Workshop (Мастерская ресурсов), существенно облегчают разработку и редактирование меню, диалогов, пиктограмм и курсоров. То же можно сказать про среду Visual C++, в которую встроены средства визуального создания и редактирования ресурсов.
- Ресурсы можно редактировать непосредственно в загрузочном файле приложения (с помощью упомянутого выше редактора ресурсов). Это, в частности, дает возможность переводить тексты сообщений и меню на другой язык, не имея исходных текстов приложений.
- Ресурсы, например, изображения элементов управления, кнопок, курсоров, пиктограмм и пр. можно извлечь из загрузочного файла приложения, чтобы использовать их в другом приложении.

Поскольку теперь наше приложение, кроме исходного файла с текстом программы, включает еще и файл ресурсов, процедура подготовки проекта немного изменяется.

Открыв кадр IDE, выберите пункт меню File➢New➢Project и в верхней части открывшейся панели Target Expert (New Target) в рамке Project Path and Name укажите имя файла проекта (с расширением .IDE).

Нажмите кнопку Advanced. В открывшейся панели Advanced options уберите галочку на кнопке выбора .DEF, так как этого файла у нас по-прежнему нет, но оставьте отмеченной кнопку, относящуюся к файлу .RC. Подтвердите настройки, выбрав OK в панели Advanced options и еще раз OK в панели Target Expert.

Обратите внимание на то, что заголовочные файлы не входят в проект приложения. Они подсоединяются к тексту программы препроцессором как результат выполнения директив #include.

Описание меню в файле ресурсов

Состав, или сценарий, меню описывается с помощью ключевых слов или предложений языка описания ресурсов. Сценарий меню главного окна начинается с ключевого слова MENU, которое может быть написано как прописными, так и строчными буквами и за которым следует перечень элементов линейки меню, заключенный в "операторные скобки" – операторы BEGIN и END – или просто в фигурные скобки { и }. Ключевое слово MENU предваряется произвольным именем (у нас это имя Main), которое выступает как идентификатор всей линейки меню и будет использовано далее в тексте программы.

Каждый элемент перечня начинается с ключевого слова POPUP, за которым следует название этого элемента. Обычно элементы линейки меню называют просто меню. Если в программе предусмотрено, как это обычно и делается, управление меню не только мышью, но и клавиатурой (нажатием комбинаций клавиш Alt+*буква*), то "горячую" букву

```
#include "6-1.h"
```

Более принципиальным является файл ресурсов, в который в нашем случае включено описание меню. Кроме того, в файле ресурсов можно описывать используемые в приложении диалоговые окна, растровые изображения, пиктограммы, курсоры, таблицы символьных строк, шрифты и т. д.

Файл ресурсов является текстовым файлом и имеет стандартное расширение .RC. Различные ресурсы описываются в нем в специальных форматах, понятных для компилятора ресурсов BRC.EXE, входящего в состав пакета Borland C++. Обработка исходного текста самой программы и исходного текста ресурсов происходит, можно сказать, параллельно (рис. 6.2): компилятор исходного текста BCC32.EXE обрабатывает файл .CPP, образуя двоичный объектный файл с расширением .OBJ, а компилятор BRC.EXE обрабатывает файл .RC, получая из него промежуточный файл с расширением .RES.

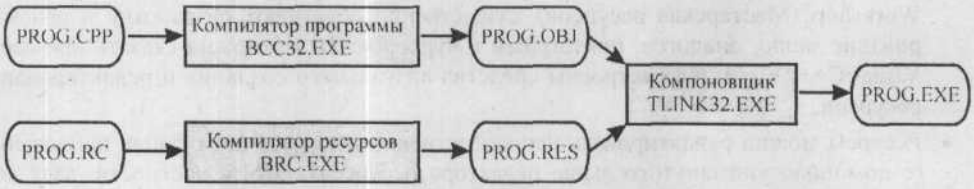


Рис. 6.2. Процесс создания выполнимого файла программы с ресурсами

Далее компоновщик TLINK32.EXE компонует файлы .OBJ и .RES, собирая из них единый загрузочный файл с расширением .EXE, который включает в себя программные коды, данные и ресурсы. После образования загрузочного файла промежуточные файлы с расширениями .OBJ и .RES уже не нужны; их вполне можно удалить, так как все необходимое для выполнения программы входит в файл .EXE.

В действительности перечисленные выше компоненты пакета Borland C++ имеют более универсальный характер. Так, программа BCC32.EXE может не только компилировать, но и компоновать загрузочный модуль, если ей предоставить результат компиляции ресурсов – файл .RES. Тем же свойством обладает и программа BRC.EXE – при наличии готового (но еще без ресурсов) файла .EXE она не только создает "объектный" файл ресурсов .RES, но и добавляет эти ресурсы в файл .EXE, создавая тем самым окончательный вариант выполнимого файла.

В вопросах взаимодействия отдельных компонентов пакета программирования приходится разбираться лишь в тех случаях, когда сборка приложения по каким-то причинам выполняется в сеансе DOS путем последовательного вызова компиляторов и компоновщиков. Обычно приложения создаются с помощью интегрированной среды разработки IDE (Borland C++ или Visual C++), где все эти программы вызываются автоматически.

Существенно, что элементы приложения, описанные в файле ресурсов (например, форматы меню и диалогов, а также тексты и рисунки) не разбросаны по загрузочному файлу, а сосредоточены в одном месте и хранятся в определенном формате (рис. 6.3). Это дает возможность специальным программам, понимающим этот формат – редакторам ресурсов – читать или даже модифицировать ресурсы непосредственно в загрузочном файле приложения – действие, совершенно невозможное в традиционных программах.

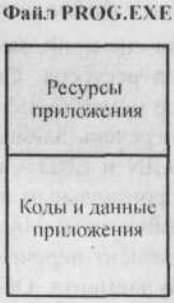


Рис. 6.3. Размещение ресурсов в загрузочном файле приложения

```
/*Функция OnCreate обработки сообщений WM_CREATE*/
BOOL OnCreate(HWND,LPCREATESTRUCT){
/*Создадим новые кисти*/
  hBlueBrush=CreateSolidBrush(RGB(100,100,255));
  hGreenBrush=CreateSolidBrush(RGB(100,255,100));
  return TRUE;
  }
/*Функция OnCommand обработки сообщений WM_COMMAND
void OnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id){//id = идентификатор выбранного пункта меню
    case MI_BLUE:
      sw=1;
      InvalidateRect(hwnd,NULL,TRUE);
      break;
    case MI_GREEN:
      sw=2;
      InvalidateRect(hwnd,NULL,TRUE);
      break;
   case MI_EXIT:
     DestroyWindow(hwnd);
     }
  }
/*Функция OnPaint обработки сообщений WM_PAINT*/
void OnPaint(HWND hwnd){
  PAINTSTRUCT ps;//Структура для функции BeginPaint()
  HDC hdc=BeginPaint(hwnd,&ps);//Получим контекст устройства
  if(sw==1)//Если выбрано "Синий"
    FillRect(hdc,&ps.rcPaint,hBlueBrush);
  if(sw==2)//Если выбрано "Зеленый"
    FillRect(hdc,&ps.rcPaint,hGreenBrush);
  EndPaint(hwnd,&ps);//Освободим контекста устройства
  }
/*Функция OnDestroy обработки сообщения WM_DESTROY*/
void OnDestroy(HWND){
  PostQuitMessage(0);
  }
```

Файлы заголовков и ресурсов

До сих пор весь исходный текст каждого приложения размещался в единственном файле с расширением .CPP. Для простых программ это допустимо, однако практически любое приложение Windows помимо файла с исходным текстом программы содержит еще и собственный файл заголовков. Необходимости в этом файле нет: все его содержимое может размещаться в исходном файле, однако его использование несколько уменьшает объем исходного файла и упрощает работу с ним. В заголовочный, или включаемый (include-), файл обычно выносят прототипы прикладных функций, используемых в программе, а также определения символических констант и макросов, разработанных программистом. Удобство использования заголовочных файлов заключается, в частности, в том, что, собрав однажды в такой файл необходимый набор прототипов и макросов, программист включает затем всю эту информацию в исходные тексты разрабатываемых приложений с помощью единственного предложения #include.

Файл 6-1.H, входящий в проект нашего приложения, содержит прототипы описываемых в программе функций и определения трех констант MI_BLUE, MI_GREEN и MI_EXIT, которые используются затем как в файле ресурсов, так и в исходном тексте программы. Для того чтобы при обработке этих файлов компиляторы могли использовать определения файла 6-1.H, и файл ресурсов и исходный файл начинаются с предложения препроцессора

```
/*Прототипы функций*/
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
BOOL OnCreate(HWND,LPCREATESTRUCT);
void OnPaint(HWND);
void OnCommand(HWND,int,HWND,UINT);
void OnDestroy(HWND);
/*Файл ресурсов 6-1.RC*/
#include "6-1.h"
Main MENU{
  POPUP "&Фон"  {
    MENUITEM "&Синий",MI_BLUE
    MENUITEM "&Зеленый",MI_GREEN
    MENUITEM SEPARATOR
    MENUITEM "&Выход",MI_EXIT
    }
  }
/*Файл с исходным текстом программы 6-1.CPP*/
#include <windows.h>
#include <windowsx.h>
#include "6-1.h"
HBRUSH hBlueBrush,hGreenBrush;//Дескрипторы кистей фона
int sw;//Переключатель для управления фоном окна
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  char szClassName[]="MainWindow";
  char szTitle[]="Программа 6-1";
  MSG Msg;
  WNDCLASS wc;
/*Зарегистрируем класс главного окна*/
  ZeroMemory(&wc,sizeof(wc));
  wc.lpfnWndProc=WndProc;
  wc.hInstance=hInst;
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
  wc.lpszMenuName="Main";//Имя меню в файле ресурсов
  wc.lpszClassName=szClassName;
  RegisterClass(&wc);
/*Создадим главное окно и сделаем его видимым*/
  HWND hwnd=CreateWindow(szClassName,szTitle,
      WS_OVERLAPPEDWINDOW,10,10,200,100,
      HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_SHOWNORMAL);
/*Организуем цикл обработки сообщений*/
  while(GetMessage(&Msg,NULL,0,0))
    DispatchMessage(&Msg);
  return 0;
  }
/*Оконная функция WndProc главного окна*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,
          WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);
    HANDLE_MSG(hwnd,WM_CREATE,OnCreate);
    HANDLE_MSG(hwnd,WM_PAINT,OnPaint);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
```

Глава 6
Ресурсы: меню и диалоги

В этой главе мы рассмотрим средства включения в программу двух важнейших инструментов Windows: меню и диалога. Линейка меню является непременным атрибутом главного окна практически всех приложений Windows и основным средством активизации тех или иных программных средств, предусмотренных в приложении; диалоговые окна используются, в частности, для настройки приложения и выбора режимов его работы. При этом диалог, сам представляя собой окно со специальными свойствами, может существовать и без главного окна, образуя специфический тип "безоконного" приложения. Такие приложения, состоящие из одного диалогового окна, очень удобны для всякого рода экспериментов и исследований и широко используются как методическое средство в повседневной работе программиста.

Меню в главном окне приложения

Простая программа с меню

Рассмотрим простую программу с меню, с помощью которого изменяется цвет фона главного окна. В программе предусмотрена линейка меню с единственным пунктом "Фон". При щелчке по нему левой клавишей мыши открывается меню из трех команд: "Синий", "Зеленый" и "Выход" (такое меню называют всплывающим, popup). Выбор каждого пункта всплывающего меню приводит к соответствующему действию: перекраске фона окна или завершению всего приложения. На рис. 6.1 показано окно приложения с открытым меню.

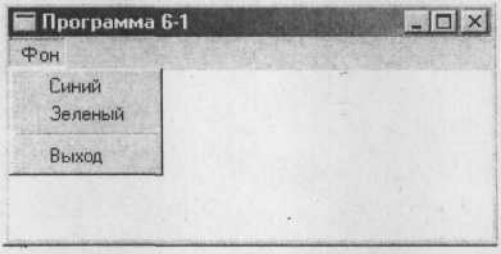


Рис. 6.1. Главное окно приложения с открытым меню

Меню можно создать в программе различными способами. Наиболее распространенным и удобным является включение в приложение меню в качестве *ресурса*; в этом случае форма меню описывается в специальном файле ресурсов (в нашем случае файл 6-1.RC). Кроме этого, в рассматриваемый программный комплекс включен еще один дополнительный файл 6-1.H (заголовочный файл), в который собраны, как это обычно и делается, определения используемых в программе констант (идентифицирующих пункты меню), а также прототипы прикладных функций.

```
/*Программа 6-1. Меню в главном окне*/
/*Заголовочный файл 6-1.H*/
/*Определения констант*/
#define MI_BLUE 100
#define MI_GREEN 101
#define MI_EXIT 102
```

определена переменная szYears[10], являющаяся *массивом указателей* на символы, о чем говорит ее тип char* (указатель на символы) и знак размерности массива [10]. Элементы этого массива представляют собой указатели на символьные строки "1993", "1994" и т. д. Таким образом, адрес строки "1993" равен szYears[0], адрес строки "1994" равен szYears[1], а адрес i-й строки – szYears[i]. Именно это обозначение использовано в операторе цикла, выводящем значения лет под графиком:

```
/*Выведем строку лет*/
  for(int i=0;i<10;i++)
    TextOut(hdc,100+i*45,330,szYears[i],strlen(szYears[i]));
```

Сколько места в памяти занимает массив szYears[10]? Каждый его элемент определен как символьная строка (в двойных кавычках). Как уже отмечалось ранее, компилятор, выделяя память под символьные строки, добавляет к концу каждой строки завершающий нуль. Таким образом, массив szYears представляет собой такую последовательность байтов:

```
'1', '9', '9', '3', 0, '1', '9', '9', '4', 0, '1', '9', '9', '5', 0 ,...
```

Полный размер массива составляет 5x10=50 байт.

Аналогично задаются надписи масштаба по вертикальной оси

```
char *szScale[3]={"100"," 50","  0"};//Шкала
```

только здесь массив указателей на символы szScale состоит всего из трех элементов.

тельные строки не накладывались друг на друга, необходимо с помощью структуры типа TEXTMETRIC и функции GetTextMetrics() определить высоту символов и, исходя из этой величины, задавать координаты следующих строк.

Функция DrawText() позиционирует текст по-другому. Из ее прототипа, взятого из интерактивного справочника

```
int DrawText(HDC hDC,LPCTSTR lpString,int nCount,LPRECT lpRect,UINT uFormat);
```

видно, что в качестве ее параметров выступают: дескриптор контекста устройства hDC, адрес выводимой строки lpString, длина выводимой строки nCount, адрес lpRect структурной переменной типа RECT с координатами прямоугольной области экрана, в которую выводится текст, а также константа форматирования текста uFormat.

Функция DrawText() выводит заданный ей текст в указанную прямоугольную область (которая может быть и гораздо больше выводимой строки по размеру), причем расположение текста внутри области определяется последним параметром функции. Использованная в программе символическая константа DT_CENTER позиционирует текст по горизонтали по центру прямоугольника (рис. 5.11).

Рис. 5.11. Расположение текстовой строки в окне приложения при использовании функции DrawText()

Текст может быть позиционирован также по левому краю заданной области (константа DT_LEFT) или по правому краю (константа DT_RIGHT).

Если текст состоит из одной строки, а высота области превышает высоту текста (в программе 5-3 они совпадают), то строку можно позиционировать и по вертикали, для чего используются константы DT_VCENTER, DT_BOTTOM и DT_TOP. Позиционирование по вертикали требует явного объявления "однострочности" текста с помощью константы DT_SINGLELINE. Все константы позиционирования можно комбинировать с помощью побитовой операции ИЛИ; таким образом, для вывода строки текста в середину относительно большого прямоугольника последний параметр функции DrawText() примет следующий вид:

```
DT_SINGLELINE|DT_VCENTER|DT_CENTER
```

В нашем примере ширина прямоугольников r для размещения заголовка "Колледж "Рубикон" и подзаголовка "Выпуск слушателей по годам" выбрана равной ширине окна XMAX, а высота совпадает с высотой соответствующего шрифта, определяемой для каждого шрифта с помощью структуры TEXTMETIC и функции GetTextMetrics(). Текст подзаголовка смещен по вертикали на высоту шрифта заголовка плюс 5 пикселов.

Стоит обратить внимание на задание надписей по осям. В предложении

```
char *szYears[10]={"1993","1994","1995","1996","1997",//Годы под
                   "1998","1999","2000","2001","2002"};//диаграммой
```

Структура LOGFONT содержит довольно много членов, однако обычно можно ограничиться заданием лишь небольшой их части. Следует только иметь в виду, что неправильная установка того или иного члена этой структуры может привести к весьма неприятным последствиям, так как Windows, не сумев создать в точности заказанный вами шрифт, будет пытаться подобрать наиболее подходящий; часто в этом случае подбирается шрифт, весьма далекий от заказанного.

Структуру LOGFONT целесообразно использовать для задания характеристик масштабируемых (TrueType) шрифтов; только в этом случае будут действовать такие, например, характеристики шрифта, как угол наклона или размер (растровые шрифты допускают изменение размера, но лишь в обусловленных пределах или при низком качестве увеличения).

Структура LOGFONT имеет следующий состав членов:

```
typedef struct tagLOGFONT
  int   lfHeight;//Высота
  int   lfWidth;//Средняя ширина; если=0, устанавливается разумно
  int   lfEscapement;//Угол наклона в 1/10 градуса
  int   lfOrientation;//Не используется
  int   lfWeight;//Насыщенность: FW_NORMAL, FW_BOLD, FW_LIGHT
  BYTE  lfItalic;//Если=1, то курсив
  BYTE  lfUnderline;//Если=1, то подчеркивание
  BYTE  lfStrikeOut;//Если=1, то перечеркивание
  BYTE  lfCharSet;//Набор символов; обычно=0
  BYTE  lfOutPrecision;//Требуемая точность соответствия; обычно=0
  BYTE  lfClipPrecision;//Способ вырезки части символа; обычно=0
  BYTE  lfQuality;//Качество; обычно=0
  BYTE  lfPitchAndFamily;//Шаг и семейство; обычно=0
  BYTE  lfFaceName[LF_FACESIZE];//Имя начертания шрифта
  } LOGFONT;
```

Наиболее важным является последней член описанной выше структуры, символьный массив lfFaceName. В него надо скопировать полное имя шрифта из числа шрифтов, установленных в Windows, например Times New Roman Cyr, Arial Cyr, Courier New Cyr и т. д. Обратите внимание на то, что в член lfFaceName заносится не адрес имени шрифта, а сама символьная строка с именем. Поэтому этот член нельзя инициализировать оператором присваивания; приходится воспользоваться функцией Си++ для копирования строк strcpy().

В нашей программе местом создания новых шрифтов выбрана функция OnCreate() обработки сообщения WM_CREATE; в ней последовательно создаются 4 различных шрифта путем заполнения необходимых полей структуры LOGFONT и вызова функции CreateFontInderect(). Возвращаемые этой функцией дескрипторы шрифтов сохраняются в глобальных переменных hFont1, hFont2, hFont3 и hFont4.

Формирование экранного кадра осуществляется, как это и положено, в функции OnPaint() обработки сообщения WM_PAINT; здесь по мере необходимости выбирается в контекст устройства дескриптор того или иного шрифта и соответствующей функцией Windows заданный текст выводится на экран.

Вывод текста с помощью функций TextOut() и DrawText()

До сих пор мы сталкивались только с одной функцией вывода на экран текстовых строк – TextOut(). В настоящей программе использована и другая функция с бóльшими возможностями – DrawText().

Функция TextOut() позволяет вывести в окно приложения строку текста начиная с точно заданной точки, координаты которой указываются в параметрах функции. Координаты относятся к верхнему левому углу знакоместа. Поскольку шрифты в Windows могут иметь различный размер, высота знакоместа обычно заранее неизвестна и для того, чтобы последова-

```
  GetTextMetrics(hdc,&tm);//Получим метрики текста
  r.left=0;//Задание координат прямоугольника для
  r.top=r.bottom+5;//строки с подзаголовком
  r.right=XMAX;
  r.bottom+=tm.tmHeight+5;
  DrawText(hdc,szSubTitle,strlen(szSubTitle),&r,DT_CENTER);//Выведем текст
/*Выведем строку лет*/
  for(int i=0;i<10;i++)
    TextOut(hdc,100+i*45,330,szYears[i],strlen(szYears[i]));
/*Выведем вертикальную строку по оси y*/
  SelectFont(hdc,hFont3);//Выберем шрифт 3 (вертикальный)
  SetTextColor(hdc,RGB(0,0,0));//Установим черный цвет
  TextOut(hdc,30,320,szYAxes,strlen(szYAxes));//Выведем надпись
  SelectFont(hdc,hFont4);//Выберем шрифт 4 (нормальный мелкий)
  for(int i=2;i>=0;i--)//Выведем значения масштаба
    TextOut(hdc,60,110+i*100,szScale[i],strlen(szScale[i]));
  SelectFont(hdc,hOldFont);
/*Нарисуем столбцовую диаграмму*/
  SelectBrush(hdc,hBrush);//Выберем кисть в контекст
  for(int i=0;i<10;i++)//Выведем стобцы диаграммы
    Rectangle(hdc,100+i*45,320-nData[i]*2,100+i*45+30,320);
  EndPaint(hwnd,&ps);//Освободим контекст устройства
  }
/*Функция OnDestroy обработки сообщения WM_DESTROY*/
void OnDestroy(HWND){
  PostQuitMessage(0);
  }
```

Программа 5-3 имеет традиционную структуру. В ней, как и в предыдущем приложении, обрабатываются 3 сообщения: WM_CREATE, WM_PAINT и WM_DESTROY. В функции OnCreate() создаются требуемые инструменты – шрифты и кисть для заливки столбцов диаграммы; в функции OnPaint() главное окно заполняется изображением; в функции OnDestroy() удаляются созданные ранее инструменты и организуется завершение программы.

Поскольку в программе несколько раз используются размеры главного окна, их значения определены в секции операторов препроцессора с помощью директивы #define:

```
#define XMAX 600//Размер окна по горизонтали
#define YMAX 400//Размер окна по вертикали
```

В секции глобальных переменных объявлены 4 переменные типа HFONT для дескрипторов шрифтов hFont1, hFont2, hFont3 и hFont4, а также переменная типа HBRUSH для дескриптора кисти hBrush.

Процедура создания нового шрифта довольно проста. Для этого нужно объявить в программе структурную переменную типа LOGFONT, заполнить ее поля требуемыми значениями и вызвать функцию Windows CreateFontInderect(). Эта функция вернет дескриптор нового шрифта; после выбора полученного дескриптора в контекст устройства любая функция вывода на экран текста будет использовать именно этот шрифт. Если в приложении желательно использовать разные шрифты, их можно создать заранее и выбирать в контекст устройства по мере необходимости. Функция Create FontInderect() использует в качестве исходного материала *физический шрифт*, хранящийся на диске в виде файла; результатом работы этой функции будет *логический шрифт*, дескриптор которого и загружается в контекст устройства. Вывести на экран текст непосредственно физическим шрифтом нельзя, так как функции GDI работают только с логическими шрифтами. Даже если мы хотим иметь шрифт с характеристиками, в точности соответствующими физическому шрифту, все равно из него сначала надо образовать логический шрифт (обнулив все члены структуры LOGFONT, кроме имени шрифта и его высоты) и лишь затем им пользоваться.

```
    HANDLE_MSG(hwnd,WM_PAINT,OnPaint);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
/*Функция OnCreate обработки сообщения WM_CREATE*/
BOOL OnCreate(HWND,LPCREATESTRUCT){
  char lpszFace1[]="Times New Roman Cyr";//Имя Windows для шрифта
  char lpszFace2[]="Arial Cyr";//Имя Windows для шрифта
  LOGFONT lf;//Структура LOGFONT для создания логических шрифтов
  ZeroMemory (&lf,sizeof(lf));//Обнулим структуру (обязательно!)
/*Создадим логический шрифт 1 (для заголовка)*/
  lf.lfHeight=60;//Размер
  strcpy(lf.lfFaceName,lpszFace1);//Скопируем имя шрифта
  hFont1=CreateFontIndirect(&lf);//Создание шрифта
/*Создадим логический шрифт 2 (для подзаголовка и цифр лет)*/
  lf.lfHeight=18;//Размер
  lf.lfItalic=1;//Курсив
  strcpy(lf.lfFaceName,lpszFace2);//Скопируем имя шрифта
  hFont2=CreateFontIndirect(&lf);//Создание шрифта
/*Создадим логический шрифт 3 (для оси y)*/
  lf.lfHeight=18;//Размер
  lf.lfItalic=0;//Отмена курсива
  lf.lfEscapement=900;//Угол наклона 90°
  hFont3=CreateFontIndirect(&lf);//Создание шрифта
/*Создадим логический шрифт 4 (для шкалы)*/
  lf.lfHeight=16;//Размер
  lf.lfEscapement=0;//Отмена угла наклона
  hFont4=CreateFontIndirect(&lf);//Создание шрифта
/*Создадим бирюзовую кисть*/
  hBrush=CreateSolidBrush(RGB(0,127,127));
  return TRUE;
  }
/*Функция OnPaint обработки сообщения WM_PAINT*/
void OnPaint(HWND hwnd){
  RECT r; //Прямоугольник для надписей
  PAINTSTRUCT ps;//Для функции BeginPaint()
  TEXTMETRIC tm;//Для получения характеристик шрифтов
  char szMainTitle[]="Колледж \"Рубикон\"";//Заголовок
  char szSubTitle[]="Выпуск слушателей по годам";//Подзаголовок
  char szYAxes[]="Количество выпускников";//Надпись по оси y
  char *szYears[10]={"1993","1994","1995","1996","1997",//Годы под
                     "1998","1999","2000","2001","2002"};//диаграммой
  char *szScale[3]={"100"," 50","  0"};//Шкала
  int nData[10]={20,35,35,30,70,75,85,90,90,95};//Данные для диаграммы
  HDC hdc=BeginPaint(hwnd,&ps); //Получим контекст устройства
/*Нарисуем большой прямоугольник*/
  Rectangle(hdc,90,120,550,320);//Рисуем прямоугольник; высота=320-120=200
/*Выведем строку заголовка*/
  HFONT hOldFont=SelectFont(hdc,hFont1);//Выберем в контекст шрифт 1
  SetTextColor(hdc,RGB(128,0,0));//Установим красный цвет
  GetTextMetrics(hdc,&tm);//Получим метрики текста
  r.left=0;//Задание координат прямоугольника, в который
  r.top=10;//будет выводиться текст заголовка
  r.right=XMAX;
  r.bottom=r.top+tm.tmHeight;//Высоту прямоугольника выберем по высоте шрифта
  DrawText(hdc,szMainTitle,strlen(szMainTitle),&r,DT_CENTER);//Выведем текст
/*Выведем строку подзаголовка*/
  SetTextColor(hdc,RGB(0,0,128));//Установим синий цвет
  SelectFont(hdc,hFont2);//Выберем шрифт 2 (курсив)
```

Рис. 5.10. Использование логических шрифтов

```
//Программа 5-3. Создание логических шрифтов
/*Операторы препроцессора*/
#define XMAX 600//Размер окна по горизонтали
#define YMAX 400//Размер окна по вертикали
#include <windows.h>
#include <windowsx.h>
/*Прототипы функций*/
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
BOOL OnCreate(HWND,LPCREATESTRUCT);
void OnPaint(HWND);
void OnDestroy(HWND);
/*Глобальные переменные*/
HFONT hFont1,hFont2,hFont3,hFont4;//Будут созданы 4 логических шрифта
HBRUSH hBrush;//Будет создана кисть
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  char szClassName[]="MainWindow";
  char szTitle[]="Программа 5-3";
  MSG Msg;
  WNDCLASS wc;
  ZeroMemory(&wc,sizeof(wc));
  wc.lpfnWndProc=WndProc;
  wc.hInstance=hInst;
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
  wc.lpszClassName=szClassName;
  RegisterClass(&wc);
  HWND hwnd=CreateWindow(szClassName,szTitle,WS_OVERLAPPEDWINDOW,
          0,0,XMAX,YMAX,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_SHOWNORMAL);
  while(GetMessage(&Msg,NULL,0,0))
    DispatchMessage(&Msg);
  return 0;
  }
/*Оконная процедура WndProc*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_CREATE,OnCreate);
```

Необходимо иметь в виду, что структура TEXTMETRIC носит чисто информационный характер и используется только для определения, но не для задания характеристик шрифта. Операция придания шрифту требуемых характеристик (гарнитуры, размера и др.) называется *созданием* шрифта и выполняется с помощью другой структуры – LOGFONT, которая допускает изменение ее элементов. Техника создания шрифтов будет рассмотрена в следующем разделе.

Логические шрифты

Разрабатывая изобразительные детали приложения, естественно воспользоваться богатыми возможностями, предоставляемыми системой Windows в части шрифтового оформления документов. Для того чтобы создать красивый и наглядный документ или экранный кадр, приходится использовать шрифты с различным начертанием, оформлением (курсив, жирный, подчеркнутый), размером и пространственной ориентацией. В зависимости от принципа хранения в памяти компьютера формы символов различают растровые (точечные) и масштабируемые шрифты, которые еще называют шрифтами TrueType. Достоинство шрифтов TrueType заключается в том, что они позволяют изменять в широких пределах размер и другие характеристики символов (например, ширину букв) без снижения качества изображения, что и обусловило их широкое применение.

Операционная система Windows поставляется с базовым набором растровых и масштабируемых шрифтов с разнообразными начертаниями и характеристиками. Среди них имеются как шрифты с равной шириной всех символов (они традиционно используются, например, в исходных текстах программ), так и более приятные для глаза пропорциональные шрифты, у которых ширина символа зависит от его формы (буква Ш, например, шире символа 1). Многие прикладные программы при их установке расширяют базовый набор Windows, добавляя свои шрифты; наконец, можно приобрести и установить в системе наборы шрифтов независимых разработчиков.

Работая с какой-либо коммерческой прикладной программой, использующей шрифты, например с текстовым редактором Word или графическим редактором CorelDraw, пользователь может выбирать для оформления документа любые шрифты из установленных в системе, назначая им требуемые характеристики (размер, интервал между символами и т. д.) с помощью средств используемой прикладной программы. Сложнее обстоит дело при разработке собственного приложения Windows. На экран можно вывести только тот шрифт, дескриптор которого загружен в контекст устройства; при необходимости изменить характеристики шрифта надо сначала *создать* новый шрифт, хотя под этим обманчивым термином понимается не разработка собственного шрифта, а выбор одного из шрифтов, установленных в системе, и придание ему требуемых характеристик.

Правда, в системе имеется несколько готовых шрифтов, дескрипторы которых можно получить со склада с помощью макроса SelectFont() или обобщенной функции SelectObject(). Дескриптор одного из них по умолчанию загружается в контекст устройства при его создании, и именно этим шрифтом мы пользовались в предыдущих примерах для вывода в окно текстовых строк. Однако возможности таких шрифтов ограничены, так как они не допускают изменения своих характеристик (кроме цвета).

Программа, создающая и использующая несколько логических шрифтов

В приводимой ниже программе 5-3 рассматривается методика создания шрифтов и использование их для формирования наглядного и выразительного документа. Вывод программы приведен на рис. 5.10.

Однако у нас фон окна цветной (бледно-желтый). Поскольку по умолчанию фон под буквами белый, строки будут иметь неприятный вид (рис. 5.9).

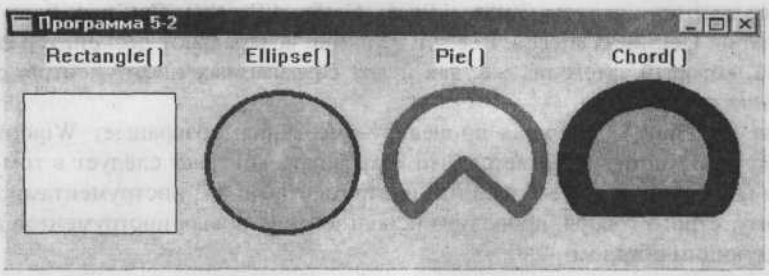


Рис. 5.9. Вывод текста по умолчанию в окно с серым фоном

Каким образом вывести текст на цветной фон? Это можно сделать двумя способами. Первый – с помощью функции SetBkColor() изменить цвет фона под символами. Однако для этого надо знать значения всех трех компонентов цвета фона окна. Другой, более удобный способ – задать режим прозрачности знакомест шрифта. Этот режим устанавливается с помощью функции SetBkMode() с параметром TRANSPARENT:

```
SetBkMode(hdc,TRANSPARENT);
```

Таким образом, в Windows имеются 3 кисти – одна для закрашивания фона окна (ее цвет загружается в структурную переменную типа WNDCLASS при регистрации класса окна), вторая – для фона знакомест под символами шрифта (ее цвет хранится в контексте устройства и по умолчанию он белый) и, как видно из рис. 5.9, еще одна кисть, которая определяет цвет закрашивания геометрических фигур. Последняя кисть по умолчанию тоже белая.

Теперь коснемся вопроса о расположении строк текста в поле окна. В простейшем случае вывод текста осуществляется с помощью уже использованной нами функции TextOut(). Функция позволяет задать графические координаты выводимой строки, причем координаты, указываемые в качестве параметров функции, относятся к верхнему левому углу первого знакоместа строки. Поскольку при выводе строк указываются *графические* координаты, возникает некоторая сложность при выводе на экран нескольких строк, поскольку смещение каждой следующей строки должно быть не меньше высоты предыдущей, а высота строки зависит от вида используемого шрифта. Поэтому в отличие от программ DOS, где можно указывать *текстовые* координаты строк, в программах для Windows при выводе текста приходится определять характеристики действующего в настоящий момент шрифта.

Для определения характеристик (или, как говорят, *метрик*) текущего шрифта (дескриптор которого находится в контексте устройства) используется структура TEXTMETRIC. В нее входят 20 элементов, определяющих различные характеристики шрифта, в частности флаги насыщенности, курсива, подчеркивания и перечеркивания, условный код гарнитуры, средняя и максимальная ширина символов шрифта и другие. Для определения межстрочного интервала надо воспользоваться элементом структуры TEXTMETRIC: tmHeight, который характеризует полную высоту символов шрифта. Для того чтобы строки не соприкасались друг с другом, межстрочный интервал можно устанавить несколько больше величины tmHeight.

Для получения характеристик текущего шрифта в GDI предусмотрена функция GetTextMetrics(), в качестве первого параметра которой передается дескриптор контекста устройства, а в качестве второго – адрес структурной переменной типа TEXTMETRIC.

нится очень ограниченный набор *встроенных* инструментов, которые, соответственно, не надо создавать, а достаточно получить со "склада". Для получения встроенных инструментов используются макросы GetStockPen(), GetStockBrush(), GetStock Font() или обобщенная функция GetStockObject(). Все эти функции возвращают дескриптор встроенного инструмента, который затем так же, как и для создаваемых инструментов, следует выбрать в контекст устройства.

Функция EndPaint(), завершая процедуру рисования, возвращает Windows занятый у нее контекст устройства. Считается, что возвращать контекст следует в том виде, в каком мы его получили, т. е. не с нашими инструментами, а с инструментами по умолчанию. Поэтому, строго говоря, процедура использования новых инструментов должна выглядеть следующим образом:

```
/*Сохраним исходные инструменты, выберем в контекст новые*/
HPEN hOldPen=SelectPen(hdc,hRedPen);
HBRUSH hOldBrush=SelectBrush(hdc,hRedBrush);
...//Рисуем новыми инструментами
/*Загрузим в контекст инструменты другого цвета
  уже без сохранения предыдущих*/
SelectPen(hdc,hBluePen);
SelectBrush(hdc,hYellowBrush);
...//Рисуем этими новыми инструментами
/*Восстановим в контексте исходные инструменты*/
SelectPen(hdc,hOldPen)
SelectPen(hdc,hOldBrush)
/*Освободим контекст устройства перед завершением функции OnPaint()*/
EndPaint(hwnd,&ps);
```

Реально, однако, современные системы Windows сами восстанавливают состояние своих объектов после их использования в программе, и пренебрежение описанной выше процедурой в большинстве случаев не приведет к неприятностям.

Такое же замечание можно сделать относительно создания новых инструментов. Вообще говоря, все созданные пользователем объекты (кисти, перья, шрифты, таймеры и пр.) необходимо перед завершением программы удалить, чтобы они не загромождали память, что обычно выполняется в функции OnDestroy() обработки сообщения WM_DESTROY. Однако система Windows при завершении приложения сама освобождает всю занятую ею память, и эти действия во многих случаях оказываются избыточными.

Особенности вывода текстовых строк

При выводе в окно приложения текстовых строк приходится решать две задачи: установку цвета шрифта и определение координат выводимых строк.

При использовании шрифта по умолчанию (т. е. не прибегая к процедуре создания нового шрифта) можно изменить только те его характеристики, которые непосредственно записываются в контекст устройства. К ним относятся цвет символов и фона под ними, режим фона знакомест (прозрачный или непрозрачный) и некоторые другие. По умолчанию для символов задан черный цвет, для фона – белый. Цвет символов изменяется с помощью функции SetTextColor(), цвет фона под символами – функцией SetBkColor(), режим прозрачности символов – функцией SetBkMode(). Поскольку эти характеристики хранятся в контексте устройства, настройка цвета будет действовать до следующего изменения или до закрытия контекста.

В программе 5-2 с помощью функции SetTextColor() устанавливается синий цвет символов. Если бы текст выводился на белое окно, то никаких проблем не возникло бы.

При необходимости с помощью функции CreateHatchBrush() можно создать штриховую кисть заданного цвета. В качестве первого параметра этой функции указывается тип штриховки (диагональная, вертикальная и пр.), в качестве второго – цвет.

В GDI предусмотрена также функция CreatePatternBrush(), позволяющая создавать кисти с произвольным рисунком, предварительно созданным в виде растрового изображения.

Функции создания кисти или пера, как и ряд других функций Windows, устанавливающих цвет каких-либо элементов графики, требуют в качестве второго параметра характеристику цвета в виде двухсловного данного типа COLORREF. Что такое тип COLORREF? В интерактивном справочнике можно найти только, что этот тип описывает 32-битовое данное. Чтобы узнать, как оно образуется, надо обратиться к файлу WINGDI.H, где определен макрос RGB(r,g,b), служащий для формирования 32-битового кода цвета по заданным составляющим (красной, зеленой и синей):

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

Даже не вдаваясь в детали этого довольно запутанного определения, можно сообразить, что данное типа COLORREF складывается из трех байтовых составляющих, причем синяя составляющая (параметр b) перед тем, как попасть в результирующее данное, сдвигается на 16 бит влево, занимая после этого байт с номером 2, зеленая составляющая (параметр g) сдвигается на 8 бит влево, занимая байт с номером 1, а красная составляющая (параметр r) остается без изменений, поступая, таким образом, в байт с номером 0. Старший байт (байт 3) результирующего данного не используется (рис. 5.8).

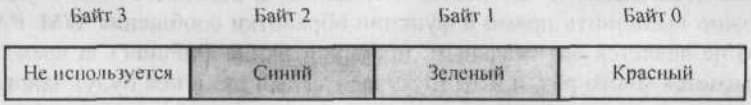


Рис. 5.8. Составление кода цвета из трех составляющих

Значения компонентов указываются в макросе RGB виде десятичных или шестнадцатеричных чисел в диапазоне от 0 до 255 каждое. Например, RGB(255,255,255) описывает белый цвет, а RGB(0,0,0) – черный.

Поскольку интенсивность каждой составляющей цвета, хранящейся в байте, может принимать любое из 256 значений, такая конструкция позволяет в принципе задавать до 256^3 = 16 М = 16 777 216 цветовых оттенков.

Если видеосистема компьютера настроена на отображение лишь 16 цветов, то значения компонентов могут составлять только 128 (половинная яркость), 255 (полная яркость компонента) или 0, причем допускается использовать любые комбинации либо половинной яркости, либо полной, но не их смесь. Результирующий цвет при этом определяется без труда:

```
RGB(128,128,0) - красно-зеленый, т. е. коричневый
RGB(255,255,0) - яркий красно-зеленый, т. е. желтый
RGB(0,128,128) - сине-зеленый, т. е. бирюзовый
```

Обычно же при работе на современном компьютере можно задавать любые другие сочетания цветовых компонентов:

```
RGB(220,220,220) - очень светло-серый
RGB(220,220,255) - очень светло-серый плюс синий, т. е. очень бледно-синий
RGB(200,200,0) - светло-коричневый
```

Как уже вскользь упоминалось в гл. 3, для облегчения работы с часто используемыми инструментами в Windows предусмотрен специальный "склад" (stock), на котором хра-

- загрузка (*выбор*) в контекст устройства дескриптора созданного инструмента с возможным сохранением дескриптора аналогичного инструмента по умолчанию; это действие выполняется с помощью функций SelectPen() или SelectBrush();
- рисование новым инструментом;
- уничтожение созданных инструментов функцией DeleteObject() или макросами DeletePen() и DeleteBrush().

Создание нового инструмента можно выполнить в любом месте программы; наиболее естественно это сделать при обработке сообщения WM_CREATE для главного окна или того окна, для которого готовятся новые инструменты. Поскольку в этом случае дескрипторы инструментов определяются в одной функции, используются в другой, а уничтожаются в третьей, их следует объявить глобальными.

Функции выбора и рисования требуют в качестве первого параметра указания дескриптора контекста устройства, поэтому эти функции обычно вызываются в процессе обработки сообщения WM_PAINT после получения контекста устройства функцией BeginPaint(). В контексте устройства хранится только один дескриптор для каждого инструмента; если требуется сначала рисовать, скажем, синим цветом, а затем зеленым, то при переходе к зеленому цвету в контекст устройства следует загрузить дескриптор зеленого инструмента.

Удаление созданных инструментов можно выполнить в функции обработки сообщения WM_DESTROY; можно, конечно, и динамически создавать и удалять инструменты по ходу программы, особенно если инструментов много, а используются они не вперемежку, а последовательно. В частности, и создание, и использование, и удаление инструментов можно выполнить прямо в функции обработки сообщения WM_PAINT(), хотя такой способ не является оптимальным, так как функция OnPaint() за время жизни программы вызывается много раз, и в этом случае каждый раз в ней будут заново создаваться инструменты.

Остановимся на некоторых деталях создания и использования графических инструментов. Из прототипа функции CreatePen()

```
HPEN CreatePen(int fnPenStyle,int nWidth, COLORREF clrref)
```

видно, что она требует передачи ей трех параметров. Первый параметр (табл. 5.1) определяет тип линии, которая будет проводиться этим пером. Второй параметр задает толщину пера в пикселах, а третий – его цвет. Функция возвращает дескриптор созданного пера, который затем следует загрузить в контекст устройства.

Таблица 5.1. Значения первого параметра функции CreatePen()

Значение	*Результат*
PS_SOLID	Сплошная линия
PS_DASH	Пунктирная линия (только для линий толщиной 1)
PS_DASHDOT	Линия из точек (только для линий толщиной 1)
PS_DASHDOTDOT	Штрихпунктирная линия (только для линий толщиной 1)
PS_NULL	Нуль-перо (невидимо; удобно для заливки фигур без контура)
PS_INSIDEFRAME	Для замкнутых фигур ограничивающие линии проводятся внутри заданных координат фигуры

Функция CreateSolidBrush() позволяет создать однотонную цветную кисть заданного цвета. Так же как и функция CreatePen(), она возвращает дескриптор созданной кисти.

```
  SetTextColor(hdc,RGB(0,0,200));//Синий цвет текста
/*Выведем четыре строки текста*/
  TextOut(hdc,25,5,szText1,strlen(szText1));
  TextOut(hdc,155,5,szText2,strlen(szText2));
  TextOut(hdc,280,5,szText3,strlen(szText3));
  TextOut(hdc,395,5,szText4,strlen(szText4));
/*Выведем четыре фигуры*/
  Rectangle(hdc,10,40,110,140);//Квадрат инструментами по умолчанию
  SelectPen(hdc,hRedPen);//Выберем в контекст новое перо
  SelectBrush(hdc,hRedBrush);//и новую кисть
  Ellipse(hdc,130,40,230,140);//Круг новыми инструментами
  SelectPen(hdc,hGreenPen);//Выберем новое перо
  SelectBrush(hdc,hGreenBrush);//и новую кисть
  Pie(hdc,250,40,350,140,350,140,250,140);//Сектор опять новыми инструментами
  SelectPen(hdc,hBluePen);//Выберем новое перо
  SelectBrush(hdc,hBlueBrush);//и новую кисть
  Chord(hdc,370,40,470,140,470,125,370,125);//Сегмент
  EndPaint(hwnd,&ps);//Освободим контекст устройства
  }
/*Функция OnDestroy обработки сообщения WM_DESTROY*/
void OnDestroy(HWND){
  PostQuitMessage(0);//Завершим программу
  }
```

По своей структуре программа 5-2 мало отличается от предыдущей. Из текста оконной функции WndProc() видно, что в программе обрабатываются 3 сообщения Windows: WM_CREATE, WM_PAINT и WM_DESTROY. Эти сообщения представляют минимальный набор для любого приложения Windows. Сообщение WM_CREATE посылается в окно после его создания, но еще до вывода на экран. В настоящей программе обработка сообщения WM_CREATE используется для создания дополнительных графических инструментов – перьев и кистей.

Функция обработки сообщения WM_DESTROY также используется стандартным образом – в ней вызывается функция PostQuitMessage(), что приводит к разрыву цикла обнаружения сообщений и завершению программы.

Наконец, в функции обработки сообщения WM_PAINT, как это и положено, осуществляется вывод в окно требуемых графических объектов – в данном случае нескольких строк текста и простых геометрических фигур.

Процедуры работы с графическими инструментами

В настоящем подразделе мы рассмотрим два наиболее употребительных графических инструмента – перо и кисть. Характеристики пера, находящегося в контексте устройства, определяют вид линий, которыми рисуются геометрические фигуры. Характеристики кисти определяют цвет и фактуру внутренних областей замкнутых фигур. Поскольку в контексте хранятся не *характеристики* этих инструментов, а их *дескрипторы*, для изменения характеристик пера или кисти надо создать *новые* инструменты и поместить их дескрипторы в контекст.

При работе с новыми инструментами следует соблюдать определенную стандартную последовательность действий:

- создание нового инструмента с заданными характеристиками с помощью, например, функций Windows CreatePen() или CreateSolidBrush() (предусмотрены и другие функции, например CreateHatchBrush() для создания штриховой кисти) и получение его дескриптора;

```
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
/*Глобальные переменные, доступные всем функциям*/
HPEN hRedPen,hGreenPen,hBluePen;//Дескрипторы новых перьев
HBRUSH hRedBrush,hGreenBrush, hBlueBrush;//и новых кистей
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  char szClassName[]="MainWindow";
  char szTitle[]="Программа 5-2";
  MSG msg;
  WNDCLASS wc;
/*Зарегистрируем класс главного окна*/
  ZeroMemory(&wc,sizeof(wc));
  wc.lpfnWndProc=WndProc;
  wc.hInstance=hInst;
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hbrBackground=CreateSolidBrush(RGB(200,200,100));//Окно бледно-желтое
  wc.lpszClassName=szClassName;
  RegisterClass(&wc);
/*Создадим главное окно и сделаем его видимым*/
  HWND hwnd=CreateWindow(szClassName,szTitle,
        WS_OVERLAPPEDWINDOW,20,20,500,190,
        HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_SHOWNORMAL);
/*Организуем цикл обнаружения сообщений*/
while(GetMessage(&msg,NULL,0,0))
    DispatchMessage(&msg);
  return 0;
  }
/*Оконная функция WndProc главного окна*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,
        WPARAM wParam,LPARAM lParam){
  switch(msg)  {
    HANDLE_MSG(hwnd,WM_CREATE,OnCreate);
    HANDLE_MSG(hwnd,WM_PAINT,OnPaint);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
/*Функция OnCreate обработки сообщений WM_CREATE*/
BOOL OnCreate(HWND,LPCREATESTRUCT){
/*Создадим новые перья*/
  hRedPen=CreatePen(PS_SOLID,4,RGB(150,0,0));
  hGreenPen=CreatePen(PS_SOLID,10,RGB(0,150,0));
  hBluePen=CreatePen(PS_SOLID,20,RGB(0,0,150));
/*Создадим новые кисти*/
  hRedBrush=CreateSolidBrush(RGB(255,120,120));
  hGreenBrush=CreateSolidBrush(RGB(120,255,120));
  hBlueBrush=CreateSolidBrush(RGB(120,120,255));
  return TRUE;
  }
/*Функция OnPaint обработки сообщений WM_PAINT*/
void OnPaint(HWND hwnd){
  PAINTSTRUCT ps;//Структура для функции BeginPaint()
  char szText1[]="Rectangle( )"; //Просто
  char szText2[]="Ellipse( )";   //надписи
  char szText3[]="Pie( )";       //над
  char szText4[]="Chord( )";     //фигурами
  HDC hdc=BeginPaint(hwnd,&ps);  //Получим контекст устройства
  SetBkMode (hdc,TRANSPARENT);   //Прозрачный фон под текстом
```

Обратите внимание на второе значение этой последовательности – y-координату верхнего края закрывающего окна. Число 6 говорит о том, что границы области вырезки определяются не относительно границ всего окна, а относительно границ его рабочей области. Расстояние между нижним краем строки заголовка и верхним краем окна даты/времени как раз и составляет 6 пикселов.

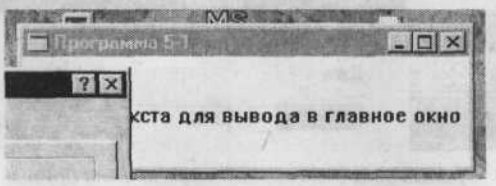


Рис. 5.6. Левый угол окна приложения закрыт другим окном

Итак, функция BeginPaint() каждый раз сообщает программе координаты области вырезки. То, что мы не используем эти данные, каждый раз заново перерисовывая *все* окно, – это уже наше дело. Однако сама Windows, как оказывается, работает более разумно, чем мы. Несмотря на то что мы в ответ на сообщение WM_PAINT полностью восстанавливаем изображение на всем экране (в нашем примере посылаем строку, которая практически простирается от левого края окна до правого), функции GDI реально перерисовывают только ту часть изображения, которая попала в область вырезки. Неповрежденная часть окна физически не перерисовывается, что повышает скорость вывода и уменьшает неприятное мерцание экрана.

Таким образом, хотя программа обычно не использует информации об области вырезки, интерфейс GDI руководствуется ею с целью ускорения работы системы.

Вывод текстовых строк и простых геометрических фигур

Рассмотрим программу, которая выводит в главное окно несколько строк текста и простые геометрические фигуры – прямоугольники и круги (приложение 5-2). Результат работы программы изображен на рис. 5.7.

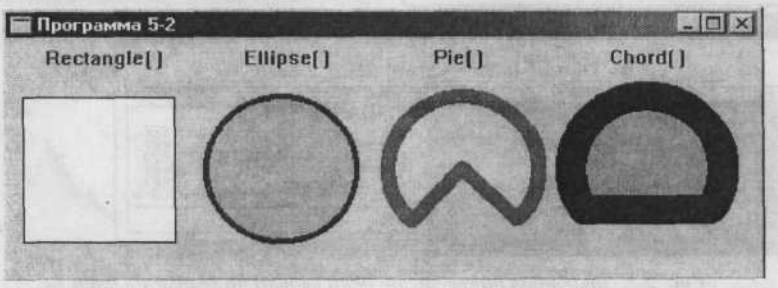


Рис. 5.7. Вывод текстовых строк и геометрических фигур

```
/*Программа 5-2. Вывод текста и простых геометрических фигур*/
/*Операторы препроцессора*/
#include <windows.h>
#include <windowsx.h>
/*Прототипы используемых в программе функций пользователя*/
BOOL OnCreate(HWND,LPCREATESTRUCT);
void OnPaint(HWND);
void OnDestroy(HWND);
```

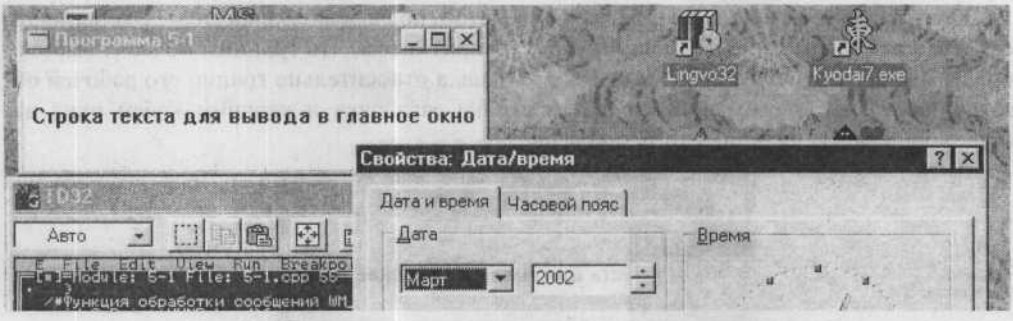


Рис. 5.4. Правый угол окна приложения закрыт другим окном

Теперь выведите наше приложение на передний план, щелкнув по нему мышью. Так как в этом случае закрытый ранее угол нашего окна должен перерисоваться (ведь Windows никак не может угадать, что у нас там ничего не нарисовано; хотя, если вдуматься, там нарисован фон, который мог бы быть к тому же не белым, а цветным), в окно нашего приложения посылается сообщение WM_PAINT. Функция BeginPaint(), вызываемая в ходе обработки этого сообщения, передает в программу координаты уже не всего окна, а именно области вырезки:

```
ps.rcPaint={208,54,292,73}
```

Как только закрытая ранее часть окна нашего приложения снова появится на экране, в окно поступит сообщение WM_PAINT и в кадре отладчика можно будет увидеть значения границ области вырезки, которые, разумеется, будут зависеть от степени перекрытия окон (рис. 5.5).

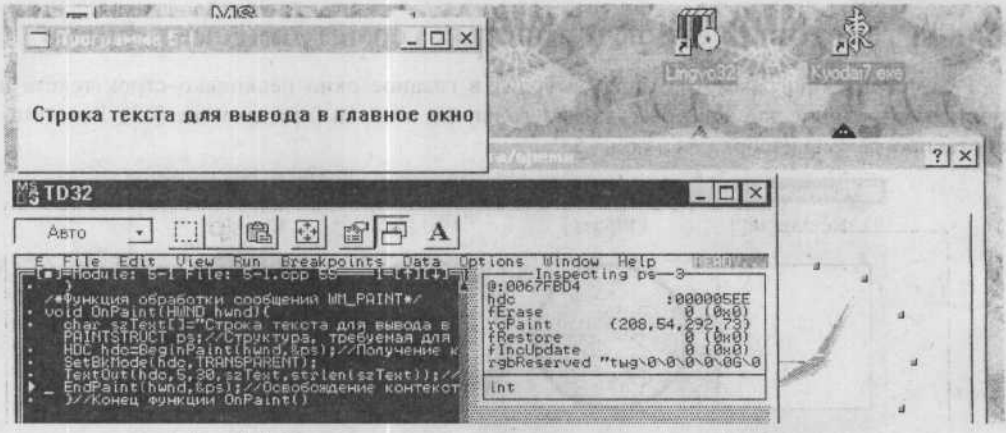


Рис. 5.5. Закрытая ранее часть окна вышла на передний план

Легко сообразить, что указанные координаты в точности соответствуют ранее закрытому уголку окна.

Повторим этот эксперимент, закрыв окно нашего приложения с левой стороны (рис. 5.6).

В этом случае после снятия закрывающего окна мы получим следующие значения границ области вырезки:

```
ps.rcPaint={0,6,71,73}
```

сле выполнения всех инициализирующих действий, но перед циклом обработки сообщений. Можно было поступить и по-другому, например вызывать функцию Draw() по нажатии клавиши мыши или в ответ на любое другое событие.

В функции OnCreate() выполняются все подготовительные действия для работы с совместимой памятью. Прежде всего создается сама совместимая память (лучше сказать – выделяется область памяти для использования ее в качестве совместимой) с помощью уже упоминавшейся функции CreateCompatibleBitmap(). Эта функция требует указания размеров выделяемой памяти; мы указываем размеры нашего окна, хотя можно было, имея в виду возможность растяжения окна, выделить память размером в весь экран. Таким образом, в настоящей программе использован другой метод выделения совместимой памяти – не с помощью загрузки готового изображения, как в программе 9-1, а резервированием требуемого объема пустой пока памяти функцией CreateCompatibleBitmap(). Для работы с этой областью необходим совместимый контекст памяти, а для его создания, в свою очередь, требуется контекст устройства (окна). Поскольку контекст устройства потребовался нам не в функции OnPaint(), его надо запросить с помощью функции GetDC(). Далее с помощью функции CreateCompatibleDC() создается совместимый контекст памяти, а макрос SelectBitmap() позволяет выбрать дескриптор совместимой памяти в совместимый контекст.

Выделенная системой память может быть заполнена "мусором", и ее надо очистить (некоторые системы Windows в такой ситуации очищают память, однако рассчитывать на это не следует). Ранее выполняли закрашивание совместимой памяти функцией FillRect(); в данном случае мы воспользовались функцией PatBlt(), которая, как и FillRect(), позволяет закрасить "устройство" (в данном случае – совместимую память) выбранной в контекст этого устройства кистью или просто белым цветом (как в нашем примере). Перед завершением функции OnCreate() необходимо освободить занятый у системы контекст окна hdc; что же касается контекста памяти hdcMem, то он нам будет нужен все время жизни приложения, так как именно через него будет осуществляться копирование изображения из памяти в окно. Поэтому контекст памяти освобождается только в функции OnDestroy().

Функция Draw() почти повторяет функцию OnPaint() программы 9-2. В ней осуществляется вычисление в цикле координат кривой и вывод точек синего цвета в память. Принципиальное отличие заключается в том, что вывод точек функцией SetPixel() выполняется не через обычный контекст устройства, т. е. контекст окна, а через контекст совместимой памяти, что и обеспечивает ее заполнение требуемым изображением.

Для того чтобы повысить наглядность работы приложения, в функцию Draw() включены дополнительные строки параллельного вывода изображения непосредственно в окно. Для этого в начале функции мы получаем контекст окна (разумеется, с помощью функции GetDC(), а не BeginPaint()), через этот контекст в окно выводятся те же точки (для наглядности красным цветом), а после завершения цикла контекст окна освобождается. В результате после запуска приложения, пока выполняется функция Draw(), мы будем наблюдать медленное рисование в окне красной кривой и понимать, что параллельно та же кривая рисуется в совместимой памяти синим цветом. По завершении функций OnCreate() и Draw() приложение получит сообщение WM_PAINT, которое было помещено в очередь функцией ShowWindow() и "висело" в очереди все это время, поскольку программа еще не дошла до цикла обработки сообщений; в результате изображение из совместимой памяти практически мгновенно будет скопировано в окно и станет синим.

Теперь понаблюдайте реакцию программы на изменение размеров окна. Любое увеличение размеров окна приведет к посылке в приложение сообщения WM_PAINT и к практически *мгновенному* копированию изображения из совместимой памяти в окно.

Глава 10
Стандартные элементы управления

В предыдущих главах мы познакомились с некоторыми элементами управления, которые обычно используются в составе диалоговых окон, однако могут выступать и элементами главного или вложенных окон приложения. В наших примерах встречались:

- нажимаемые кнопки;
- альтернативные кнопки;
- списки;
- комбинированные списки;
- статические элементы управления – тексты;
- групповые рамки;
- линейки прокрутки;
- линейки с ползунками;
- окна редактирования.

В действительности состав элементов управления шире. В частности, для 32-разрядных приложений разработана группа элементов управления с современным интерфейсом, объединяемых странным названием "стандартные элементы управления" (Common controls).

В настоящей главе будут рассмотрены примеры приложений с некоторыми из элементов управления, входящих в эту группу.

Графический список

Графический список (list view) является многофункциональным элементом управления, широко используемым, в частности, для вывода на экран содержимого каталогов. Графическим он называется потому, что позволяет включать в его состав не только текстовые строки, но и значки (пиктограммы). На рис. 10.1 изображена одна из форм графического списка (в виде таблицы), где значки присутствуют только в самом левом, основном столбце, а остальные, дополнительные столбцы заполнены текстовыми строками.

Характерной чертой графического списка является наличие в его верхней части заголовка списка – самостоятельного стандартного элемента управления, который можно использовать и отдельно от графического списка, в сочетании с другими элементами управления. Заголовок списка содержит отдельные поля, совпадающие со столбцами списка. Размер полей заголовка и, соответственно, столбцов списка можно изменять, перетаскивая мышью вертикальные линейки, разделяющие поля заголовка. Щелчок мышью по основному (самому левому) элементу списка формирует сообщение Windows, обрабатывая которое можно выполнить любые запланированные действия над этим элементом. Графический список имеет встроенные средства динамического изменения его вида (таблица, список, мелкие значки, крупные значки), что делает его чрезвычайно удобным для просмотра содержимого дисков и каталогов (папок) и обслуживания файловой системы.

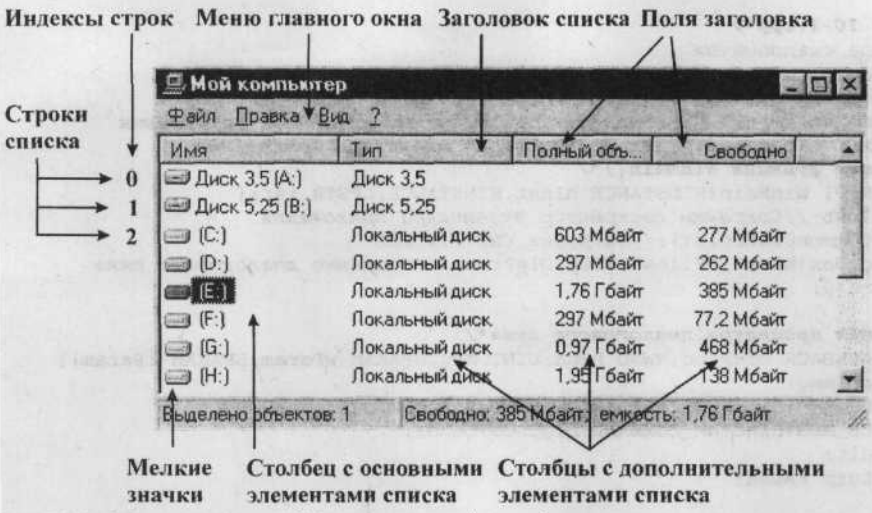


Рис. 10.1. Кадр программы "Мой компьютер" с графическим списком в главном окне

Программное формирование графического списка

Мы рассмотрим здесь формирование графического списка в его простейшей (и возможно, наиболее важной для прикладных программ) форме, включающей заголовок списка с разделяющими полями вместе с соответствующим числом столбцов данных – текстовых строк.

Приводимая ниже программа 10-1 представляет собой модификацию программы 6-2, в которой в диалоговое окно (конкретно – в элемент управления *список*) выводился перечень файлов, удовлетворяющих заданному в программе шаблону. В программе 10-1 перечень файлов сопровождается их размерами, и вся эта информация выводится не в простой список (в котором, кстати, трудно разместить разнородную информацию), а в графический (рис. 10.2).

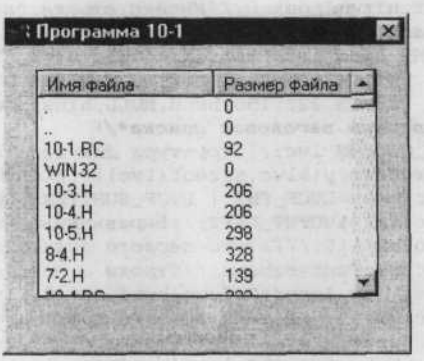


Рис. 10.2. Графический список с составом каталога

```
/*Программа 10-1. Графический список*/
/*Файл 10-1.h*/
BOOL CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM);
void DlgOnCommand(HWND,int,HWND,UINT);
BOOL DlgOnInitDialog(HWND,HWND,LPARAM);
```

```
/*Файл 10-1.rc*/
Files DIALOG 40, 20, 130, 100
CAPTION "Программа 10-1"
  BEGIN
  END
```

```
/*Файл 10-1.cpp*/
#include <windows.h>
#include <windowsx.h>
#include "10-1.h"
#include <commctrl.h>//Поддержка стандартных элементов управления
HINSTANCE hInstance;//Для дескриптора экземпляра приложения
/*Главная функция WinMain()*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  hI=hInst;//Сохраним дескриптор экземпляра приложения
  InitCommonControls();//Загрузка CMCTL32.DLL
  DialogBox(hInst,"Files",NULL,DlgProc);//Создание диалогового окна
  return 0;
  }
/*Оконная процедура диалогового окна*/
BOOL CALLBACK DlgProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
  HANDLE_MSG(hwnd,WM_INITDIALOG,DlgOnInitDialog);
  HANDLE_MSG(hwnd,WM_COMMAND,DlgOnCommand);
  default:
    return FALSE;
    }
  }
/*Функция обработки сообщения об инициализации диалога*/
BOOL DlgOnInitDialog(HWND hwnd,HWND,LPARAM){
  char szName[]="Имя файла";//Текст первого поля заголовка
  char szSize[]="Размер файла";//Текст второго поля заголовка
  char szFileName[80];//Символьная строка для имен файлов
  char szFileSize[10];//Символьная строка для размеров файлов
  HANDLE hFile,hOpenFile;//Для дескрипторов файлов
  DWORD dwFileSize;//Целочисленная переменная для размера файлов
  int nItemIndex=0;//Индекс строки списка
/*Создадим графический список*/
  HWND hwndList=CreateWindowEx(0,WC_LISTVIEW,NULL,//Класс графического списка
      WS_CHILD|WS_BORDER|WS_VISIBLE|LVS_REPORT,//Стиль: таблица (отчет)
      20,15,222,150,hwnd,NULL,hInstance,NULL);
/*Заполним заголовок списка*/
  LV_COLUMN lvc;//Структура для заголовка списка
  ZeroMemory(&lvc,sizeof(lvc));//Обнуляем стрктуру
  lvc.mask=LVCF_FMT | LVCF_SUBITEM | LVCF_TEXT | LVCF_WIDTH;//Маска
  lvc.fmt=LVCFMT_LEFT;//Выравнивание влево
  lvc.cx=115;//Размер первого поля заголовка
  lvc.pszText=szName;//Строка с текстом первого поля заголовка
  ListView_InsertColumn(hwndList,0,&lvc);//Включаем поле в заголовок
  lvc.cx=90;//Размер второго поля заголовка
  lvc.pszText=szSize;//Строка с текстом второго поля заголовка
  ListView_InsertColumn(hwndList,1,&lvc);//Включаем поле в заголовок
/*Подготавливаем структуру для заполнения списка*/
  LV_ITEM lvi;//Структура для элементов списка
  ZeroMemory(&lvi,sizeof(lvi));//Обнуляем структуру
  lvi.mask=LVIF_TEXT;//Маска
  lvi.pszText=szFileName;//Строка с текстом элемента
/*Прочитаем и выведем в диалог список файлов*/
  WIN32_FIND_DATA fd;//Структура для поиска файлов
  hFile=FindFirstFile("*.*",&fd);//Шаблон поиска
  if(hFile==INVALID_HANDLE_VALUE)//Если нет файлов
    return TRUE;
  else{
    hOpenFile=CreateFile(fd.cFileName,GENERIC_READ,0,0,OPEN_EXISTING,0,NULL);
    if(hOpenFile==INVALID_HANDLE_VALUE)//Если нельзя открыть
      dwFileSize=0;//Пусть размер в этом случае будет = 0
    else{//Если файл открылся
```

```
      dwFileSize=GetFileSize(hOpenFile,NULL);//Найдем его длину
      CloseHandle(hOpenFile);//Закроем файл, освободим его дескриптор
      }
    wsprintf(szFileSize,"%d",dwFileSize);//Преобразуем длину в символы
    strcpy(szFileName,fd.cFileName);//Получим имя файла
    lvi.iItem=nItemIndex;//Текущий индекс элемента списка (=0)
    ListView_InsertItem(hwndList,&lvi);//Включим имя файла в список
    ListView_SetItemText(hwndList,nItemIndex,1,szFileSize;//Включим длину
    nItemIndex++;//К следующему элементу
    }
  while(FindNextFile(hFile,&fd)){//Пока есть файлы
    hOpenFile=CreateFile(fd.cFileName,GENERIC_READ,0,0,OPEN_EXISTING,0,NULL);
    if(hOpenFile==INVALID_HANDLE_VALUE)//Если нельзя открыть
      dwFileSize=0;//Пусть в этом случае размер будет = 0
    else{//Если файл открылся
      dwFileSize=GetFileSize(hOpenFile,NULL);//Найдем его длину
      CloseHandle(hOpenFile);//Закроем файл, освободим его дескриптор
      }
    wsprintf(szFileSize,"%d",dwFileSize);//Преобразуем длину в символы
    strcpy(szFileName,fd.cFileName);//Получим имя файла
    lvi.iItem=nItemIndex;//Текущий индекс элемента списка
    ListView_InsertItem(hwndList,&lvi);//Включим имя файла в список
    ListView_SetItemText(hwndList,nItemIndex,1,szFileSize);//Включим длину
    nItemIndex++;//К следующему элементу
    }
  return TRUE;
  }
/*Функция обработки сообщений от элементов управления*/
void DlgOnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id){
    case IDCANCEL:
    EndDialog(hwnd,0);
    }
  }
```

Заголовочный файл состоит из одних только прототипов функций – символических констант в этом примере нет.

Файл ресурсов тоже очень прост – в нем описано пустое, без элементов управления, диалоговое окно с заголовком. Обратите внимание на то, что предложение DIALOG, даже при отсутствии состава диалога, требует после себя указания операторных скобок BEGIN-END (которые можно заменить парой фигурных скобок {}).

Использование в программе стандартных элементов управления требует подключения к ней заголовочного файла COMMCTRL.H, в котором собраны определения констант, макросов, структур и прототипов функций, поддерживающих средства взаимодействия со стандартными элементами управления.

Приложение 10-1 построено на базе модального диалога, без главного окна. Поэтому в главной функции приложения WinMain() отсутствует процедура регистрации и создания окна, а также цикл обработки сообщений. После сохранения дескриптора экземпляра приложения в глобальной переменной hInstance (она будет нами использована при создании окна графического списка) вызывается функция InitCommonControls(). Эта функция загружает библиотеку динамической компоновки COMCTL32.DLL, программы которой обеспечивают функционирование стандартных элементов управления. Наконец, вызовом функции DialogBox() активизируются процедуры Windows создания и вывода на рабочий стол модального диалогового окна.

Оконная диалоговая процедура DlgProc() построена обычным образом. С помощью макросов HANDLE_MSG обеспечивается обработка двух сообщений Windows:

WM_INITDIALOG – о создании диалогового окна и WM_COMMAND – от элементов управления диалогом.

Практически весь текст программы сосредоточен в функции DlgOnInitDialog(), вызываемой в процессе инициализации диалогового окна. В начале функции расположен список переменных, используемых далее по ходу программы; их назначение будет понятно из дальнейших объяснений. Первое важное действие – создание стандартного элемента управления "графический список". В отличие от обычных элементов управления графический список создается не с помощью описания его в сценарии диалога в файле .RC, а путем вызова функции CreateWindowEx() с указанием в качестве имени класса создаваемого окна символического имени WC_LISTVIEW. Стиль этого окна помимо обычных констант WS_CHILD, WS_BORDER и WS_VISIBLE включает также константу LVS_REPORT, определяющую вид отображения графического списка – в виде таблицы. Функция CreateWindowEx() возвращает дескриптор созданного окна графического списка, который нам понадобится в дальнейшем.

Следующий этап – формирование заголовка списка, для чего используется структурная переменная lvc типа LV_COLUMN. Она определена в файле COMMCTRL.H и имеет следующий состав:

```
struct LV_COLUMN {
  UINT mask;//Маска, определяющая набор используемых элементов структуры
  int fmt;//Вид выравнивания текста в столбце
  int cx;//Ширина столбца в пикселах
  LPTSTR pszText;//Адрес строки с заголовком столбца
  int cchTextMax;//Размер заголовка
  int iSubItem;//Порядковый номер дополнительного столбца
  }
```

После обнуления всей структурной переменной lvc 32-разрядной функцией ZeroMemory() инициализируются ее элементы. Набор символических констант, определяющих значение элемента lvc.mask, говорит о том, что в структуре будут использоваться фактически все ее элементы: fmt, cx, pszText и iSubItem. В элементе fmt устанавливается тип выравнивания текста (влево), в элементе cx – ширина первого поля заголовка (115 пикселов), а в элементе pszText – имя строки с текстом заголовка для первого поля ("Имя файла"). После этого вызовом макроса ListView_InsertColumn в диалоговое окно посылается сообщение LVM_INSERTCOLUMN о формировании первого поля заголовка. Это сообщение можно было послать в диалог и непосредственно, как это мы делали в примерах гл. 6. В параметрах макроса ListView_InsertColumn указываются дескриптор окна графического списка hwndList, номер поля (0) и адрес структурной переменной lvc.

Далее точно таким же образом формируется второе (и последнее в нашем случае) поле заголовка. Для него устанавливается другой размер и другой текст заголовка. В параметрах макровызова ListView_InsertColumn указывается номер столбца – 1.

Следующий этап – заполнение самого списка содержательной информацией. В этом фрагменте программы сочетаются как формальные действия по занесению данных в столбцы списка, так и действия по добыванию необходимой информации.

Для заполнения списка понадобится структурная переменная lvi типа LV_ITEM, имеющая следующий состав:

```
struct LV_ITEM {
  UINT   mask;//Маска, определяющая набор используемых элементов структуры
  int    iItem;//Номер строки списка
  int    iSubItem;//Номер столбца списка
  UINT   state;//Состояние элемента списка
  UINT   stateMask;//Возможные состояния элемента
```

```
LPTSTR  pszText;//Адрес строки с текстом элемента списка
int     cchTextMax;//Размер строки с текстом элемента
int     iImage;//Индекс значка элемента
LPARAM lParam;//32-битовое значение, связанное с элементом
}
```

После обнуления структурной переменной lvi в ее элемент lvi.mask заносится значение LVIF_TEXT. Эта константа обозначает, что в структурной переменной lvi значащим является только поле с текстом. В элемент lvi.pszText заносится адрес пустой пока строки с текстом элемента.

Дальнейшие несколько строк повторяют соответствующий фрагмент примера 6-2. В соответствии с указанным шаблоном ищется сначала первый удовлетворяющий шаблону файл, а затем в цикле все остальные файлы. Однако обработка найденных файлов осуществляется более сложным образом. Прежде всего найденный файл открывается с помощью функции CreateFile(). Если файл не открылся, функция CreateFile() возвращает значение INVALID_HANDLE_VALUE. Такое может произойти, если найден не файл, а каталог или файл используется системой Windows и, следовательно, обращение к нему заблокировано. В этих случаях определить размер файла нельзя и переменной dwFileSize присваивается нулевое значение.

Если файл открылся успешно, вызовом функции GetFileSize() определяется его длина и полученное значение преобразуется в символьную форму в виде десятичного числа. Результирующая строка заносится в переменную szFileSize. Далее имя найденного файла копируется из элемента структурной переменной fd.cFileName в переменную szFileName, на чем заканчиваются действия по подготовке данных для внесения в графический список.

В элемент структурной переменной lvi.iItem заносится текущий номер строки списка (он хранится и наращивается в переменной nItemIndex), и вызовом макроса ListView_InsertItem в графический список посылается сообщение LVM_INSERTITEM, что приводит к созданию новой строки и помещению в ее основной столбец имени файла. После этого вызовом макроса ListView_SetItemText в дополнительный столбец с номером 1 той же строки помещается длина файла из переменной szFileSize. Заметьте, что макрос ListView_InsertItem использует данные из структурной переменной lvi, в которую мы заранее занесли все необходимые значения. В последовательных шагах цикла просмотра будут только изменяться фактические имена файлов в символьном массиве szFileName, откуда они и переносятся в список. В то же время в параметрах макроса ListView_SetItemText указываются и номер столбца (1 в нашем случае), и адрес строки с данными (szFileSize).

Все описанные действия повторяются в цикле while до исчерпания файлов в просматриваемом каталоге.

Наше стремление максимально упрощать рассматриваемые программы привело в данном случае к несовершенству примера. Программа 10-1 правильно определяет длину файлов только в том каталоге, в котором она сама находится. Действительно, если в шаблоне поиска файлов указать не только спецификацию файлов (например, *.* или *.DAT), но и путь к файлам, то *просматриваться* будет, естественно, тот каталог, который указан в спецификации, но *открываться* будут файлы из текущего каталога, так как в качестве спецификации открываемых файлов для функции CreateFile() берется информация из элемента cFileName структуры WIN32_FIND_DATA, а этот элемент содержит только имя файла, но не путь к нему. В результате будет сделана попытка открыть файл из текущего каталога, в то время как функции поиска просматривают совсем иной ката-

лог. Для того чтобы программа работала во всех случаях правильно, следует более сложным образом формировать спецификацию файла для функции CreateFile().

Графический список и уведомляющие сообщения

В рассмотренной выше программе графический список выполнял пассивную роль – он просто отображал перечень файлов. Дополним программу средствами взаимодействия пользователя со списком. Самое естественное ввести в программу отклик на выбор пользователем того или иного элемента списка. Пусть при щелчке мышью по имени файла этот файл сразу копируется на дискету. Мы получим специализированную программу, позволяющую просматривать файлы текущего каталога и сохранять на дискете нужные. С таким же успехом можно было предусмотреть любую другую операцию с файлами, например их удаление.

Если Windows фиксирует взаимодействие пользователя с графическим списком, например одиночный или двойной щелчок по элементу списка, список посылает в родительское окно (в нашем случае – окно модального диалога) уведомляющее сообщение WM_NOTIFY. Включив в оконную функцию диалогового окна обработку сообщения WM_NOTIFY, мы получим возможность реагировать требуемым образом на действия пользователя.

Приводимая ниже программа 10-2 отличается от предыдущего примера главным образом наличием функции DlgOnNotify() обработки сообщения WM_NOTIFY. В этой функции выполняется анализ действия пользователя и при обнаружении однократного щелчка мышью по какой-либо строке списка копирование выбранного файла на дискету.

```
/*Программа 10-2. Уведомляющие сообщения*/
/*Файл 10-2.h*/
#define ID_LISTVIEW 100
BOOL CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM);
void DlgOnCommand(HWND,int,HWND,UINT);
BOOL DlgOnInitDialog(HWND,HWND,LPARAM);
BOOL DlgOnNotify(HWND,INT,NMHDR*);//Включен еще один прототип
/*Файл 10-2.rc*/
Files DIALOG 40, 20, 130, 100
CAPTION "Программа 10-2"
  BEGIN
  END
/*Файл 10-2.cpp*/
#include <windows.h>
#include <windowsx.h>
#include "10-2.h"
#include <commctrl.h>
HINSTANCE hInstance;//Для дескриптора экземпляра приложения
LV_ITEM lvi;//Структура для элементов списка стала глобальной
char szFileName[80];//Символьная строка для имен файлов стала глобальной
/*Главная функция WinMain()*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  hInstance=hInst;//Сохраним дескриптор экземпляра приложения
  InitCommonControls();//Загрузка CMCTL32.DLL
  DialogBox(hInst,"Files",NULL,DlgProc);
  return 0;
  }
/*Оконная процедура диалогового окна*/
BOOL CALLBACK DlgProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_INITDIALOG,DlgOnInitDialog);
    HANDLE_MSG(hwnd,WM_COMMAND,DlgOnCommand);
```

```
    HANDLE_MSG(hwnd,WM_NOTIFY,DlgOnNotify);//Добавилась обработка WM_NOTIFY
    default:
      return FALSE;
    }
  }
/*Функция обработки сообщения об инициализации диалога*/
BOOL DlgOnInitDialog(HWND hwnd,HWND,LPARAM){
  char szName[]="Имя файла";//Текст первого поля заголовка
  char szSize[]="Размер файла";//Текст второго поля заголовка
  char szFileSize[10];//Символьная строка для размеров файлов
  HANDLE hFile,hOpenFile;//Для дескрипторов файлов
  DWORD dwFileSize;//Целочисленная переменная для размера файлов
  int nItemIndex=0;//Индекс строки списка
/*Создадим графический список*/
  HWND hwndList=CreateWindowEx(0,WC_LISTVIEW,NULL,//Класс графического списка
      WS_CHILD|WS_BORDER|WS_VISIBLE|LVS_REPORT,//Стиль: таблица (отчет)
      20,15,222,150,hwnd,(HMENU)ID_LISTVIEW,hInstance,NULL);
/*Заполним заголовок списка*/
  LV_COLUMN lvc;//Структура для заголовка списка
  ZeroMemory(&lvc,sizeof(lvc));//Обнуляем стрктуру
  lvc.mask=LVCF_FMT | LVCF_SUBITEM | LVCF_TEXT | LVCF_WIDTH;//Маска
  lvc.fmt=LVCFMT_LEFT;//Выравнивание влево
  lvc.cx=115;//Размер первого поля заголовка
  lvc.pszText=szName;//Строка с текстом первого поля заголовка
  ListView_InsertColumn(hwndList,0,&lvc);//Включаем поле в заголовок
  lvc.cx=90;//Размер второго поля заголовка
  lvc.pszText=szSize;//Строка с текстом второго поля заголовка
  ListView_InsertColumn(hwndList,1,&lvc);//Включаем поле в заголовок
/*Подготавливаем структуру для заполнения списка*/
  ZeroMemory(&lvi,sizeof(lvi));//Обнуляем структуру
  lvi.mask=LVIF_TEXT;//Маска
  lvi.pszText=szFileName;//Строка с текстом элемента
/*Прочитаем и выведем в диалог список файлов*/
  WIN32_FIND_DATA fd;//Структура для поиска файлов
  hFile=FindFirstFile("*.*",&fd);//Шаблон поиска
  if(hFile==INVALID_HANDLE_VALUE)//Если нет файлов
    return TRUE;
  else{
    hOpenFile=CreateFile(fd.cFileName,GENERIC_READ,0,0,OPEN_EXISTING,0,NULL);
    if(hOpenFile==INVALID_HANDLE_VALUE)//Если нельзя открыть
      dwFileSize=0;//Пусть размер в этом случае будет = 0
    else{//Если файл открылся
      dwFileSize=GetFileSize(hOpenFile,NULL);//Найдем его длину
      CloseHandle(hOpenFile);//Закроем файл, освободим его дескриптор
      }
    wsprintf(szFileSize,"%d",dwFileSize);//Преобразуем длину в символы
    strcpy(szFileName,fd.cFileName);//Получим имя файла
    lvi.iItem=nItemIndex;//Текущий индекс элемента списка (=0)
    ListView_InsertItem(hwndList,&lvi);//Включим имя файла в список
    ListView_SetItemText(hwndList,nItemIndex,1,szFileSize);//Включим длину
    nItemIndex++;//К следующему элементу
    }
  while(FindNextFile(hFile,&fd)){//Пока есть файлы
    hOpenFile=CreateFile(fd.cFileName,GENERIC_READ,0,0,OPEN_EXISTING,0,NULL);
    if(hOpenFile==INVALID_HANDLE_VALUE)//Если нельзя открыть
      dwFileSize=0;//Пусть в этом случае размер будет = 0
    else{//Если файл открылся
      dwFileSize=GetFileSize(hOpenFile,NULL);//Найдем его длину
      CloseHandle(hOpenFile);//Закроем файл, освободим его дескриптор
      }
    wsprintf(szFileSize,"%d",dwFileSize);//Преобразуем длину в символы
```

```
    strcpy(szFileName,fd.cFileName);//Получим имя файл
    lvi.iItem=nItemIndex;//Текущий индекс элемента списка
    ListView_InsertItem(hwndList,&lvi);//Включим имя файла в список
    ListView_SetItemText(hwndList,nItemIndex,1,szFileSize);//Включим длину
    nItemIndex++;//К следующему элементу
    }
  return TRUE;
  }
/*Функция обработки сообщений от элементов управления*/
void DlgOnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id){
    case IDCANCEL:
      EndDialog(hwnd,0);
    }
  }
/*Функция обработки уведомляющих сообщений от графического списка */
BOOL DlgOnNotify(HWND hwnd,INT,NMHDR* lpnmhdr){
/*Получим имя выбранного в списке файла*/
  if(lpnmhdr->code==NM_CLICK){//Если щелкнули по имени файла
    SendDlgItemMessage(hwnd,ID_LISTVIEW,LVM_GETITEMTEXT,//Пошлем
      ((NM_LISTVIEW)lpnmlv)->iItem,(LPARAM)&lvi);//сообщение: получить текст
/*Сформируем спецификацию того же файла на дискете*/
    char szFileNameA[80];//Вспомогательная строка
    strcpy(szFileNameA,"a:\\");//Скопируем в нее имя диска
    strcat(szFileNameA,szFileName);//Добавим имя файла
    CopyFile(szFileName,szFileNameA,FALSE);//Копирование файла на дискету
    }
  return TRUE;
  }
```

Рассмотрим, чем приведенный текст отличается от программы 10-1.

В файл заголовков включен прототип новой функции DlgOnNotify() обработки сообщений WM_NOTIFY. Формат этой функции, т. е. типы принимаемых ею параметров и возвращаемого значения, можно определить, изучив расширение макроса HANDLE_WM_NOTIFY, содержащееся в файле COMMCTRL.H. О назначении параметров этой функции мы поговорим позднее.

Включение в программу новой функции, которая должна обращаться к некоторым переменным программы, потребовало объявление этих переменных глобальными. Соответственно объявления переменных lvi и szFileName перенесены из функции DlgOnInitDialg() в начало программы, до главной функции WinMain().

В оконную функцию диалога DlgProc() к двум имевшимся макросам HANDLE_MSG добавлен третий:

```
HANDLE_MSG(hwnd,WM_NOTIFY,DlgOnNotify);
```

Существенное для выполнения программы дополнение внесено в блок подготовки структурной переменной lvi типа LVI_ITEM. В этой переменной инициализируется не только адрес строки с текстом lvi.pszText, но и максимально возможная длина этой строки (элемент структуры cchTextMax). Лишь в этом случае структурная переменная lvi сможет принимать из Windows атрибуты элементов списка (в нашем случае – текст элемента, т. е. имя файла).

Весь дальнейший текст функции DlgOnInitDialog() – поиск файлов, определение их размеров и заполнение графического списка именами и размерами файлов – перенесен из программы 10-1 без изменений.

Функция DlgOnNotify() получает управление, когда пользователь выполняет какие-либо манипуляции со списком, и в окно диалога приходит уведомляющее сообщение

WM_NOTIFY. Как известно, вместе с любым сообщением Windows в приложение поступают два параметра, в общем случае имеющие типы WPARAM и LPARAM. Смысл этих параметров зависит от вида сообщения. Макросы HANDLE_WM *_сообщение* преобразуют для каждого конкретного сообщения типы параметров соответственно их смыслу для данного сообщения и в ряде случаев распаковывают параметры, выделяя из них составляющие. Вместе с сообщением WM_NOTIFY, поступающим от стандартного элемента управления, Windows присылает в качестве первого параметра идентификатор элемента управления (типа int), а в качестве второго – адрес структуры типа NMHDR, содержащей дополнительную информацию:

```
typedef struct tagNMHDR {
  HWND hwndFrom;//Дескриптор элемента управления - отправителя
  UINT idFrom;//Идентификатор элемента управления - отправителя
  UINT code;//Код уведомления
  } NMHDR;
```

Для нас в этой структуре важен элемент code. Он называется *кодом уведомления* и через него передается информация о действии пользователя. Например, при щелчке по строке списка левой клавишей мыши передается код уведомления, равный NM_CLICK (конкретно число -2); при двойном щелчке – код NM_DBLCLK (-3); при щелчке правой клавишей – код NM_RCLICK (-5) и т. д.

Оба параметра передаются в функцию обработки уведомляющего сообщения DlgOnNotify(), что и определило ее прототип:

```
BOOL DlgOnNotify(HWND hwnd, int idCtrl, NMDR FAR* lpnmhdr)
```

Однако в случае графического списка в качестве второго параметра уведомляющего сообщения в приложение поступает адрес большей по объему структуры типа NM_LISTVIEW, в качестве первых элементов которой выступают члены "стандартной" структуры NMHDR:

```
typedef struct tagNM_LISTVIEW {
  NMHDR hdr;//Структура NMHDR
  int   iItem;//Основной элемент списка
  int   iSubItem;//Дополнительный элемент списка
  UINT  uNewState;//Новое состояние элемента
  UINT  uOldState;//Старое состояние элемента
  UINT  uChanged;//Набор флагов изменения атрибутов
  POINT ptAction;//Структура POINT с координатами события
  LPARAM lParam;//Произвольное 32-битовое данное для приложения
  } NM_LISTVIEW;
```

Нам из этой структуры будет нужен элемент iItem – порядковый номер выбранной пользователем строки списка.

Поскольку структура NM_LISTVIEW начинается с элементов структуры NMHDR, их адреса совпадают и передаваемый из Windows в приложение параметр lpnmhdr можно считать указателем как на одну, так и на другую структуру. Для того чтобы получить возможность обращаться через этот указатель к "полной" структуре NM_LISTVIEW, достаточно преобразовать его тип, что и делается в том предложении функции DlgOnNotify(), где требуется обращение к элементу iItem, выходящему за рамки структуры NMHDR:

```
SendDlgItemMessage(hwnd,ID_LISTVIEW,LVM_GETITEMTEXT,//Пошлем
      ((NM_LISTVIEW)lpnmlv)->iItem,(LPARAM)&lvi);//сообщение: получить текст
```

Этим предложением графическому списку (идентификатор ID_LISTVIEW) посылается характерное для этого элемента управления сообщение LVM_GETITEMTEXT – "получить текст основного или дополнительного элемента списка". Номер требуемой

строки извлекается из соответствующего элемента структуры NM_LISTVIEW и передается через 4-й параметр сообщения, а номер столбца сообщение получает через свой 5-й, последний параметр, в качестве которого выступает адрес структурной переменной lvi типа LV_ITEM. В этой структуре номер столбца определяется значением элемента iSubItem, которое в нашем случае равно нулю.

Через тот же 5-й параметр Windows возвращает результат действия сообщения LVM_GETITEMTEXT. Искомый текст элемента списка (имя файла) поступает в символьную строку, адрес которой содержится в элементе pszText структуры LV_ITEM. Таким образом, результат действия сообщения как бы проходит два уровня косвенной адресации: через параметр lvi происходит обращение к структуре LV_ITEM, а через ее элемент pszText – к программной переменной szFileName.

В промежуточной переменной szFileNameA происходит сцепление полученного из списка имени файла с именем диска. Сначала в переменную szFileNameA функцией strcpy() копируется строка "a:\", а затем к ней функцией strcat() добавляется имя файла непосредственно из переменной szFileName. Далее с помощью 32-разрядной функции CopyFile() осуществляется копирование файла на дискету.

Индикатор прогресса

Индикатор прогресса (progress bar) представляет собой окно в виде линейки, заполняемое небольшими яркими полосками по мере продвижения некоторого процесса, например копирования файлов, установки системы или просто длительных вычислений (рис. 10.3).

Рис. 10.3. Индикатор прогресса

Для того чтобы индикатор прогресса мог выполнять свои функции, индицируемый процесс должен состоять из некоторого известного заранее количества шагов, завершение каждого из которых и будет продвигать индикатор прогресса на одно деление.

Рассмотрим простую программу 10-3, которая создает в памяти большой массив объемом, например, 10 М целых чисел или более и заполняет его случайными числами. Такого рода тестовые массивы (заполненные, возможно, натуральным рядом чисел, или четными числами, или просто нулями, или, наконец, как у нас, случайными числами) часто приходится создавать в процессе отладки и тестирования программ обработки данных. Поскольку эта операция может занять значительное время, которое зависит и от скорости работы процессора, и от наличного объема оперативной памяти, предусмотрим в программе индикатор прогресса. Чтобы обеспечить его функционирование, разобьем цикл заполнения памяти на два вложенных цикла, выполняя в конце каждого шага внутреннего цикла продвижение индикатора прогресса на одну позицию. На рис. 10.4 показан вывод программы на этапе заполнения массива.

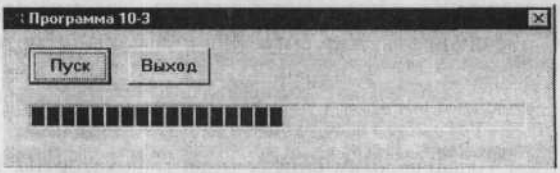


Рис. 10.4. Промежуточное состояние процесса

```
/*Программа 10-3. Индикатор прогресса*/
/*Файл 10-3.h*/
#define ID_START 101
#define ID_CLOSE 102
#define ID_TEXT 103
BOOL CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM);
void DlgOnCommand(HWND,int,HWND,UINT);
BOOL DlgOnInitDialog(HWND,HWND,LPARAM);
/*Файл 10-3.rc*/
#include "8-4.h"
Main DIALOG 40, 30, 220, 55
CAPTION "Программа 10-3" {
  PUSHBUTTON "Пуск", ID_START, 10, 7, 33, 15
  PUSHBUTTON "Выход", ID_CLOSE, 50, 7, 33, 15
  LTEXT "",ID_TEXT,90,10,120,9
  }
/*Файл 10-3.cpp*/
#include <windows.h>
#include <windowsx.h>
#include "10-3.h"
#include <commctrl.h>
HINSTANCE hI;//Дескриптор экземпляра приложения
HWND hwndBar;//Дескриптор индикатора прогресса
const NMAX=100;//Константное данное - число внешних шагов
int nTest[100000*NMAX];//Тестовый массив в памяти
/*Главная функция WinMain()*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  hI=hInst;
  InitCommonControls();
  DialogBox(hInst,"Main",NULL,DlgProc);
  return 0;
  }
/*Оконная процедура диалогового окна*/
BOOL CALLBACK DlgProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_INITDIALOG,DlgOnInitDialog);
    HANDLE_MSG(hwnd,WM_COMMAND,DlgOnCommand);
    default:
      return FALSE;
    }
  }
/*Функция обработки сообщения об инициализации диалога*/
BOOL DlgOnInitDialog(HWND hwnd,HWND,LPARAM){
/*Создадим индикатор прогресса*/
  hwndBar=CreateWindowEx(0,PROGRESS_CLASS,NULL,
      WS_CHILD|WS_VISIBLE,20,60,400,20,hwnd,NULL,hI,NULL);
  SendMessage(hwndBar,PBM_SETRANGE,0,MAKELPARAM (0,NMAX));//Диапазон
  SendMessage(hwndBar,PBM_SETSTEP,(WPARAM)1,0);//Шаг
  return TRUE;
  }
/*Функция обработки сообщений от элементов управления*/
void DlgOnCommand(HWND hwnd,int id,HWND,UINT){
  char szStopText[]="Заполнение памяти завершено";
  switch(id){
    case ID_START:{//Начнем заполнение памяти
      for(int i=0;i<NMAX;i++){//Внешний цикл
        for(int j=0;j<100000;j++)//Внутренний цикл
          nTest[j]=random(1000000000);//Случайные числа в память
        SendMessage(hwndBar,PBM_STEPIT,0,0);//Шаг индикатора
        }//Конец внешнего for
```

```
      SetWindowText(GetDlgItem(hwnd,ID_TEXT),szStopText);//Текст на экран
      break;
      }//Конец case ID_START
    case ID_CLOSE:
    case IDCANCEL:
      EndDialog(hwnd,0);
    }//Конец switch(id)
  }//Конец DlgOnCommand
```

В заголовочном файле 10-3.H описаны идентификаторы трех элементов управления в диалоговом окне – двух кнопок (ID_START и ID_CLOSE) и строки с текстом (ID_TEXT). Идентификатор индикатора прогресса нам не понадобится, в данной программе мы будем использовать дескриптор, полученный в процессе создания этого элемента управления.

В файле ресурсов 10-3.RC описано диалоговое окно с тремя элементами управления – двумя нажимаемыми кнопками класса PUSHBUTTON и текстом (класс LTEXT). Поскольку текст мы планируем выводить на экран лишь после завершения процесса, в файле ресурсов текст элемента управления ID_TEXT указан в виде пустой строки.

Индикатор прогресса принадлежит к числу стандартных элементов управления, для реализации которых в программе требуется, во-первых, подключение заголовочного файла COMMCTRL.H (оператором #include) и, во-вторых, загрузка динамической библиотеки COMCTL32.DLL (вызовом функции InitCommonControls()).

В начале программы объявляются глобальные переменные, которые будут использоваться в нескольких функциях. Отметим среди них константную переменную NMAX, которая будет определять объем заполняемой памяти, число шагов внешнего цикла и диапазон индикатора прогресса. Особого смысла вводить константную переменную в нашем случае нет, можно было объявить это число как константу в заголовочном файле:

```
#define NMAX 100
```

Программа 10-3 построена на базе модального диалога, который создается в главной функции WinMain() вызовом функции DialogBox(). Действия по созданию индикатора прогресса выполняются на этапе инициализации диалогового окна в функции DlgOnInitDialog(), а действия по заполнению массива памяти вынесены в функцию DlgOnCommand(), чтобы можно было их инициировать нажатием кнопки "Пуск".

Окно индикатора прогресса, как и окно графического списка, создается вызовом функции CreateWindowEx() с указанием в качестве класса окна символического обозначения PROGRESS_CLASS. Стиль окна индикатора прогресса включает обычные для таких случаев константы WS_CHILD и WS_VISIBLE; при подготовке рис. 10.3 к ним для наглядности была добавлена константа WS_BORDER, определяющая наличие вокруг элемента управления тонкой черной рамки.

Далее посылкой в созданный элемент управления двух сообщений PBM_SETRANGE и PBM_SETSTEP задаются диапазон и шаг индикатора. Для задания диапазона требуется указать его начальное и конечное значения, т. е. два числа, которые должны быть упакованы в один (последний) параметр функции SendMessage(). Для такой упаковки предусмотрен специальный макрос MAKELPARAM, в качестве двух параметров которого указываются упаковываемые числа. Макрос помещает первый из указанных параметров в младшую половину длинного данного типа LPARAM, а второй параметр – в его старшую половину.

Заполнение массива памяти случайными числами выполняется с помощью вызова функции C++ random(), в качестве параметра которой указывается диапазон генерируемых случайных чисел. В нашем примере в качестве диапазона произвольно выбрано значение 1 миллиард.

Как уже отмечалось, для обеспечения функционирования индикатора прогресса цикл разбит на два, внешний с числом шагов NMAX и внутренний из 100000 шагов. В конце каждого шага внешнего цикла в индикатор прогресса посылается сообщение PBM_STEPIT, которое приводит к продвижению указателя индикатора на одно деление.

После завершения всего процесса заполнения памяти вызывается функция SetWindowText(), которая служит для изменения текста элемента управления. В качестве первого параметра этой функции должен быть указан дескриптор элемента управления, в нашем случае – дескриптор статического элемента управления класса LTEXT. Мы не знаем дескриптора этого элемента, потому что он был создан не программно, вызовом функции CreateWindow(), а путем описания его в файле ресурсов. Однако нам известен его идентификатор ID_TEXT; в этом случае для получения дескриптора элемента управления следует воспользоваться функцией GetDlgItem(). В качестве второго параметра функции SetWindowTex() указывается адрес выводимого текста. В результате после завершения процесса диалоговое окно выглядит так, как показано на рис. 10.5.

Рис.10.5. Процесс завершился

В рассмотренной программе не предусмотрен вывод содержимого массива или вообще какая-либо работа с ним; чтобы убедиться в правильной работе программы, следует остановить ее выполнение в отладчике после некоторого шага внутреннего цикла и посмотреть содержимое переменной nTest. Пример вывода отладчика приведен на рис. 10.6.

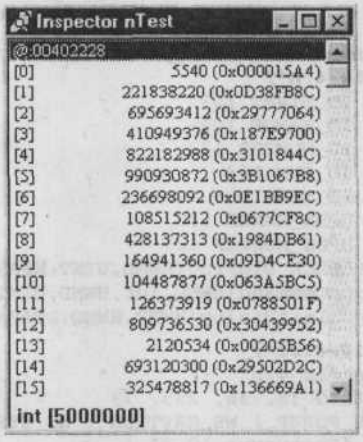


Рис. 10.6. Память заполнена случайными числами

Можно видеть, что числа, заполняющие массив nTest, действительно носят случайный характер; во всяком случае среди них встречаются как довольно маленькие (55 540), так и весьма солидные (693 120 300) числа.

Наборный счетчик

Наборный счетчик (up-down control или spin control) (рис. 10.7) представляет собой пару кнопок со стрелками, с помощью которых можно увеличивать или уменьшать значение некоторой величины и использовать затем это значение в программе. Часто рядом со стрелками располагают небольшое окно ввода, в котором фиксируется выбранное с помощью стрелок значение и которое можно также использовать для ввода требуемого значения с клавиатуры. Поскольку это окно тесно связано с основным окном счетчика, его называют "приятелем счетчика" (buddy).

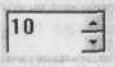


Рис. 10.7. Наборный счетчик с "приятелем" – окном ввода

Рассмотрим программу, в которой генерируется массив из некоторого количества случайных чисел и затем определяется сумма и среднее значение полученной выборки. При небольшом усложнении такая программа может быть полезна при изучении законов распределения случайных величин. Воспользуемся наборным счетчиком для ввода в программу двух исходных значений: диапазона генерируемых чисел и их количества. Результат одного из прогонов программы представлен на рис.10.8.

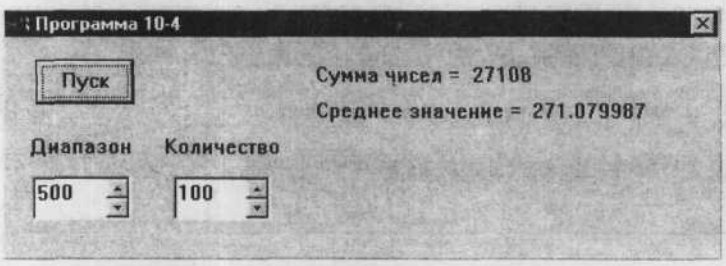


Рис. 10.8. Результат работы программы 10-4

```
/*Программа 10-4. Наборный счетчик*/
/*Файл 10-4.h*/
#define ID_START 101
#define ID_RANGE 102
#define ID_NUMBER 103
#define ID_SUM 104
#define ID_MEAN 105
#define UPDN_RANGE 106
#define UPDN_NUMBER 107
BOOL CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM);
void DlgOnCommand(HWND,int,HWND,UINT);
BOOL DlgOnInitDialog(HWND,HWND,LPARAM);
```

```
/*Файл 10-4.rc*/
#include "10-4.h"
Main DIALOG 38, 36, 239, 73
STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Программа 10-4" {
  PUSHBUTTON "Пуск", ID_START, 10, 7, 33, 15
  EDITTEXT ID_RANGE, 9,46,33,15, WS_CHILD|WS_VISIBLE|WS_BORDER|ES_NUMBER
  EDITTEXT ID_NUMBER, 56,46,33,15, WS_CHILD|WS_VISIBLE|WS_BORDER|ES_NUMBER
  LTEXT "Сумма чисел = ", -1, 104, 8, 50, 11, WS_CHILD | WS_VISIBLE
  LTEXT "Среднее значение = ", -1, 104, 20, 71, 11, WS_CHILD | WS_VISIBLE
  LTEXT "Диапазон", -1, 8, 32, 36, 10, WS_CHILD | WS_VISIBLE
  LTEXT "Количество", -1, 53, 32, 41, 10, WS_CHILD | WS_VISIBLE
```

```
  LTEXT "", ID_SUM, 157, 8, 49, 11, WS_CHILD | WS_VISIBLE
  LTEXT "", ID_MEAN, 177, 20, 54, 11, WS_CHILD | WS_VISIBLE
  }
/*Файл 10-4.cpp */
#include <windows.h>
#include <windowsx.h>
#include "10-4.h"
#include <commctrl.h>
#include <stdio.h>//Ради прототипа sprintf()
HINSTANCE hInstance;
/*Главная функция WinMain()*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  hInstance=hInst;
  InitCommonControls();
  DialogBox(hInst,"Main",NULL,DlgProc);
  return 0;
  }
/*Оконная функция диалогового окна*/
BOOL CALLBACK DlgProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_INITDIALOG,DlgOnInitDialog);
    HANDLE_MSG(hwnd,WM_COMMAND,DlgOnCommand);
    default:
      return FALSE;
    }
  }
/*Функция обработки сообщения об инициализации диалога*/
BOOL DlgOnInitDialog(HWND hwnd,HWND,LPARAM){
  CreateUpDownControl//Создаем 1-й наборный счетчик
    (WS_CHILD|WS_VISIBLE|WS_BORDER|UDS_ALIGNRIGHT|
     UDS_ARROWKEYS|UDS_SETBUDDYINT,0,0,0,0,hwnd,
     UPDN_RANGE,hInstance,GetDlgItem(hwnd,ID_RANGE),1000,1,100);
  CreateUpDownControl//Создаем 2-й наборный счетчик
    (WS_CHILD|WS_VISIBLE|WS_BORDER|UDS_ALIGNRIGHT|
    UDS_ARROWKEYS|UDS_SETBUDDYINT,0,0,0,0,hwnd,
    UPDN_NUMBER,hInstance,GetDlgItem(hwnd,ID_NUMBER),100,1,10);
  return TRUE;
  }
/*Функция обработки сообщений от элементов управления*/
void DlgOnCommand(HWND hwnd,int id,HWND,UINT){
int nRange,nNumber;//Диапазон генерируемых чисел, их количество
int nSum;//Сумма выборки
float fMean;//Среднее значение
PINT pRandoms;//Указатель на целые для массива случайных чисел
  switch(id){
    case ID_START:{
      nRange=SendDlgItemMessage
               (hwnd,UPDN_RANGE,UDM_GETPOS,0,0);//Читаем диапазон
      nNumber=SendDlgItemMessage
               (hwnd,UPDN_NUMBER,UDM_GETPOS,0,0);//Читаем количество
/*Выделим память под случайные числа и заполняем ее*/
      HANDLE hHeap=GetProcessHeap();//Указатель на кучу процесса
      pRandoms=(PINT)HeapAlloc
               (hHeap,HEAP_ZERO_MEMORY,nNumber*4);//Выделяем память
      for(int i=0;i<nNumber;i++)//В цикле от 0 до nNumber получаем случайные
        pRandoms[i]=random(nRange);//числа и заполняем ими выделенную память
/*Вычислим сумму и среднее */
      nSum=0;
      for(int j=0;j<nNumber;j++)//В цикле складываем все числа
        nSum+=pRandoms[j];//Сумма
```

```
      fMean=(float)nSum/nNumber;//Среднее арифметическое
/*Выведем результаты в окна диалога*/
      char szT[20];
      wsprintf(szT,"%ld",nSum);//Сумму в символьный вид
      SetWindowText(GetDlgItem(hwnd,ID_SUM),szT);//В окно диалога
      sprintf(szT,"%f",fMean);//Среднее в символьный вид
      SetWindowText(GetDlgItem(hwnd,ID_MEAN),szT);//В окно диалога
      HeapFree(hHeap,0,pRandoms);//Освободим память
      break;
      }//Конец case ID_START
    case IDCANCEL:
      EndDialog(hwnd,0);
    }//Конец seitch(id)
  }//Конец DlgOnCommand
```

В заголовочном файле определено большое количество констант, служащих идентификаторами элементов управления диалогом, а в файле ресурсов описаны сами элементы за исключением двух наборных счетчиков, которые создаются программно с помощью специально предназначенной для этого функции CreateUpDownControl(). На рис. 10.9 выделены все элементы управления диалоговым окном с указанием их классов и идентификаторов.

Рис. 10.9. Элементы управления диалоговым окном

Текстовые строки "Диапазон", "Количество", "Сумма чисел" и "Среднее значение" не являются управляемыми и всем им присвоен одинаковый идентификатор –1. Текстовые поля для суммы и среднего значения заполняются программно, и для того, чтобы к ним можно было обращаться, им присвоены уникальные идентификаторы ID_SUM и ID_MEAN. "Приятели" наборных счетчиков объявлены как поля для редактирования текста (класс EDITTEXT) и также имеют уникальные идентификаторы ID_RANGE и ID_NUMBER. Приданный им стиль ES_NUMBER предохраняет от случайного ввода алфавитных символов, позволяя задавать в этих полях только числа.

Сценарий диалога в файле ресурсов содержит предложение STYLE, в котором указывается стиль всего диалогового окна – всплывающий, с заголовком и системным меню. Для диалога, выступающего в качестве главного окна приложения, стиль можно не указывать, однако при составлении сценария диалога с помощью редактора ресурсов, входящего в состав среды разработки, это предложение включается автоматически.

Построение всей программы традиционно для приложений на основе модального диалога. В главной функции WinMain() открывается диалог; в диалоговой процедуре DlgProc() с помощью макросов HANDLE_MSG определены функции обработки сообщений Windows; в функции DlgOnInitDialg() обработки сообщения об инициализации диалога выполняются все необходимые подготовительные действия; в функцию DlgOnCommand() обработки сообщений от элементов управления вынесена содержательная часть программы.

Создать наборный счетчик можно разными способами: предложениями файла ресурсов (как обычный элемент управления), вызовом функции CreateWindowEx(), как это мы делали для графического списка, и, наконец, с помощью функции CreateUpDownControl(). Мы использовали последний вариант.

Прототип функции CreateUpDownControl() выглядит следующим образом:

```
HWND CreateUpDownControl(
  DWORD  dwStyle,//Набор констант стиля наборного счетчика
  int  x,//Горизонтальная координата верхнего левого угла счетчика
  int  y,//Вертикальная координата верхнего левого угла счетчика
  int  cx,//Ширина наборного счетчика в пикселах
  int  cy,//Высота наборного счетчика в пикселах
  HWND  hParent,//Дескриптор родительского окна
  int  nID,//Идентификатор наборного счетчика
  HINSTANCE  hInst,//Дескриптор экземпляра приложения
  HWND  hBuddy,//Дескриптор окна-"приятеля"
  int  nUpper,//Верхний предел значений наборного счетчика
  int  nLower,//Нижний предел значений наборного счетчика
  int  nPos//Начальная позиция наборного счетчика
  );
```

Среди многочисленных параметров этой функции на первом месте задается комбинация стилей. В нашем примере кроме обычных для любых элементов управления стилей WS_CHILD, WS_VISIBLE и WS_BORDER указаны специфические для наборного счетчика стили: UDS_ALIGNRIGHT задает расположение окна счетчика справа от "приятеля" (константа UDS_ALIGNLEFT – слева), UDS_ARROWKEYS позволяет выполнять инкремент или декремент числа в окне-"приятеле" щелчками по стрелкам счетчика, а UDS_SETBUDDYINT приводит к отображению в окне-"приятеле" нового значения, если оно изменено с помощью наборного счетчика.

Следующие 4 параметра определяют положение и размеры наборного счетчика. Они нужны, только если наборный счетчик используется без окна-"приятеля"; при наличии последнего размеры счетчика подстраиваются под размеры "приятеля".

Далее следуют дескриптор родительского окна (в данном случае дескриптор окна диалога, переданный из Windows через параметр функции DlgOnInitDialog()), произвольный идентификатор счетчика, дескриптор экземпляра приложения.

В качестве девятого параметра функции CreateUpDownControl() выступает дескриптор окна-"приятеля". Нам этот дескриптор неизвестен, и мы находим его с помощью функции GetDlgItem(), в качестве входного параметра которой указывается идентификатор окна-приятеля", определенный нами в файле ресурсов.

Наконец, последние 3 параметра обозначают максимальную и минимальную границы задаваемого с помощью счетчика значения и значение по умолчанию, устанавливаемое в окне-"приятеле" при запуске программы.

Как уже отмечалось, все содержательные действия описаны в функции обработки сообщений от элементов управления DlgOnCommand() и выполняются после нажатия пользователем кнопки "Пуск" (идентификатор ID_START). С помощью функции SendDlgItem

Message() обоим наборным счетчикам посылаются предусмотренные для них сообщения UDM_GETPOS для чтения состояния счетчиков (или, что то же самое, содержимого окон ввода). Поскольку объем создаваемого в программе массива случайных чисел заранее неизвестен, память под этот массив выделяется динамически из кучи процесса. Сначала с помощью функции GetProcessHeap() определяется дескриптор кучи, затем функцией HeapAlloc() выделяется память в количестве nNumber*4 байт (случайные числа являются целыми, и каждое занимает 4 байта). Полученная от системы память в цикле заполняется случайными числами с помощью функции C++ random(). Эта функция генерирует равномерно распределенные псевдослучайные числа; нетрудно ввести в программу изменение закона распределения с помощью соответствующих математических преобразований и тем самым сделать программу более универсальной.

После заполнения массива исходными случайными числами можно приступить к вычислению требуемых статистических характеристик полученной выборки, например среднего, дисперсии, среднеквадратического отклонения и др. Мы в нашем примере ограничились вычислением суммы чисел и их среднего значения.

После преобразования полученных числовых характеристик в символьную форму результаты вычислений с помощью функции SetWindowText() посылаются соответствующим статическим элементам управления диалога класса LTEXT и отображаются в окне диалога.

Перед завершением функции DlgOnCommand() выполняется обязательное действие – возврат системе с помощью функции HeapFree() выделенной памяти.

Глава 11
Работа с файлами

Базовые операции с файлами

Те или иные операции с файлами используются почти в любом приложении Windows. Действительно, трудно представить себе программу, которая вводит информацию только с клавиатуры и выводит ее только на экран. В этом случае объемы обрабатываемых данных будут невелики, а результаты работы программы недолговечны. Чаще и исходные для программы данные, и данные, полученные в результате ее работы, хранятся на дисках в виде файлов; иногда программе приходится по ходу ее работы создавать еще и временные файлы для хранения промежуточной информации. Win32 предоставляет как традиционные средства работы с файловой системой (создание и удаление файлов и каталогов, чтение и запись, поиск, изменение характеристик и др.), так и специфические средства, не поддерживаемые в Win16 или MS-DOS. К последним прежде всего следует отнести проецирование файлов в память и расширенное толкование понятия физической памяти как суммы оперативной памяти и страничных файлов (файлов подкачки).

В предыдущих главах книги мы уже сталкивались с некоторыми файловыми операциями, в частности с получением информации о файловой системе, а также с поиском, чтением и копированием файлов. В настоящей главе будет дан краткий обзор наиболее важных файловых операций Win32, а также рассмотрены новые средства работы с файлами.

Открытие и закрытие файла

Перед тем как начать работать с файлом, программа должна либо его создать заново, либо открыть, если он был создан ранее. И та и другая операция выполняются в Win32 с помощью одной универсальной функции – CreateFile(). При успешном выполнении эта функция возвращает 32-разрядный дескриптор файла типа HANDLE, через который потом и выполняются все операции с данным файлом. Если файл открыть не удалось, функция CreateFile() возвращает 0xFFFFFFFF (–1). Функция CreateFile()имеет следующий прототип:

```
HANDLE CreateFile(
  LPCTSTR  lpFileName,//Адрес спецификации файла
  DWORD  dwDesiredAccess,//Режим доступа
  DWORD  dwShareMode,//Режим разделения
  LPSECURITY_ATTRIBUTES  lpSecurityAttributes,//Адрес дескриптора защиты
  DWORD  dwCreationDistribution,//Режим открытия
  DWORD  dwFlagsAndAttributes,//Атрибуты файла
  HANDLE  hTemplateFile //Дескриптор открытого файла-шаблона
  );
```

Параметр lpFileName описывает спецификацию файла по обычным правилам, например "Myfile.001" или "C:\\DataDir\\1.1" .

Параметр dwDesiredAccess при работе с файлами чаще всего указывается в виде GENERIC_READ | GENERIC_WRITE, что обеспечивает доступ как для чтения, так и для записи.

Параметр dwShareMode имеет значение в тех случаях, когда к файлу могут одновременно обращаться несколько приложений (или несколько потоков одного приложения).

Параметр может принимать значение FILE_SHARE_READ, FILE_SHARE _WRITE или их комбинации. Если файл будет использоваться в монопольном режиме, на месте этого параметра указывается 0.

Параметр lpSecurityAttributes указывает на структурную переменную типа SECURITY_ATTRIBUTES, которая позволяет задать атрибуты защиты для данного объекта. Этот параметр имеет смысл главным образом в серверных приложениях; создавая обычную прикладную программу, которая будет выполняться на автономной или локальной машине, на месте этого параметра можно указать NULL.

Параметр dwCreationDistribution может принимать следующие значения:

CREATE_NEW – создается новый файл; если файл уже существует, функция CreateFile() возвращает ошибку;

CREATE_ALWAYS – создается новый файл; если файл с указанным именем уже существует, он затирается вновь создаваемым;

OPEN_EXISTING – открывается существующий файл; если такого файла нет, функция CreateFile() возвращает ошибку;

OPEN_ALWAYS – открывается существующий файл; если такого файла нет, функция CreateFile() создает новый файл с указанным именем;

TRUNCATE_EXISTING – существующий файл открывается и урезается до нуля. Если такого файла нет, функция CreateFile() возвращает ошибку. Используется только при указании в режиме доступа значения GENERIC_WRITE.

Параметр dwFlagsAndAttributes позволяет установить флаги, оптимизирующие алгоритмы кеширования при операциях с файлом, а также задать атрибуты файла, например FILE_ATTRIBUTE_READONLY, если файлу требуется придать атрибут "только для чтения". Один из устанавливаемых с помощью этого параметра флагов, конкретно – флаг FILE_FLAG_OVERLAPPED, позволяет организовать асинхронные операции. Суть асинхронной операции заключается в том, что система, запустив, например, операцию чтения из устройства, возвращает управление в программу, которая может таким образом продолжить выполнение параллельно с операцией чтения данных. Параллельные процессы будут рассмотрены в этой книге в последующих главах.

Параметр hTemplateFile может содержать дескриптор открытого файла-шаблона, атрибуты которого применяются и для вновь создаваемого файла. Обычно в качестве значения этого параметра используется NULL.

Таким образом, типичным для прикладной программы будут следующие формы вызова функции CreateFile():

при создании нового файла:

```
HANDLE hFile=CreateFile("001.dat", GENERIC_READ | GENERIC_WRITE,
              0, NULL, CREATE_ALWAYS, 0, NULL);
```

при открытии существующего файла:

```
HANDLE hFile=CreateFile("001.dat", GENERIC_READ | GENERIC_WRITE,
              0, NULL, OPEN_EXISTING, 0, NULL);
```

После завершения работы с файлом или перед завершением программы все открытые в ней файлы следует закрыть. Для этого используется функция CloseHandle() с указанием в качестве ее единственного параметра дескриптора закрываемого файла:

```
CloseHandle(hFile);
```

В процессе завершения приложения Windows закрывает все созданные им объекты, в частности файлы, так что во многих случаях заботу о закрытии файлов можно возложить на Windows.

Запись и чтение файла

При работе с данными, содержащимися в файле, чаще других используются 3 функции: записи в файл WriteFile(), чтения из файла ReadFile() и установки файлового указателя SetFilePointer().

Функции чтения и записи имеют фактически одинаковые прототипы:

```
BOOL ReadFile(HANDLE hFile,LPVOID lpBuffer,DWORD nNumberOfBytesToRead,
              LPDWORD lpNumberOfBytesRead,LPOVERLAPPED lpOverlapped);

BOOL WriteFile(HANDLE  hFile,LPCVOID lpBuffer,DWORD nNumberOfBytesToWrite,
              LPDWORD lpNumberOfBytesWritten,LPOVERLAPPED lpOverlapped);
```

Параметр hFile обозначает полученный от функции CreateFile() дескриптор созданного или открытого файла.

Параметр lpBuffer представляет собой адрес программного буфера, предназначенного для хранения данных, записываемых в файл или считываемых из файла. Обобщенный тип этого буфера – LPVOID или LPCVOID – позволяет обмениваться с файлом данными любой организации – отдельными байтами или скалярными переменными большей длины, а также массивами и структурами.

Третий параметр рассматриваемых функций определяет объем пересылаемых данных (в байтах).

Четвертый параметр является адресом программной переменной типа DWORD, в которую после завершения передачи данных будет записано истинное число переданных байтов. Иногда этой переменной пользуются для контроля правильности записи или чтения.

Последний параметр, lpOverlapped, используется в тех случаях, когда над файлом предполагается выполнять асинхронные операции. В случае обычных операций чтения или записи значение этого параметра должно быть NULL.

Функция SetFilePointer() установки файлового указателя служит для перемещения указателя в файле на заданный байт, чтобы последующая операция чтения или записи выполнила пересылку данных не с начала файла, а от положения указателя. Прототип этой функции имеет следующий вид:

```
DWORD SetFilePointer(HANDLE hFile,LONG lDistanceToMove,
            PLONG lpDistanceToMoveHigh,DWORD dwMoveMethod);
```

Параметр hFile обозначает полученный от функции CreateFile() дескриптор созданного или открытого файла.

Параметр lDistanceToMove задает 32-битовое смещение в файле до точки установки указателя.

Параметр lpDistanceToMoveHigh является адресом старшей половины смещения указателя, если в качестве этого смещения используется 64-битовое значение. Обычно на месте этого параметра указывается NULL.

Последний параметр dwMoveMethod определяет способ установки указателя. Этот параметр может принимать 3 значения:

FILE_BEGIN – указатель устанавливается от начала файла и, следовательно, рассматривается как положительное число или, более точно, число без знака. Максимальное смещение в файле достигает $2^{32} - 2 = 4$ Гбайт - 2;

FILE_CURRENT – указатель устанавливается от текущей позиции указателя и, следовательно, рассматривается как число со знаком, что позволяет смещать указатель как вперед, так и назад от текущего положения в пределах до 2 Гбайт;

FILE_END – указатель устанавливается от конца файла, причем смещение может быть как положительным, так и отрицательным.

Рассмотрим простую программу (пример 11-1), осуществляющую базовые операции над файлом.

```
/*Программа 11-1. Работа с файлом*/
#include <windows.h>
struct GOODS{//Структура, описывающая изделия на складе
  int Nmb;//Номер изделия
  int Price;//Цена изделия
  };
const int N=1000000;//Пусть на складе миллион образцов изделий
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
  GOODS Stock[N];//Создадим в памяти массив для всех изделий
  GOODS Unit1,Unit2;//Структурные переменные для двух изделий
  DWORD dwCount;//Для функций чтения/записи
  for(int i=0;i<N;i++){//Заполним массив структур
    Stock[i].Nmb=i+1;//тестовыми данными
    Stock[i].Price=(i+1)*10;
    }
  HANDLE hFile=CreateFile("stock.dat",GENERIC_READ|GENERIC_WRITE,
              0,NULL,CREATE_ALWAYS,0,NULL);
/*Запишем весь массив в файл*/
  WriteFile(hFile,Stock,sizeof(Stock),&dwCount,NULL);
/*Сместим указатель и прочитаем последнюю запись файла */
  SetFilePointer(hFile,sizeof(GOODS)*(N-1),NULL,FILE_BEGIN);
  ReadFile(hFile,&Unit1,sizeof(GOODS),&dwCount,NULL);
/*Закроем файл, откроем снова и прочитаем первую запись файла */
  CloseHandle(hFile);
  hFile=CreateFile("stock.dat",GENERIC_READ,0,NULL,OPEN_EXISTING,0,NULL);
  ReadFile(hFile,&Unit2,sizeof(GOODS),&dwCount,NULL);
/*Выведем прочитанное в окно сообщения*/
  char szT[200];
  wsprintf(szT,"Изделие № %d\nЦена %d рублей\n----------\n"
               "Изделие № %d\nЦена %d рублей",
               Unit1.Nmb,Unit1.Price,Unit2.Nmb,Unit2.Price);
  MessageBox(NULL,szT,"Info",MB_OK);
  CloseHandle(hFile);//Закроем для порядка файл
  return 0;
  }
```

Программа оформлена в виде одной только главной функции WinMain(), в которой не создается каких-либо окон, а просто выполняются необходимые действия по созданию, заполнению и чтению файла. В конце программы прочитанная информация выводится в окно сообщения.

Структура GOODS, описанная в начале программы, имитирует информацию о многочисленных образцах изделий, хранящихся на складе. В структуру входят две целочисленные переменные: Nmb для номера изделия и Price для его цены. В главной функции объявляется массив Stock структур типа GOODS, состоящий из миллиона элементов. Размерность массива задается с помощью константной переменной N, описанной перед главной функцией.

По ходу своего выполнения программа заполняет в цикле массив Stock тестовыми данными, которые легко проверить: натуральным рядом чисел от 1 до 1000000 для номеров изделий и теми же числами, умноженными на 10, для цен изделий. Затем функцией CreateFile() создается файл stock.dat с доступом для чтения и записи, а функцией WriteFile() весь этот массив (занимающий в общей сложности $8*10^6$ байт) записывается в файл.

С помощью функции SetFilePointer() файловый указатель смещается к последней записи файла и эта запись считывается в переменную Unit1. Для демонстрации действий по открытию файла файл STOCK.DAT сначала закрывается, а затем снова открывается той же функцией CreateFile(). Непосредственно после открытия файла его указатель содер-

жит 0, поэтому при использовании функции ReadFile() из файла считывается самая первая запись (с индексом 0).

Прочитанная из файла в переменные Unit1 и Unit2 информация преобразуется функцией wsprintf() в символьную форму с добавлением поясняющего текста и выводится в окно сообщения. После закрытия файла программа завершается. Вывод программы 11-1 приведен на рис. 11.1.

Рис. 11.1. Вывод программы 11-1

Обратите внимание на переменную dwCount, адрес которой передается функциям чтения/записи. В программе нам эта переменная, в сущности, не нужна, но присутствовать она должна обязательно, так как функции чтения/записи используют ее для возврата числа переданных байтов. После операции записи в этой переменной будет число 0x7A1200 = $= 8*10^6$, а после операций чтения – число 8.

Рассмотренная программа дает нам повод сказать несколько слов о файле определения модуля. До сих пор мы использовали в этом качестве (неявным образом) файл DEFAULT.DEF, имеющийся в составе пакета разработки и подключаемый к проекту по умолчанию. Если пример 11-1 подготовить к выполнению таким же образом, то трансляция и компоновка пройдут без ошибок, но программа при ее запуске будет разрушать систему. Причина здесь в том, что мы объявили в программе относительно большой локальный массив данных, имеющий размер около 8 Мбайт. Как известно, все локальные переменные компилятор C++ размещает в стеке, и для такого большого массива в стеке может не хватить места.

Сколько памяти выделяется под стек и как изменить его размер? Для этого надо с помощью главного меню среды разработки IDE выбрать команду Options➢Project, а затем, развернув в открывшемся списке пункт +Linker, выбрать пункт 32-bit Linker (рис. 11.2).

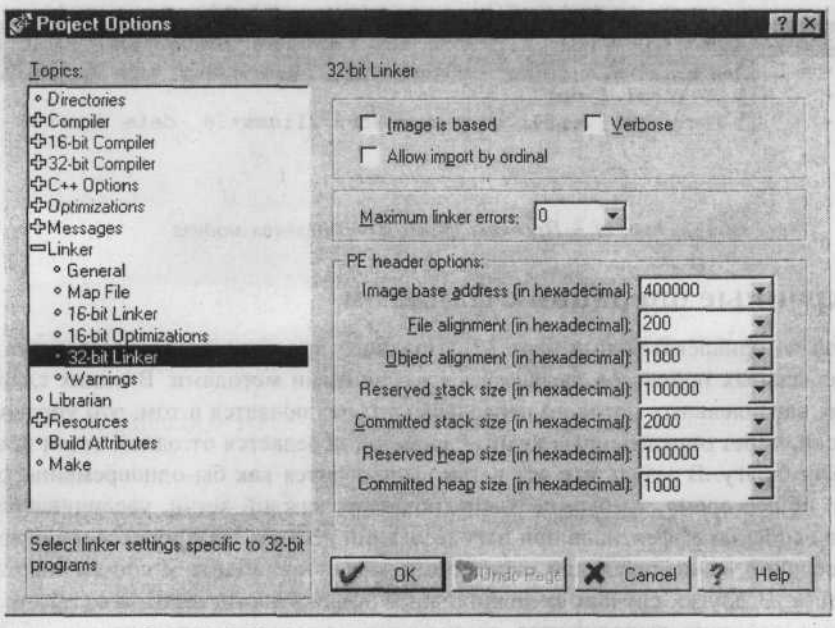


Рис. 11.2. Характеристики программы по умолчанию

Видно, что по умолчанию для стека резервируется 0x100000 = 1 Мбайт, причем реально отображено на память всего 0x2000 = 8 Кбайт, а остальная зарезервированная память будет отображаться автоматически по мере надобности. Для того чтобы наша программа работала правильно, в окне редактирования Reserved stack size (in hexadecimal) следует установить, например, число 8000000.

Мы при подготовке программы использовали другой метод – указали требуемый размер стека в собственном файле определения модуля 11-1.DEF:

```
/*Файл 11-1.def*/
EXETYPE WINDOWS
CODE PRELOAD MOVEABLE DISCARDABLE
DATA PRELOAD MOVEABLE
STACKSIZE 10000000
```

Для того чтобы не рассуждать над составом этого файла, можно просто скопировать в каталог с проектом файл DEFAULT.DEF из каталога BC5\LIB (переименовав его при этом в 11-1.DEF) и включить в него предложение

```
STACKSIZE 10000000
```

(число 10000000 здесь является десятичным и обозначает 10 миллионов, что несколько превышает потребности конкретной задачи).

После включения в состав проекта файла 11-1.DEF проект примет вид, изображенный на рис. 11.3.

Из широко используемых операций с файлами следует еще упомянуть операцию копирования файла, выполняемую с помощью функции CopyFile(), а также поиск нужной группы файлов по указанному шаблону групповой операции (типа *.DAT или FILES.00?), для чего предусмотрены функции FindFirstFile() и FindNextFile(). Эти весьма простые в использовании функции рассматривались ранее (см. программы 6-2 и 10-2) и мы к ним возвращаться не будем.

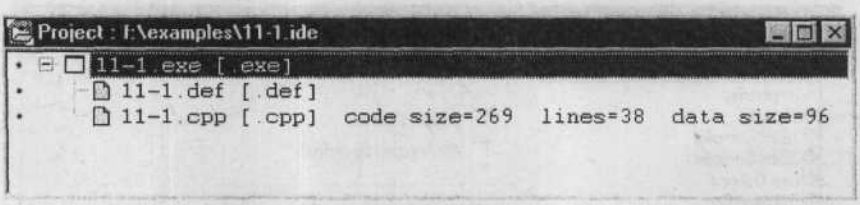


Рис. 11.3. Проект с файлом определения модуля

Асинхронные операции с файлами

Одной из привлекательных черт 32-разрядного программирования является концепция параллельных процессов, реализуемая различными методами. В одних случаях (организация параллельных потоков) параллельность заключается в том, что управление периодически, через определенные кванты времени, передается от одной ветви программы к другой по кругу. В результате все ветви выполняются как бы одновременно, хотя, разумеется, общее время, требуемое для выполнения каждой ветви, увеличивается. Такая методика особенно эффективна при использовании нескольких процессоров, хотя и в однопроцессорной вычислительной системе она может приводить к повышению производительности. В других случаях положительный эффект достигается за счет того, что параллельные действия выполняются различными элементами аппаратного обеспечения

компьютера. Так, чтение данных из файла – наиболее медленная операция в современных компьютерах – осуществляется в значительной степени аппаратурой контроллера диска, работающего в режиме прямого доступа. Занятость в этой операции процессора незначительна, однако в 16-разрядных приложениях программа, выдавшая запрос на чтение данных из файла (или запись данных в файл), останавливается до завершения этой операции, хотя в течение всего этого времени процессор почти свободен и мог бы выполнять полезную работу.

Такой режим выполнения программы, когда программа приостанавливается на время операции ввода-вывода, носит название синхронного или режима с ожиданием; если же программа, поставив запрос на ввод-вывод, может продолжить свое выполнение, то такой режим называется асинхронным или режимом с перекрытием.

В настоящем разделе будут рассмотрены основы организации асинхронного режима применительно к операциям с файлами.

Предупреждение. *Система Windows 98 не поддерживает асинхронных операций с файлами. Примеры этого раздела следует выполнять в системах Windows NT или Windows 2000.*

Приводимая ниже программа 11-2 по своей структуре весьма проста. Программа создает на диске файл с именем "TEST.DAT" и заполняет его тестовым массивом данных объемом 4 Мбайт (1 М целых чисел). Эта операция выполняется обычным синхронным образом. Далее тот же файл открывается повторно и программа ставит к системе запрос на асинхронное чтение из него всего массива данных. Пока длится чтение (при объеме файла 4 Мбайт на это требуется заметное время), программа продолжает выполнение, конкретно – читает и выводит в окно сообщения последний элемент тестового массива, предварительно инициализированный некоторым числом (12345678 в программе 11-2). Появление в окне сообщения именно этого числа говорит о том, что чтение элемента выполняется не после завершения операции ввода, а параллельно с ней. Спустя некоторое время тот же элемент массива читается и выводится в окно сообщения повторно. Теперь уже в нем содержится данное, прочитанное из файла (рис. 11.4).

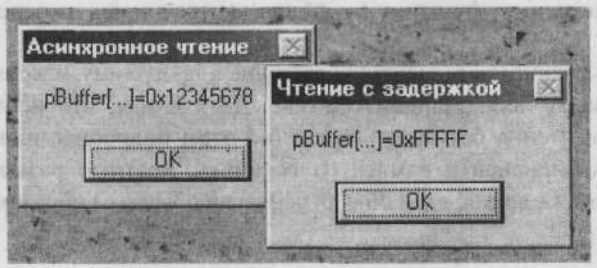


Рис. 11.4. Последовательные сообщения приложения 11-2

```
/*Программа 11-2. Асинхронные операции с файлом*/
#include <windows.h>
const int N=0x100000;//Количество данных в файле
/*Главная функция WinMain()*/
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
  char szFileName[]="Test.dat";//Имя создаваемого файла
  char szT[100];//Для формирования выводимого сообщения
  DWORD dwCount;//Для операций записи/чтения файла
```

```
  OVERLAPPED ovlp;//Структура поддержки асинхронных операций
  ZeroMemory(&ovlp,sizeof(ovlp));//Обнулим структурную переменную ovlp
/*Выделим память под буфер и заполним его*/
  PINT pBuffer=(PINT)VirtualAlloc(NULL,N*4,//Выделим память под буфер чтения
          MEM_RESERVE|MEM_COMMIT,PAGE_READWRITE);
  for(int i=0;i<N;i++)//Заполним буфер тестовыми данными
    pBuffer[i]=i;
/*Создадим файл и запишем в него тестовые данные*/
  HANDLE hFile=
        CreateFile(szFileName,GENERIC_READ|GENERIC_WRITE,//Создадим файл
        0,NULL,CREATE_ALWAYS,0,NULL);
  WriteFile(hFile,pBuffer,N*4,&dwCount,NULL);//Заполним файл
  ZeroMemory(pBuffer,N*4);//Очистка буфера
  CloseHandle(hFile);//Закроем файл, чтобы открыть иначе
/*Приступим к асинхронной операции*/
  hFile=CreateFile(szFileName,GENERIC_READ,0,NULL,//Откроем файл
      OPEN_EXISTING,FILE_FLAG_OVERLAPPED|//для асинхронных операций
      FILE_FLAG_NO_BUFFERING,NULL);//без буферизации
  pBuffer[N-1]=0x12345678;//Инициализируем последний элемент буфера чтения
  ReadFile(hFile,pBuffer,N*4,&dwCount,&ovlp);//Читаем файл
/*Последующие действия выполняются параллельно с чтением файла!*/
  wsprintf(szT,"pBuffer[...]=%d",pBuffer[N-1]);//Выведем последнее данное
  MessageBox(NULL,szT,"Асинхронное чтение",MB_OK);//из буфера чтения
/*К этому моменту файл наверняка прочитан*/
  wsprintf(szT,"pBuffer[...]=0x%X",pBuffer[N-1]);//Повторно выведем то же
  MessageBox(NULL,szT,"Чтение с задержкой",MB_OK);//данное
  VirtualFree(pBuffer,0,MEM_RELEASE);//Освободим память
  CloseHandle(hFile);//Закроем файл
  return 0;
  }
```

Объявленная в начале программы константная переменная N характеризует объем сохраняемых в файле данных и, соответственно, размер всего файла (N*4 байт). Структурная переменная ovlp типа OVERLAPPED используется при организации асинхронных операций (не обязательно с файлами). Назначение этой переменной будет описано ниже. Некоторые ее элементы используются системой, другие устанавливаются программой. В нашем случае с помощью функции ZeroMemory() все элементы переменной ovlp обнуляются.

Определенные сложности вызывает включение в программу массива для чтения в него содержимого файла асинхронным образом. Дело в том, что использованный нами в программе режим чтения без буферизации (об этом подробнее ниже) требует, чтобы буфер чтения был выровнен в памяти на величину, кратную размеру сектора диска, т. е. 512 байтам. В то же время, если объявить массив обычным образом

```
int nArray[0x100000];
```

то его адрес, назначаемый компилятором, может оказаться каким угодно. Поэтому буфер чтения выделяется в памяти динамически с помощью функции VirtualAlloc(). Особенностью этой функции является то, что она, выделяя программе участок памяти из объема свободного адресного пространства, обеспечивает выравнивание этого участка на границу 64 Кбайт; память выделяется страницами по 4 Кбайт. Через параметры функции VirtualAlloc() передается необходимая для ее работы информация. Прототип функции VirtualAlloc() выглядит следующим образом:

```
LPVOID VirtualAlloc(
  LPVOID lpvAddress,//Адрес резервируемого или передаваемого региона
  DWORD cbSize,//Размер региона
  DWORD fdwAllocationType,//Тип выделения памяти
  DWORD fdwProtect//Способ доступа к региону
  );
```

С помощью первого параметра lpvAddress можно указать желательный линейный адрес начала выделяемого участка, однако обычно поиск свободной памяти возлагается на систему и в качестве первого параметра указывается NULL.

Параметр cbSize задает размер в байтах заказываемой памяти.

Третий параметр fdwAllocationType определяет тип выделения. Выделение памяти действующему процессу состоит из двух этапов. Сначала в адресном пространстве процесса нужно найти свободный участок и зарезервировать его за процессом. Вспомним, что 32-разрядные приложения работают в плоской памяти и им в принципе доступны все 4 Гбайт линейного адресного пространства, так что скорее всего свободный участок найдется. Выделяемый участок иногда называют регионом памяти. Требование выделить программе новый регион указывается с помощью флага MEM_RESERVE.

Второй этап заключается в отображении на выделенный участок линейных адресов физической памяти. Как мы увидим в дальнейшем, в Win32 понятие физической памяти включает в себя не только установленную на компьютере оперативную память, но также и страничный файл, что дает возможность работать с массивами данных, значительно превышающими по размеру наличную оперативную память. Сейчас, однако, это обстоятельство не имеет для нас особого значения; для простоты можно считать, что отображается реально существующая оперативная память. Операция отображения физической памяти на зарезервированный регион линейного адресного пространства носит название передачи памяти (committing). Требование передать программе физическую память для выделенного региона указывается с помощью флага MEM_COMMIT в третьем параметре функции VirtualAlloc().

Последний параметр функции VirtualAlloc() определяет способ доступа к выделенной памяти. Для нас естественно указать доступ для чтения и записи (константа PAGE_READWRITE).

В случае своего успешного выполнения функция VirtualAlloc() возвращает линейный адрес выделенного участка (обобщенного типа LPVOID). Мы преобразуем его в тип PINT (указатель на целое) и сохраняем в переменной pBuffer. Заметим, что выделенной памятью мы пользуемся сначала для формирования записываемого в файл массива (для этой операции не требуется какое-либо выравнивание адреса памяти), а потом для чтения содержимого файла без буферизации (здесь выравнивание уже требуется).

Следующая операция после выделения памяти и заполнения ее тестовыми данными (попросту натуральным рядом чисел) заключается в создании файла и записи в него тестового массива. Эта операция выполняется точно так же, как и в предыдущем примере. Далее для порядка очищается буфер с адресом pBuffer и созданный ранее файл закрывается функцией CloseHandle(), чтобы затем снова открыть его уже в другом режиме – для чтения (флаг GENERIC_READ), только при его наличии (флаг OPEN_EXISTING), для асинхронных операций (флаг FILE_FLAG_OVERLAPPED) и с отказом от буферизации (флаг FILE_FLAG_NO_BUFFERING).

Чем нам помешала буферизация, т. е. кеширование файла в процессе его чтения? Дело в том, что, приступая к операции чтения файла, система сначала смотрит, не был ли этот файл

(или его участок) прочитан ранее. Если указанный участок файла уже содержится в системном буфере, операция физического чтения диска не выполняется; в программный буфер просто копируется содержимое системного буфера. В нашем случае такое вполне разумное действие системы приведет к тому, что после первого прогона программы она будет работать как запланировано, однако при последующих запусках система будет обнаруживать содержимое файла в системном буфере и никакого чтения из файла не будет, а следовательно, не будет и асинхронной операции. Отказ от буферизации позволил нам продемонстрировать интересующее нас явление в чистом виде.

После открытия файла в последнее слово буфера чтения помещается контрольное число 0x12345678 и запускается асинхронная операция чтения всего файла в тот же буфер. Обратите внимание на последний параметр функции ReadFile(). При чтении (или записи) в асинхронном режиме в качестве этого параметра должен быть указан адрес структурной переменной типа OVERLAPPED.

Поскольку операция чтения у нас асинхронная, физическая передача данных с диска идет параллельно с выполнением программы. Файл длинный, и, пока данные пересылаются в буфер чтения, программа успевает прочитать последнее, еще не затертое слово из буфера и вывести его в окно сообщения.

Как известно, окно сообщения блокирует дальнейшее выполнение программы до нажатия клавиши OK. Пока мы выполняем это действие, операция чтения скорее всего успевает завершиться, поэтому повторное обращение к тому же слову буфера читает из него уже не контрольное число 0x12345678, а данное, полученное из файла (0xFFFFF).

После освобождения выделенной ранее памяти функцией VirtualFree() и повторного закрытия файла программа завершается.

В рассмотренной программе имеется еще одна скрытая особенность. Обычно чтение из файла осуществляется начиная с байта, на который указывает файловый указатель. Сразу после открытия файла в указателе 0, и первое чтение осуществляется с начала файла. Однако это правило не действует для асинхронных операций. Если в качестве последнего параметра функций чтения или записи выступает адрес структурной переменной типа OVERLAPPED, то позиция, с которой следует начать чтение, берется из этой структуры, которая определена в файле WINBASE.H следующим образом:

```
typedef struct _OVERLAPPED {
  DWORD  Internal;
  DWORD  InternalHigh;
  DWORD  Offset;
  DWORD  OffsetHigh;
  HANDLE hEvent;
  } OVERLAPPED, *LPOVERLAPPED;
```

Первые два элемента предназначены для использования системой, а третий и четвертый как раз и определяют начальное 64-разрядное смещение в файле. Таким образом, если бы мы хотели прочитать не весь файл, а, например, его вторую половину, то запрос на чтение в программе выглядел бы следующим образом:

```
ovlp.Offset=N*2;//Устанавливаем позицию начала чтения в файле
ReadFile(hFile,pBuffer,N*2,&dwCount,&ovlp);//Читаем N*2 байт из файла
```

Рассмотренный пример составлен весьма рискованно в том отношении, что в нем не предусмотрено никаких средств синхронизации двух протекающих в программе процессов: выполнения программных кодов и чтения данных из файла. Если бы в программе не

выводились на экран сообщения, чтение которых пользователем требует значительного времени, то скорее всего программа успела бы освободить выделенную память и завершиться еще до того, как все данные прочитаны в буфер. Использование асинхронных операций почти неминуемо приводит нас к необходимости синхронизации процессов, идущих в программе.

Различные методы синхронизации процессов будут подробно рассмотрены в последующих главах этой книги. Там же мы расскажем и о многочисленных объектах Windows, которые можно использовать в целях синхронизации. Таких объектов довольно много: процессы, потоки, события, файлы, семафоры, мьютексы и др. Здесь мы проиллюстрируем лишь один, наиболее простой и естественный метод синхронизации, а именно использование в качестве объекта синхронизации дескриптора нашего файла.

Для объектов, которые могут служить целям синхронизации, в системе Windows предусматриваются два состояния – свободное и занятое. Занятое состояние ассоциируется с запретом на выполнение некоторых действий (главным образом выполнения синхронизируемого участка программы); свободное состояние, наоборот, разрешает выполнение.

Применительно к дескриптору файла, если он используется для целей синхронизации, дело обстоит следующим образом.

Функции ReadFile() и WriteFile() начинают свою работу с того, что сбрасывают дескриптор открытого файла в занятое состояние. Анализ состояния дескриптора (как и любого другого синхронизирующего объекта) осуществляется с помощью функции WaitForSingleObject(); эта функция устанавливается перед тем участком программы, который должен получить право на выполнение только после завершения операции с файлом. В результате программа, выполнив какую-то работу одновременно с операцией ввода-вывода, "застревает" затем на функции WaitForSingleObject(), которая блокирует выполнение программы в ожидании окончания ввода-вывода. Завершение файловой операции сбрасывает дескриптор файла в свободное состояние, в результате чего функция WaitForSingleObject() снимает блокировку и программа свободно продолжает свое выполнение.

Продемонстрировать описанную методику очень легко с помощью модификации рассмотренной выше программы 11-2. Достаточно сразу после вызова функции ReadFile() включить предложение

```
WaitForSingleObject(hFile,INFINITE);
```

В этом варианте первое сообщение (рис. 11.5) будет выведено на экран лишь после завершения операции чтения, о чем свидетельствует значение последнего слова массива 0xFFFFF. Фактически операция ввода и оставшаяся часть программы будут выполняться в синхронном режиме, друг за другом, так как перед функцией WaitForSingleObject() в программе не предусмотрено никаких действий, которые бы выполнялись параллельно с вводом.

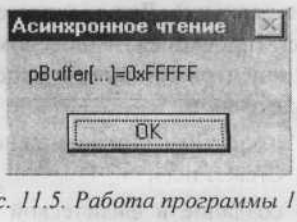


Рис. 11.5. Работа программы 11-2 в синхронном режиме

Файлы, проецируемые в память

Как известно, процессор не может работать с содержимым файла, находящегося на диске, – для обращения к какому-либо участку файла этот участок необходимо сначала прочитать в память. Если же требуется не просто обработать данные из файла, а модифицировать его содержимое, то к операции чтения файла добавляется еще и операция его записи. Поэтому стандартная для 16-разрядных приложений процедура работы с дисковым файлом (в наиболее полном виде, когда требуется модификация файла) включает по меньшей мере 4 этапа:

- открытие файла на диске;
- чтение требуемого участка файла в созданную заранее программную переменную;
- модификация этой программной переменной по заданному алгоритму;
- запись модифицированной переменной на то же место в файле на диске.

Обычно работа с файлом требует обращения не к одному его участку, а к нескольким, и тогда перечисленные выше операции приходится выполнять повторно. Во многих случаях, особенно при обслуживании обширных баз данных, программа по ходу своего выполнения запрашивает различные участки файла сотни и тысячи раз, и если эти запросы происходят не упорядоченно, а вразбивку, как это обычно и бывает, то каждое обращение к файлу требует выполнения операций физического чтения или записи, что существенно замедляет скорость выполнения программы.

Значительно более эффективным было бы чтение *всего* файла в оперативную память. В этом случае потребуется только одна операция физического чтения в начале сеанса работы и одна операция физической записи в конце сеанса. Однако такая методика возможна только с файлами относительно небольшого размера; к тому же в этом случае резко снижается надежность системы, так как новые, модифицированные данные записываются на диск только в конце сеанса и любой сбой программы может привести к потере всех модификаций.

Win32 предоставляет чрезвычайно удобный механизм работы с файлом, вообще исключающий (с точки зрения программиста) операции физического чтения или записи. Сущность этого механизма, называемого проецированием файла в память (file mapping) заключается в том, что конкретный файл включается в состав физической памяти, а затем в адресном пространстве приложения резервируется регион достаточного размера и часть физической памяти, включающая в себя файл, отображается на этот регион (передается региону). Дальнейшие операции с файлом заменяются операциями с его отображением (проекцией) в адресном пространстве приложения.

Процедура создания и использования проекции файла (рис. 11.6) распадается на несколько этапов:

- открытие (или, возможно, создание) файла обычным образом с помощью функции CreateFile();
- создание в физической памяти объекта "проекция файла" с помощью функции CreateFileMapping();
- создание в виртуальной (линейной) памяти приложения региона адресного пространства и отображение созданной проекции на адресное пространство процесса с помощью функции MapViewOfFile();

- обращение (чтение или запись) по адресам памяти, на которые отображена проекция, так же, как это выполняется для любых программных переменных.

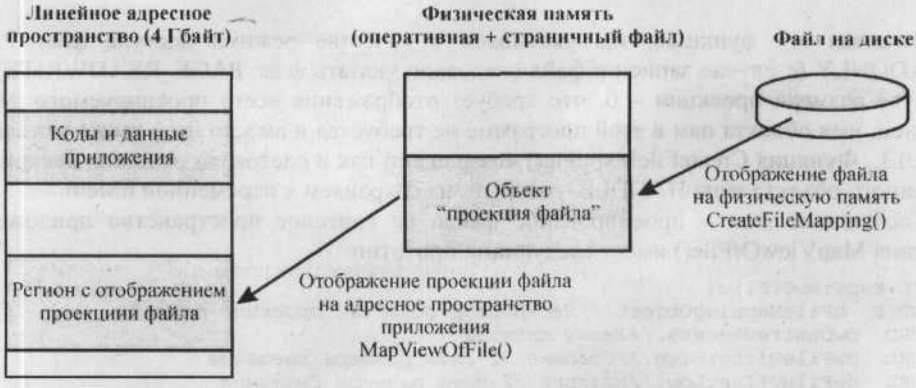


Рис. 11.6. Процесс проецирования файла в память

Рассмотрим методику проецирования файла в память и работы с его проекцией с помощью простого примера, в котором выполняется чтение отдельных записей созданного нами ранее (см. программу 11-1) файла STOCK.DAT, содержащего сведения об образцах изделий, хранящихся на складе.

```
/*Программа 11-3. Чтение файла, спроецированного в память*/
#include <windows.h>
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
  struct GOODS{//Структура, описывающая записи файла
    int Nmb;//Номер изделия
    int Price;//Его цена
    };
  char szT[100];
  HANDLE hFile=CreateFile("stock.dat",GENERIC_READ,//Открываем
              0,NULL,OPEN_EXISTING,0,NULL);//имеющийся файл
/*Создание в памяти объекта "проекция файла"*/
  HANDLE hMem=CreateFileMapping(hFile,NULL,PAGE_READONLY,0,0,NULL);
/*Отображение проекции файла на адресное пространство процесса*/
  GOODS *ptr=(GOODS*)MapViewOfFile(hMem,FILE_MAP_READ,0,0,0);
/*Чтение участков файла*/
  wsprintf(szT,"Изделие № %d\nЦена %d рублей\n-------------\n"
               "Изделие № %d\nЦена %d рублей",
               ptr[0].Nmb,ptr[0].Price,ptr[999999].Nmb,ptr[999999].Price);
  MessageBox(NULL,szT,"Info",MB_OK);
  UnmapViewOfFile(ptr);//Уничтожим отображение файла на адресное пространство
  CloseHandle(hMem);//Закроем проекцию файла
  CloseHandle(hFile);//Закроем файл
  return 0;
  }
```

Имеющийся на диске файл STOCK.DAT открывается обычным образом с указанием в качестве режима доступа константы GENERIC_READ, так как в этом примере предполагается только чтение файла без его модификации.

Функция CreateFileMapping() имеет следующий прототип:

```
HANDLE CreateFileMapping(
  HANDLE  hFile,//Дескриптор проецируемого файла
  LPSECURITY_ATTRIBUTES  lpFileMappingAttributes,//Атрибуты защиты
  DWORD  flProtect,//Режим доступа к проекции файла
```

```
  DWORD  dwMaximumSizeHigh,//Старшие 32 бита размера проекции
  DWORD  dwMaximumSizeLow,//Младшие 32 бита размера проекции
  LPCTSTR  lpName//Имя объекта "проекция файла"
  );
```

Вызывая эту функцию, мы указываем в качестве режима доступа флаг PAGE_READONLY (с случае записи в файл следовало указать флаг PAGE_READWRITE), а в качестве размера проекции – 0, что требует отображения всего проецируемого файла. Наконец, имя объекта нам в этой программе не требуется и вместо него также указывается NULL. Функция CreateFileMapping() возвращает, как и следовало ожидать, дескриптор созданного объекта типа HANDLE, который мы сохраняем в переменной hMem.

Следующий этап – проецирование файла на адресное пространство приложения. Функция MapViewOfFile() имеет следующий прототип:

```
LPVOID MapViewOfFile(
  HANDLE  hFileMappingObject,//Дескриптор объекта "проекция файла"
  DWORD  dwDesiredAccess,//Режим доступа
  DWORD  dwFileOffsetHigh,//Старшие 32 бита размера смещения
  DWORD  dwFileOffsetLow,//Младшие 32 бита размера смещения
  DWORD  dwNumberOfBytesToMap//Число отображаемых байтов
  );
```

Как видно из прототипа, функция MapViewOfFile() позволяет при необходимости проецировать в память любую часть файла, для чего следует указать смещение проецируемого участка от начала файла, а также размер этого участка. Если проецируется весь файл (как в нашем примере), на месте всех этих трех параметров указываются нули. Использованный в примере режим доступа FILE_MAP_READ обеспечивает чтение файла; при необходимости записывать в файл следовало указать флаг FILE_MAP_ALL_ACCESS.

Функция MapViewOfFile() возвращает обобщенный указатель (типа LPVOID) на выделенный регион адресного пространства. Мы сохраняем его в переменной ptr, предварительно преобразовав в тип GOODS*, соответствующий типу записей, из которых состоит файл STOCK.DAT.

Дальнейшие операции с содержимым файла осуществляются через указатель ptr на его проекцию в памяти так, как если бы весь файл уже содержался в памяти в виде массива с начальным адресом ptr. Если бы файл состоял из скалярных переменных, например целых чисел, для обращения к ним следовало использовать конструкцию ptr[0], ptr[1] и т. д. Наш файл состоит из структур, и мы обращается к отдельным элементам этих структур по обычным правилам с использованием индекса, указывающего номер записи и оператора "точка" для указания конкретного элемента структурной переменной: ptr[0].Nmb, ptr[0].Price, ptr[999999].Nmb, ptr[999999].Price.

На рис. 11.7 показано окно сообщения, возникающее при выполнении приложения 11-3.

Таким образом, после проецирования файла в память к нему можно обращаться как к обычной программной переменной, не пользуясь операциями чтения или записи файла. Однако очевидно, что в действительности файл, чтобы к его содержимому можно было обратиться, должен быть предварительно прочитан в память.

Рис. 11.7. Чтение записей из файла, спроецированного в память

Загрузку в оперативную память проецированных файлов полностью берет на себя система, освобождая от этого программиста. При этом, если размер файла велик и весь он в память не помещается, система обеспечивает последовательную загрузку тех его участков, к которым обращается программа. В программе же файл, независимо от его размера, выглядит как полностью находящийся в памяти массив данных.

Любая модификация записей спроецированного файла приводит к автоматическому переносу модифицированного участка на диск, что также обеспечивается системой без вмешательства программиста. Рассмотрим программу, в которой спроецированный в память файл не только читается, но и модифицируется. Для этого проще всего воспользоваться предыдущим примером, внеся в него незначительные изменения.

```
/*Программа 11-4. Модификация файла, спроецированного в память*/
#include <windows.h>
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
  struct GOODS{//Структура, описывающая записи файла
    int Nmb;
    int Price;
    };
  char szT[100];
  HANDLE hFile=CreateFile("stock.dat",//Откроем файл с доступом
                    GENERIC_READ|GENERIC_WRITE,//для чтения и записи
                    0,NULL,OPEN_EXISTING,0,NULL);
/*Создание в памяти объекта "проекция файла"*/
  HANDLE hMem=CreateFileMapping(hFile,NULL,PAGE_READWRITE,0,0,NULL);
/*Отображение проекции файла на адресное пространство процесса*/
  GOODS *ptr=(GOODS*)MapViewOfFile(hMem, FILE_MAP_ALL_ACCESS,0,0,0);
/*Запись в файл*/
  ptr[0].Price=15;//Изменим запись № 0
  ptr[999999].Price=512;//Изменим запись № 999999
/*Чтение записей файла с преобразованием в символы и помещением их в символьную
строку*/
  wsprintf(szT,"Изделие № %d\nЦена %d рублей\n-------------\n"
               "Изделие № %d\nЦена %d рублей",
               ptr[0].Nmb,ptr[0].Price,ptr[999999].Nmb,ptr[999999].Price);
  MessageBox(NULL,szT,"Info",MB_OK);
  UnmapViewOfFile(ptr);//Уничтожим отображение файла на адресное пространство
  CloseHandle(hMem);//Закроем проекцию файла
  CloseHandle(hFile);//Закроем файл
  return 0;
  }
```

Во всех трех функциях – CreateFile(), CreateFileMapping() и MapViewOfFile() – установлен режим доступа, разрешающий как чтение, так и запись. После завершения процедуры проецирования файла выполняется запись новых значений в его первый и последний элементы. Вывод этих значений в окно сообщения показывает, что модификация выполнена успешно (рис. 11.8).

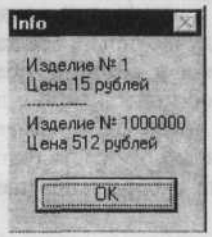


Рис. 11.8. Записи спроецированного файла, прочитанные после их модификации

Как и в предыдущем примере, перед завершением программы выполняются действия по освобождению использованных системных ресурсов: уничтожается отображение файла на адресное пространство приложения и закрываются дескрипторы файла и его проекции.

Использование страничного файла

Выше уже не раз отмечалось, что в Win32 под физической памятью понимается сумма оперативной памяти, установленной на данном компьютере, и страничного файла (рис. 11.9).

Файл подкачки, или страничный файл, создается системой при ее установке на компьютере и в разных вариантах Windows выглядит по-разному. В системах Windows NT и Windows 2000 этот файл имеет имя PAGEFILE.SYS и занимает определенную часть свободного дискового пространства (которую можно при необходимости изменить с помощью Панели управления). В системе Windows 98 страничный файл создается системой динамически по мере необходимости и также динамически уничтожается. При необходимости страничные файлы могут быть созданы не только на системном логическом диске, но и на остальных, что ускоряет работу приложений, использующих большие объемы данных.

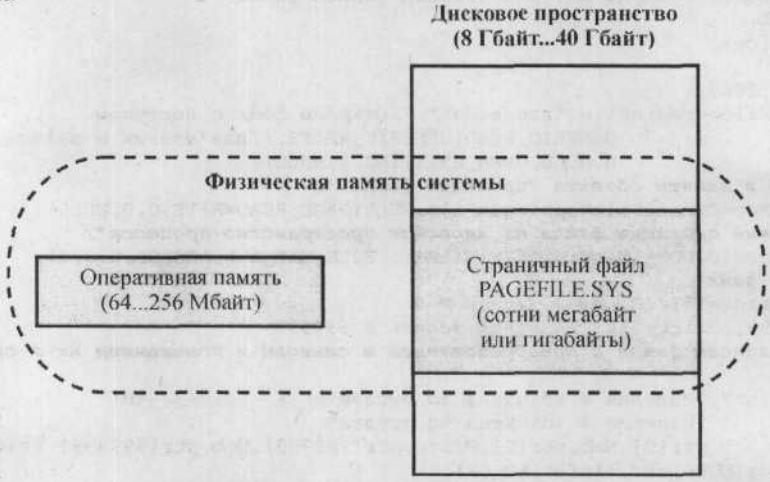


Рис. 11.9. Физическая память в Win32

Система использует страничный файл для хранения в нем приложений или их частей, временно выгруженных из оперативной памяти с целью предоставления памяти для других, активных в настоящий момент приложений. Фактически страничный файл вступает в игру в том случае, когда все запущенные приложения не помещаются в наличной оперативной памяти и система, обеспечивая их поочередное выполнение, вынуждена выполнять перенос приложений с диска в память и назад. Свое название страничный файл получил из-за того, что он имеет страничную структуру. Обмен данными между этим файлом и памятью осуществляется теми же самыми страницами по 4 Кбайт, которые используются для отображения памяти на адресное пространство выполняемых приложений (см. гл. 1).

Явное обращение к страничному файлу используется в прикладных программах в тех случаях, когда требуется организовать массив данных, общий для нескольких приложений. Этот вопрос будет рассмотрен в гл. 13. Здесь же мы приведем пример программы, использующей страничный файл неявным образом как часть физической памяти системы.

Пусть нам требуется создать (программным образом) и обработать большой массив данных, объем которого превышает оперативную память. Такая задача может возник-

нуть, в частности, при моделировании природных или социальных процессов. Учитывая, что в физическую память входит не только оперативная память, но и страничный файл, и имея в виду возможность настройки страничного файла с целью увеличения его размера, мы в принципе можем работать с массивом, близким по своему объему к свободному дисковому пространству. Приведем пример программы, обрабатывающей массив размером 128 Мбайт (будем считать, что программа выполняется на компьютере, оснащенном памятью объемом 64 Мбайт).

```
/*Программа 11-5. Массив данных, превышающий размер оперативной памяти*/
#include <windows.h>
const int N=0x2000000;//0x2000000 = 32 M данных * 4 байта = 128 Мбайт
int wArray[N];//Объявляем массив размером 128 Мбайт
char szT[80];
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
  for(int i=0;i<N;i++)//Заполним массив тестовыми данными
    wArray[i]=i+1;
  wsprintf(szT,"Первый элемент = %d\nПоследний элемент = %d",
           wArray[0],wArray[N-1]);
  MessageBox(NULL,szT,"Заполнение массива",MB_ICONINFORMATION);
  return 0;
  }
```

Вывод программы показан на рис. 11.10. При наличии в массиве 32 К последовательных чисел, начинающихся с единицы, последнее число действительно должно иметь значение 33554432.

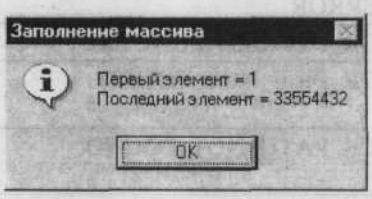


Рис. 11.10. Работа с массивом объемом 128 Мбайт

Отладка программ, использующих сложные системные средства

Отладка программ, насыщенных специфическими функциями Win32, в частности предназначенными для работы с файлами и их проекциями, может представить значительные сложности. Как было показано в гл. 2, обычная методика отладки программ заключается в их пошаговом выполнении под управлением отладчика с контролем внешних результатов выполнения функций Windows и других операторов программы. В ряде случаев полезную информацию могут предоставить возвращаемые функциями Windows значения. Часто, однако, такая методика не дает удовлетворительных результатов. Действительно, многие функции Windows возвращают лишь булеву величину, нулевое значение которой свидетельствует об ошибке при выполнении данной функции, однако не дает никакой информации о причине ошибки. Кроме того, программы, содержащие, например, асинхронные операции или параллельные процессы, в среде отладчика могут работать просто неверно.

Дополнительную информацию о причине ошибки при обращении к тому или иному системному средству позволяет получить специальная функция GetLastError(), которая возвращает (в виде значения типа DWORD) код ошибки, образованный при выполнении последней вызванной функции Windows. Отладка программы облегчается, если после каждой "подозрительной" функции Windows содержится вызов функции GetLastError():

```
DWORD dwError;
...
```

```
HANDLE hFile=CreateFile(...);
dwError=GetLastError();
...
HANDLE hMem=CreateFileMapping(hFile,...);
dwError=GetLastError();
```

В статьях интерактивного справочника, посвященных конкретным функциям Windows, описываются условия, при которых данная функция устанавливает свои коды ошибок, и в очень немногих случаях указаны символические обозначения этих кодов. Определения большей части кодов ошибок собраны в файле WINERROR.H, где можно найти их символические обозначения, числовые значения и краткие комментарии. В табл. 11.1 приведена информация о некоторых кодах ошибок, чаще других встречающихся в прикладных программах общего назначения.

Таблица 11.1. Некоторые коды ошибок, возникающие при выполнении функций Windows

Символическое обозначение кода ошибки	*Числовое значение*	*Описание*
ERROR_SUCCESS	0L	Успешное завершение
NO_ERROR	0L	То же
ERROR_INVALID_FUNCTION	1L	Неверная функция
ERROR_FILE_NOT_FOUND	2L	Система не может найти указанный файл
ERROR_PATH_NOT_FOUND	3L	Система не может найти указанный путь
ERROR_TOO_MANY_OPEN _FILES	4L	Система не может открыть файл
ERROR_ACCESS_DENIED	5L	Доступ запрещен
ERROR_INVALID_HANDLE	6L	Неверный дескриптор
ERROR_INVALID_ACCESS	12L	Неправильный код доступа
ERROR_INVALID_DATA	13L	Неправильные данные
ERROR_OUTOFMEMORY	14L	Для завершения этой операции не хватает памяти
ERROR_INVALID_DRIVE	15L	Система не может найти указанный дисковод
ERROR_NOT_SAME_DEVICE	17L	Система не может переместить файл на другой диск
ERROR_NO_MORE_FILES	18L	Больше файлов нет
ERROR_WRITE_PROTECT	19L	Среда защищена от записи
ERROR_BAD_UNIT	20L	Система не может найти указанное устройство
ERROR_NOT_READY	21L	Устройство не готово
ERROR_BAD_COMMAND	22L	Устройство не распознает команду
ERROR_WRITE_FAULT	29L	Система не может записать на указанное устройство

Продолжение табл. 11.1

Символическое обозначение кода ошибки	*Числовое значение*	*Описание*
ERROR_READ_FAULT	30L	Система не может читать из указанного устройства
ERROR_GEN_FAILURE	31L	Устройство, подсоединенное к системе, не функционирует
ERROR_SHARING_VIOLATION	32L	Процесс не имеет доступа к файлу, так как он используется другим процессом
ERROR_LOCK_VIOLATION	33L	Процесс не имеет доступа к файлу, так как часть файла заблокирована другим процессом
ERROR_SHARING_BUFFER _EXCEEDED	36L	Слишком много разделяемых файлов
ERROR_HANDLE_EOF	38L	Достигнут конец файла
ERROR_HANDLE_DISK_FULL	39L	Диск полон
ERROR_NOT_SUPPORTED	50L	Запрос не поддерживается
ERROR_INVALID_PARAMETER	87L	Неправильный параметр
ERROR_NO_PROC_SLOTS	89L	Система не может запустить другой процесс в данное время
ERROR_TOO_MANY _SEMAPHORES	100L	Невозможно создать другой системный семафор
ERROR_EXCL_SEM_ALREADY _OWNED	101L	Исключительным семафором владеет другой процесс
ERROR_SEM_IS_SET	102L	Семафор установлен и не может быть закрыт
ERROR_TOO_MANY_SEM _REQUESTS	103L	Семафор не может быть снова установлен
ERROR_SEM_OWNER_DIED	105L	Предыдущий владелец этого семафора завершил свою работу
ERROR_DRIVE_LOCKED	108L	Диск используется или заблокирован другим процессом
ERROR_OPEN_FAILED	110L	Система не может открыть указанные файл или устройство
ERROR_BUFFER_OVERFLOW	111L	Имя файла слишком длинное
ERROR_DISK_FULL	112L	На диске не хватает места
ERROR_SEM_TIMEOUT	121L	Лимит времени семафора исчерпан
ERROR_INVALID_NAME	123L	Неправильный синтаксис имени файла, каталога или метки тома

Окончание табл. 11.1

Символическое обозначение кода ошибки	*Числовое значение*	*Описание*
ERROR_NEGATIVE_SEEK	131L	Попытка установить указатель в файле до его начала
ERROR_SEEK_ON_DEVICE	132L	Для указанного устройства или файла нельзя установить указатель
ERROR_TOO_MANY_TCBS	155L	Нельзя создать еще один поток
ERROR_BAD_PATHNAME	161L	Указанный путь недопустим
ERROR_MAX_THRDS_REACHED	164L	В системе нельзя создать новые потоки
ERROR_BUSY	170L	Запрошенный ресурс занят
ERROR_ALREADY_EXISTS	183L	Нельзя создать файл, если он существует
ERROR_INVALID_FLAG_NUMBER	186L	Переданный флаг неверен
ERROR_SEM_NOT_FOUND	187L	Семафор с указанным именем не найден
ERROR_DIRECTORY	267L	Неверное имя каталога
ERROR_NOT_OWNER	288L	Попытка освободить мьютекс, не имея на него права владения
ERROR_INVALID_ADDRESS	487L	Попытка доступа по недопустимому адресу
ERROR_ARITHMETIC _OVERFLOW	534L	Арифметический результат превысил 32 бита
ERROR_OPERATION_ABORTED	995L	Операция ввода-вывода завершена по завершении потока или по запросу приложения
ERROR_IO_INCOMPLETE	996L	Асинхронное событие ввода-вывода не в установленном состоянии
ERROR_IO_PENDING	997L	Асинхронная операция ввода-вывода выполняется
ERROR_NOACCESS	998L	Недопустимый доступ к ячейке памяти
ERROR_SWAPERROR	999L	Ошибка при выполнении чтения страницы
ERROR_STACK_OVERFLOW	1001L	Слишком глубокая рекурсия, стек переполнен
ERROR_CAN_NOT_COMPLETE	1003L	Функция не может быть завершена
ERROR_INVALID_FLAGS	1004L	Недопустимые флаги

Приведенный выше фрагмент отладочного варианта программы предполагает выполнение программы под управлением отладчика с остановкой после каждого предложения GetLastError() и анализом значения переменной dwError. Однако, как уже отмеча-

лось, многие программы нельзя запустить в отладчике. В этих случаях приходится включать в текст программы отладочные фрагменты, в которых результат, возвращаемый функцией getLastError(), а также, возможно, и самой "подозрительной" функцией, выводится на экран в виде окна сообщения:

```
DWORD dwError;
char szT[100];
...
HANDLE hFile=CreateFile(...);//Выполняем "подозрительную" операцию
dwError=GetLastError();//Получаем код ошибки
wsprintf(szT,"hFile=%X, dwError=%d", hFile,dwError);//Выводим то и другое
MessageBox(NULL,szT,"Создание файла",MB_OK);//в окно сообщения
...
HANDLE hMem=
        CreateFileMapping(hFile,...);//Выполняем "подозрительную" операцию
dwError=GetLastError();//Получаем код ошибки
wsprintf(szT,"hMem=%X, dwError=%d", hMem,dwError);//Выводим то и другое
MessageBox(NULL,szT,"Создание проекции файла",MB_OK);//в окно сообщения
```

Обратите внимание на формат выводимых чисел: возвращаемые функциями значения удобнее наблюдать в шестнадцатеричном виде, а коды ошибок – в десятичном, так как именно таким образом они определены в файле WINERROR.H (см. табл. 11.1).

Такой метод при всей его громоздкости чрезвычайно удобен, так как позволяет динамически по ходу программы получать информацию о ее выполнении и при этом еще изучать особенности работы программы в разных условиях или режимах. На рис. 11.11 показаны окна сообщений программы 11-4, в который включены приведенные выше отладочные фрагменты, в случае нормальной работы программы. Файл при его открытии функцией CreateFile() получил произвольный с нашей точки зрения дескриптор 0x2C, а его проекция – дескриптор 0x4C. Функция GetLastError() в обоих случаях вернула код 0 (NO_ERROR).

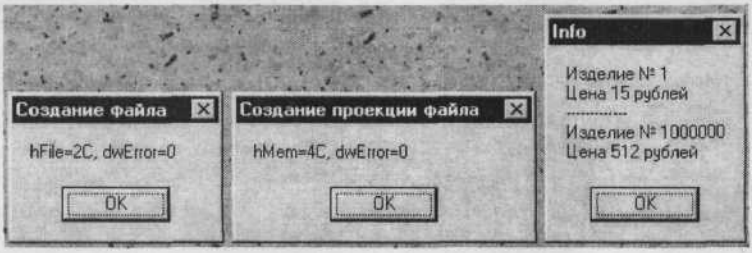


Рис. 11.11. Сеанс успешного выполнения программы с динамическим выводом сообщений об ошибках

Приведенные результаты были получены при запуске модифицированного варианта программы 11-4 в системе Windows NT4. Система Windows 95 ведет себя иначе: при успешном открытии имеющегося файла функция CreateFile() возвращает правильный дескриптор файла, который можно в дальнейшем использовать, однако функция GetLastError() сообщает о коде ошибки 6 (ERROR_INVALID_HANDLE).

На рис. 11.12 показаны окна сообщений того же примера, который был запущен при отсутствии на диске требуемого файла. Этот эксперимент также проводился в системе Windows NT4, хотя в Windows 95 получились бы такие же результаты. Функция CreateFile() вернула код 0xFFFFFFFF=-1 (INVALID_HANDLE_VALUE), а функция GetLastEr-

ror() правильно показала ошибку 2 (ERROR_FILE_NOT_FOUND). Функция CreateFile Mapping() в качестве кода ошибочного завершения вернула 0 (точнее говоря, NULL), а код ошибки оказался 87 (ERROR_INVALID_PARAMETER). Действительно, в качестве первого параметра этой функции было передано недопустимое значение дескриптора 0xFFFFFFFF.

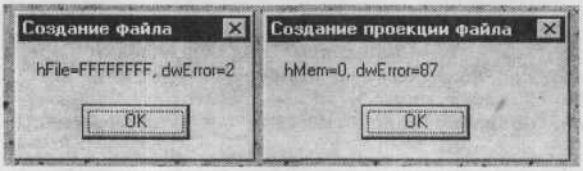


Рис. 11.12. Динамические сообщения об ошибках в программе при неправильном выполнении

Глава 12
Процессы и потоки

Общие понятия

Процессом называют экземпляр выполняемой программы – не ход выполнения программы, а именно саму программу вместе с ее ресурсами. В понятие процесса включают виртуальное адресное пространство, коды и данные, принадлежащие программе, а также такие системные ресурсы, как файлы, синхронизирующие объекты, библиотеки динамического связывания и др. Таким образом, процесс в Win32, в противоречии со смыслом этого слова, не является динамическим объектом – это скорее оболочка, набор ресурсов, которые нужны для выполнения программы.

Потоком называют ход выполнения программы. Поток выполняется в рамках владеющего им процесса или, как говорят, в контексте процесса. Именно потоку операционная система выделяет кванты процессорного времени. Поток выполняет программные коды, принадлежащие процессу. Когда операционная система запускает приложение, она создает первичный поток процесса, составляющего это приложение. Таким образом, любой процесс имеет по крайней мере один поток выполнения. Первичный поток может запустить дополнительные потоки, которые в программе оформляются как более или менее самостоятельные функции. Эти потоки будут выполняться параллельно, конкурируя за процессорное время. Довольно очевидно, что все потоки процесса разделяют виртуальное адресное пространство процесса, его глобальные переменные (потому что они доступны всем функциям), а также ресурсы операционной системы, используемые процессом.

В однопроцессорной системе все потоки процесса так же, как и потоки разных процессов, выполняются хотя и параллельно, но не одновременно, а по очереди, вытесняя друг друга по мере выделения им квантов процессорного времени. Однако на компьютере с несколькими процессорами операционная система Windows NT или более современная может закрепить за каждым процессором свой поток, и в этом случае потоки будут выполняться строго одновременно, что, естественно, повысит скорость выполнения приложения.

Итак, любое запущенное приложение представляет собой процесс, и в контексте этого процесса всегда выполняется по крайней мере один первичный поток, создаваемый системой. Если пользователь запускает несколько приложений, то в системе действуют столько же процессов и первичных потоков. Однако новый *процесс* может быть создан не только пользователем, но и текущим потоком. Такой процесс называют дочерним, хотя он совсем не обязательно должен играть подчиненную роль. Прикладная программа может запустить в качестве дочернего процесса любое приложение Windows – например, Калькулятор или Блокнот. Разумеется, для прикладных программных комплексов более полезным является запуск взаимосвязанных *прикладных* процессов, между которыми разумным образом разделены функции всего программного комплекса.

Если запуск дочерних процессов можно рассматривать как способ организации сложных программных комплексов, состоящих из ряда более или менее автономных программ, то создание в рамках одной программы нескольких потоков преследует главным образом цель повышения производительности вычислительной системы. Можно сфор-

мулировать ряд условий, при выполнении хотя бы одного из которых многопоточный режим приводит к благоприятным последствиям:

- В приложении имеются фрагменты, которые должны активизироваться периодически, по сигналам времени, что характерно, в частности, для программ, управляющих медицинским или физическим измерительным оборудованием. Выделение таких фрагментов в отдельные потоки позволит эффективно выполняться остальным частям программы.
- Фрагменты приложения должны активизироваться лишь при возникновения какого-либо события. Типичный пример – обработка разного рода сбоев и аварийных ситуаций, равно как и сигналов от измерительного оборудования или линий связи.
- Приложение получает данные от многих удаленных источников, например от других компьютеров. Закрепление за каждым источником данных отдельного потока позволит выполнять сбор данных параллельно, что особенно важно в тех случаях, когда каналы связи работают медленно или ненадежно и длительная задержка в одном из каналов может остановить работу всей системы.
- Приложение сочетает в себе медленный ввод данных, например с клавиатуры или по линиям связи, с вычислительной работой, не связанной непосредственно с текущим вводом. Выделение ввода данных в отдельный поток позволит не останавливать вычисления на время получения данных.
- В приложении выполняется ввод, обработка и визуальное представление некоторой информации, причем пользователю желательно следить за ходом обработки, чтобы иметь возможность вносить в работу программы необходимые коррективы. Разделение функций ввода и визуального представления по отдельным потокам позволит пользователю наблюдать частичные результаты обработки.
- Назначением приложения является моделирование каких-либо случайных процессов, например обслуживания клиентов в некоторой организации или развития популяции определенного вида животных в зависимости от внешних условий. Создание для каждого клиента или особи отдельного потока позволяет проследить поведение совокупности объектов в реальном времени при весьма эффективном использовании процессорного времени и других ресурсов компьютера.

К сожалению, реальные примеры приложений, в которых наглядно и эффективно используется многопоточная обработка, или непомерно сложны, или вообще не могут быть продемонстрированы на автономном компьютере (как, например, задачи по управлению производственным или научным оборудованием). Поэтому большинство приводимых ниже примеров будет поневоле носить формальный характер или являться крайним упрощением реальных задач. В основном мы будем ставить перед собой цель показать технику использования того или иного вычислительного средства, связанного с понятиями параллельного выполнения процессов и потоков. Все эти средства можно условно разбить на 3 группы:

- создание процессов и потоков;
- передача информации между процессами и потоками;
- временная и событийная синхронизация процессов и потоков.

В настоящей главе рассматриваются процедуры создания процессов и потоков; возможностям передачи информации и средствам синхронизации будут посвящены две следующие главы.

Создание процесса

Рассмотрим программу, которая запускает другую программу в качестве дочернего процесса. Чтобы подчеркнуть независимость процессов, сделаем обе программы по всем правилам, с главными окнами и циклами обработки сообщений. В программе родительского процесса предусмотрим еще линейку меню с пунктом "Запуск процесса". Назовем родительскую программу 12-1a, а дочернюю – 12-1b. Приведем сначала текст дочерней программы, в которой, впрочем, нет ничего нового или интересного.

```
/*Программа 12-1b. Дочерний процесс*/
/*Файл 12-1b.h*/
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
void OnPaint(HWND);
void OnDestroy(HWND);
/*Файл 12-1b.cpp*/
#include <windows.h>
#include <windowsx.h>
#include "12-1b.h"
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
void OnPaint(HWND);
void OnDestroy(HWND);
char szClassName[]="MainWindow";
char szTitle[]="Программа 12-1b";
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  MSG Msg;
  WNDCLASS wc;
  ZeroMemory(&wc,sizeof wc);
  wc.lpfnWndProc=WndProc;
  wc.hInstance=hInst;
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hbrBackground=GetStockBrush(LTGRAY_BRUSH);//Светло-серый цвет окна
  wc.lpszClassName=szClassName;
  RegisterClass(&wc);
  HWND hwnd=CreateWindow(szClassName,szTitle,WS_OVERLAPPEDWINDOW,
                         100,300,310,100,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_NORMAL);
  while(GetMessage(&Msg,NULL,0,0))
    DispatchMessage(&Msg);
  return 0;
  }
/*Оконная функция WndProc главного окна*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_PAINT,OnPaint);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
/*Функция обработки сообщений WM_PAINT*/
void OnPaint(HWND hwnd){
  PAINTSTRUCT ps;
  char szText[]="Простое приложение для всяких испытаний";
  HDC hdc=BeginPaint(hwnd,&ps);
  SetBkMode(hdc,TRANSPARENT);
  TextOut(hdc,5,30,szText,strlen(szText));
  EndPaint(hwnd,&ps);
```

```
  }
/*Функция обработки сообщения WM_DESTROY*/
void OnDestroy(HWND){
  PostQuitMessage(0);
  }
```

В приложении 12-1b на Рабочий стол Windows выводится главное окно приложения со строкой текста. Вывод в окно текста потребовал ввести в оконную функцию обработку сообщения WM_PAINT. Поскольку главное окно сделано светло-серым, перед выводом текста в функции OnPaint() устанавливается режим прозрачного фона.

Еще раз подчеркнем, что приложение, которое мы здесь называем дочерним, в действительности является обычным приложением Windows, которое можно запустить с помощью команды Пуск➢Выполнить или с Рабочего стола, если на нем создать ярлык, соответствующий этому приложению. Дочерний статус приложения создается не стилем его написания или содержанием, а исключительно способом запуска. В программе 12-1b по сравнению с обычными приложениями Windows нет *абсолютно никаких* особенностей.

Другое дело программа 12-1a, где по команде пользователя выполняется запуск дочернего процесса. Впрочем, в целом эта программа тоже имеет стандартный вид, и единственно интересным местом в ней является вызов функции CreateProcess() запуска (создания) дочернего процесса.

```
/*Программа 12-1a. Запуск дочернего процесса*/
/*Файл 12-1a.h*/
#define MI_NEW 100
#define MI_EXIT 101
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
void OnCommand(HWND,int,HWND,UINT);
void OnDestroy(HWND);
/*Файл 12-1a.rc*/
#include "12-1a.h"
Main MENU{
  POPUP "Файл"{
    MENUITEM "Запуск процесса",MI_NEW//Пункт меню для запуска процесса
    MENUITEM SEPARATOR
    MENUITEM "Выход",MI_EXIT
    }
  }
/*Файл 12-1a.cpp*/
#include <windows.h>
#include <windowsx.h>
#include "12-1a.h"
char szClassName[]="MainWindow";
char szTitle[]="Программа 12-1a";
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  MSG Msg;
  WNDCLASS wc;
  ZeroMemory(&wc,sizeof(wc));
  wc.lpfnWndProc=WndProc;
  wc.hInstance=hInst;
  wc.lpszMenuName="Main";//В главном окне будет меню
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);//Белый фон окна
  wc.lpszClassName=szClassName;
  RegisterClass(&wc);
```

```
  HWND hwnd=CreateWindow(szClassName,szTitle,WS_OVERLAPPEDWINDOW,
       10,10,320,120,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_NORMAL);
  while(GetMessage(&Msg,NULL,0,0)){
    DispatchMessage(&Msg);
    }
  return 0;
  }
/*Оконная функция WndProc главного окна*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg)  {
    HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
/*Функция обработки сообщений WM_COMMAND */
void OnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id){
    case MI_NEW:{//Выбран пункт меню "Запуск процесса"
      STARTUPINFO si;//Служебная структурная переменная
      PROCESS_INFORMATION pi;//Служебная структурная переменная
      ZeroMemory(&si,sizeof(si));//Очистим переменную si
      si.cb=sizeof(si);//Заносим в член si.cb размер структуры
      BOOL bSuccess=CreateProcess
                    ("12-1b.exe",NULL,NULL,NULL,FALSE,0,NULL,NULL,&si,&pi);
        if(bSuccess){//Если процесс успешно создан,
          CloseHandle(pi.hThread);//закроем дескриптор его первичного потока
          CloseHandle(pi.hProcess);//и дескриптор всего процесса
          }
      break;
      }
    case MI_EXIT:
      DestroyWindow(hwnd);
    }
  }
/*Функция обработки сообщения WM_DESTROY */
void OnDestroy(HWND){
  PostQuitMessage(0);
  }
```

Работа созданного нами программного комплекса продемонстрирована на рис. 12.1.

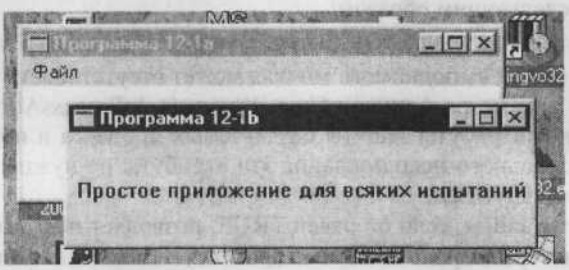


Рис. 12.1. Запуск дочернего процесса

Рассмотрим особенности запуска дочернего процесса. Для создания процесса используется функция CreateProcess() со следующим прототипом:

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,//Указатель на имя выполнимой программы
```

```
LPTSTR lpCommandLine,//Указатель на параметры командной строки
LPSECURITY_ATTRIBUTES lpProcessAttributes,//Указатель на атрибуты
                                          //защиты процесса
LPSECURITY_ATTRIBUTES lpThreadAttributes,//Указатель на атрибуты
                                         //защиты потока
BOOL bInheritHandles,//Флаг наследования
DWORD dwCreationFlags,//Флаги создания
LPVOID lpEnvironment,//Указатель на новый блок окружения
LPCTSTR lpCurrentDirectory,//Указатель на имя каталога
                           //создаваемого процесса
LPSTARTUPINFO lpStartupInfo,//Указатель на структуру STARTUPINFO
LPPROCESS_INFORMATION lpProcessInformation//Указатель на структуру
                                          //PROCESS_INFORMATION
);
```

Параметр lpApplicationName описывает имя выполнимого модуля создаваемого процесса. Если, как в нашем примере, в имени отсутствует путь, запускаемая программа ищется в текущем каталоге текущего диска. Имя модуля должно обязательно сопровождаться его расширением (по умолчанию расширение .EXE не подставляется).

Параметр lpCommandLine описывает строку с параметрами запуска, если создаваемому процессу надо передать параметры. В использовании параметра lpCommand Line имеется особенность: в дочерний процесс передается *второй* после пробела параметр. В качестве первого может использоваться имя выполнимого модуля, и в этом случае на месте параметра lpApplicationName указывается NULL. Таким образом, если требуется запустить программу 12-1B.EXE и передать ей в качестве параметра командной строки имя файла MYFILE.001, с которым она будет работать, то для этого можно использовать любой из следующих вариантов:

```
CreateProcess
      (NULL,"12-1b myfile.001",NULL,NULL,FALSE,0,NULL,NULL,&si,&pi);
CreateProcess
      ("12-1b.exe",12-1b myfile.001,NULL,NULL,FALSE,0,NULL,NULL,&si,&pi);
CreateProcess
      ("12-1b.exe",abc myfile.001,NULL,NULL,FALSE,0,NULL,NULL,&si,&pi);
```

В последнем варианте в командную строку пришлось вставить бессмысленный первый параметр "abc", чтобы сдвинуть на второе место передаваемый параметр myfile.001. На практике чаще всего используют первый из трех приведенных вариантов.

Учитывая, что имя запускаемой программы может передаваться функции CreateProcess() в качестве ее второго параметра, запуск процесса без параметров командной строки обычно выполняют следующим образом:

```
CreateProcess(NULL,"12-1b",NULL,NULL,FALSE,0,NULL,NULL,&si,&pi);
```

В этом случае в имени выполнимого модуля может отсутствовать расширение.

Следующие два параметра функции CreateProcess(), lpProcessAttributes и lpThreadAttributes, характеризуют атрибуты защиты создаваемых процесса и потока. В прикладных программах индивидуального использования эти атрибуты не нужны и вместо обоих параметров можно указать NULL.

Параметр bInheritHandles, если он равен TRUE, позволяет передать открытые в родительском процессе дескрипторы в дочерний процесс. Нам это не нужно, и на месте параметра указано FALSE.

Параметр dwCreationFlags позволяет указать набор флагов, управляющих созданием нового процесса. Эти флаги используются только в специальных случаях, например при запуске 16-разрядных приложений Windows или если выполнение первичного потока запускаемого процесса надо отложить до специального указания.

Параметр lpEnvironment позволяет создать и передать в дочерний процесс индивидуальный вариант окружения. В прикладных программах используется редко.

Параметр lpCurrentDirectory указывает на полную спецификацию рабочего каталога для дочернего процесса. При указании NULL в качестве рабочего каталога система использует каталог родительского процесса.

Через параметр lpStartupInfo функции CreateProcess передается адрес структурной переменной типа STARTUPINFO. Инициализируя отдельные элементы этой структуры, можно в определенной степени управлять видом главного окна запускаемого приложения, изменяя его размеры, положение на Рабочем столе, заголовок окна и т. д. Правда, для того чтобы такая настройка стала возможна, в дочернем приложении следует указать, что параметры окна определяются по умолчанию (а не путем задания фиксированных констант, как это делается во всех наших примерах). Далее, многие элементы структуры STARTUPINFO используются только при запуске консольных приложений, т. е. программ, написанных по правилам разработки приложений для MS-DOS, однако оформленных как приложения Windows. Однако даже если мы и не намереваемся настраивать дочернее приложение, структурная переменная типа STARTUPINFO в программе должна быть, причем всем ее членам следует присвоить нулевые значения. В программе 12-1a эта переменная названа si.

Функция CreateProcess(), выполнив свою работу, возвращает в вызывающую программу некоторые характеристики, конкретно – дескрипторы и идентификаторы, созданных процесса и потока. Эти характеристики заносятся в элементы структурной переменной типа PROCESS_INFORMATION, адрес которой передается функции CreateProcess() через ее последний параметр. Эта переменная, как и переменная типа STARTUPINFO, должна обязательно иметься в приложении.

Состав структуры PROCESS_INFORMATION виден из ее объявления, содержащегося в файле WINBASE.H:

```
typedef struct _PROCESS_INFORMATION {
  HANDLE hProcess;//Дескриптор созданного процесса
  HANDLE hThread;//Дескриптор первичного потока созданного процесса
  DWORD dwProcessId;//Уникальный глобальный идентификатор созданного процесса
  DWORD dwThreadId;//Такой же идентификатор первичного потока
}PROCESS_INFORMATION, *LPPROCESS_INFORMATION;
```

Если функция CreateProcess() выполнилась успешно, то, создав дочерний процесс и запустив выполнение его первичного потока, она возвращает значение TRUE, в противном случае – FALSE.

Вернемся к предложениям программы 12-1a, в которых создается дочерний процесс:

```
BOOL bSuccess=CreateProcess("12-
1b.exe",NULL,NULL,NULL,FALSE,0,NULL,NULL,&si,&pi);
if(bSuccess){//Если процесс успешно создан,
  CloseHandle(pi.hThread);//закроем дескриптор его первичного потока
  CloseHandle(pi.hProcess);//и дескриптор всего процесса
  }
```

На первый взгляд может показаться странным, что, создав процесс и запустив его первичный поток, мы как бы сразу их уничтожаем. В действительности вызовом функции CloseHandle() закрываются не созданные объекты, а лишь их дескрипторы, которыми после этой операции мы уже пользоваться не можем. Однако сами объекты "процесс" и "поток" остаются в памяти и нормально функционируют до своего завершения. Дескрипторы можно и не закрывать; завершение процесса (т. е. приложения), безусловно, уничтожит все использованные в процессе объекты и освободит от них память. Однако кор-

ректное закрытие дескрипторов, как только в них отпала необходимость (а у нас она и не возникала), несколько оптимизирует использование памяти.

Рассмотрим еще один пример запуска дочернего процесса. Пусть у нас есть программа, назначение которой – заполнение файлов некоторыми данными, например полученными программой из измерительной установки. Для того чтобы не запутаться потом с этими файлами, в каждом должна содержаться информация об имени оператора и номере эксперимента. Предусмотрим в программе такую последовательность действий:

- формирование массива экспериментальных данных;
- запуск в качестве дочернего процесса стандартного текстового редактора Windows Блокнот с передачей ему имени файла;
- ввод с клавиатуры и помещение в файл требуемых данных – имени оператора и номера эксперимента;
- ожидание отработки редактора Блокнот;
- дописывание в конец файла сформированных экспериментальных данных из программного массива.

Текст программы 12-2 состоит всего из нескольких строк, так как в ней отсутствуют окна и цикл обработки сообщений.

```
/*Программа 12-2. Передача командной строки в дочерний процесс*/
#include <windows.h>
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
  STARTUPINFO si;//Для создания процесса
  PROCESS_INFORMATION pi;//Для создания процесса
  BYTE Data[26];//Для данных
  DWORD dwCnt;//Для работы с файлом
  for(int i=0;i<26;i++)//Формирование массива
    Data[i]=(BYTE)(i+'A');//"экспериментальных" данных
  ZeroMemory(&si,sizeof(si));
  si.cb=sizeof(si);
  CreateProcess
        (NULL,"notepad.exe f:\\examples\\12-2.txt",//Открываем Блокнот,
         NULL,NULL,FALSE,0,NULL,NULL,&si,&pi);//передаем имя файла
  WaitForSingleObject(pi.hProcess,INFINITE);//Будем ждать завершения
                                            //программы Блокнот
  HANDLE hFile=CreateFile("12-2.txt",GENERIC_READ|GENERIC_WRITE,//Откроем файл
              0,NULL,OPEN_EXISTING,0,NULL);
  SetFilePointer(hFile,0,0,FILE_END);//Сдвинемся к концу файла
  WriteFile(hFile,Data,26,&dwCnt,NULL);//Запись данных
  CloseHandle(hFile);//Закрываем дескриптор файла
  return 0;
  }
```

Будем считать, что перед запуском программы мы создали заранее текстовый файл с именем 12-2.TXT, содержащий всего две строки:

```
Оператор:
Серия:
```

Программа формирует массив данных (для простоты и наглядности массив данных в примере состоит из букв латинского алфавита) и создает процесс с именем выполнимого файла NOTEPAD.EXE. В качестве параметра командной строки ему передается спецификация файла 12-2.TXT. В результате на экран выводится рабочий кадр программы Блокнот с уже открытым файлом 12-2.TXT, куда можно вписать имя оператора и номер серии эксперимента (рис. 12.2).

Существенным моментом рассматриваемой программы является необходимость синхронизации двух работающих процессов. После запуска программы Блокнот оба процесса выполняются параллельно в соответствии с выдачей им операционной системой квантов времени. Если не предусмотреть приостановку нашей программы, то она, возможно, завершится еще до того, как Блокнот успеет активизироваться и открыть файл данных. Может быть и другой вариант: Блокнот не сможет открыть файл данных, так как он еще используется нашей программой. В любом случае программный комплекс будет работать не так, как требуется. Для того чтобы после открытия программы Блокнот остановить исходный процесс, используется функция WaitForSingleObject(), с которой мы уже сталкивались в гл. 11. Там синхронизирующим объектом служил дескриптор открытого файла; здесь мы используем дескриптор открытого процесса, который приходится извлекать из структуры PROCESS_ INFORMATION.

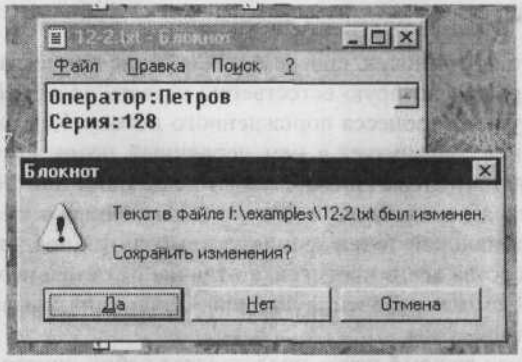


Рис. 12.2. Наше приложение запустило программу Блокнот, оператор вписал в файл необходимые данные и дал команду закрытия программы Блокнот

При создании нового процесса его дескриптор переводится системой в занятое (сброшенное) состояние, запрещая тем самым дальнейшее выполнение фрагмента программы, "защищенного" функцией WaitForSingleObject(). Завершение программы NOTEPAD.EXE устанавливает дескриптор этого процесса в свободное состояние, разрешая дальнейшее выполнение нашей программы. В ней снова открывается файл 12-2.TXT, его указатель устанавливается в позицию 0 байт от конца файла и от позиции указателя в файл дописываются "экспериментальные" данные. Установка указателя на конец файла гарантирует правильное дописывание данных независимо от того, какой длины текстовые строки мы добавили в этот файл на этапе его редактирования в программе Блокнот. В результате файл 12-2.TXT принимает вид, показанные на рис. 12.3.

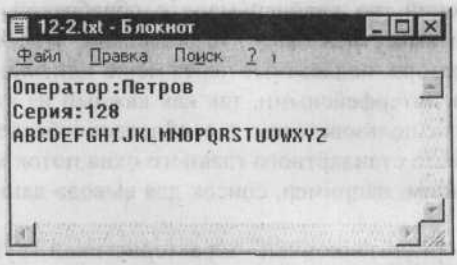


Рис. 12.3. Файл 12-2.TXT после действия программы 12-2

Заметим, что в приведенном примере было бы недопустимо закрыть дескриптор процесса сразу после его создания, как это мы сделали в программе 12-1а. Действительно, дескриптор процесса используется затем функцией WaitForSingleObject() в целях синхронизации. Освобождение дескриптора процесса сделало бы синхронизацию процессов невозможной. А вот дескриптор первичного потока программы Блокнот можно и освободить: он нами не используется. Точно так же нельзя было освободить дескриптор открытого файла до окончания всех операций с файлом, которые используют этот дескриптор, конкретно – вызова функций SetFilePointer() и WriteFile().

Рассмотренный пример показывает, что динамическое создание процессов в выполняемой программе почти неминуемо приводит к необходимости использования средств синхронизации. Этому довольно сложному вопросу будет посвящена гл. 14

Создание потока

Итак, процесс – это, в сущности, программа. Создание процесса обозначает запуск из текущей программы другой, которую естественно назвать дочерней. Рассмотрим теперь создание в рамках текущего процесса порожденного потока. Как уже отмечалось, система, запуская приложение, организует в нем первичный поток выполнения. Этот поток может с помощью функции CreateThread() создать еще один или несколько потоков, которые будут выполняться параллельно в контексте владеющего ими процесса (запущенной программы). С формальной точки зрения каждый поток представляет собой одну из функций, входящих в состав всего процесса; в отличие от остальных эту функцию можно назвать функцией потока или рабочей функцией. Однако организационно потоки могут принимать самые разные формы.

В простейшем случае каждый поток представляет собой просто подпрограмму, которая выполняет некоторую вычислительную работу, например решает свою систему уравнений. Получаемые ими результаты потоки могут сохранять в общей памяти процесса или выводить в окно процесса. Все системы уравнений, если их несколько, решаются параллельно и в известной степени одновременно. При необходимости можно организовать передачу данных из потока в поток (вспомним, что для всех потоков доступны все глобальные переменные процесса) или синхронизировать выполнение потоков.

Другой способ организации потоков заключается в том, что каждому из них "выдается" свое окно и, соответственно, внутри каждого потока предусматривается свой цикл обработки сообщений. В этом случае потоки ведут более самостоятельный образ жизни – каждый из них может получать от пользователя предназначенные для него данные и выводить в свое окно результаты работы. Следует подчеркнуть, что наличие у потока (неважно, первичного или вторичного, порожденного) окна требует также и организации цикла обработки сообщений, по крайней мере с обработкой сообщения WM_PAINT (иначе как можно что-то вывести в окно?) и, возможно, WM_COMMAND, чтобы воспринимать команды оператора, подаваемые через меню или элементы управления. Потоки такого рода называют интерфейсными, так как каждый из них может независимо от других взаимодействовать с пользователем, т. е. обладает собственным интерфейсом.

В частном случае вместо стандартного главного окна поток может ограничиться диалоговым окном, содержащим, например, список для вывода данных или окно редактирования для их ввода.

Еще одной важной "организационной" характеристикой группы порожденных потоков является степень их независимости. Каждый из запущенных потоков может иметь собственную функцию обработки сообщений и свой собственный, отличный от других

потоков алгоритм выполнения (свою собственную функцию потока). Однако очень часто, например в задачах моделирования или обработки данных, несколько потоков должны иметь весьма схожие алгоритмы, хотя и призваны обрабатывать различающиеся, индивидуальные данные. В таком случае все порожденные потоки могут иметь одну и ту же функцию обработки сообщений и один и тот же алгоритм. Это, однако, не помешает каждому потоку обрабатывать собственные данные и выводить результаты в собственное окно.

Рассмотрим несколько программ, иллюстрирующих порождение потоков с разной организацией. Начнем с формального примера создания в рамках процесса некоторого количества однородных потоков, характеризующихся единой функцией потока, хотя и несколько различающихся деталями своего поведения.

Потоки с общей рабочей функцией

```
/*Программа 12-3. Создание однородных интерфейсных потоков*/
/*Файл 12-3.h*/
#define MI_EXIT 101
#define MI_NEW 102
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
void OnCommand(HWND,int,HWND,UINT);
void OnDestroy(HWND);
void OnPaint(HWND);
DWORD WINAPI Thread(LPVOID);
LRESULT CALLBACK ThreadWndProc(HWND,UINT,WPARAM,LPARAM);
void ThreadOnDestroy(HWND);
void ThreadOnPaint(HWND);
```

```
/*Файл 12-3.rc*/
#include "12-3.h"
Main MENU {
  POPUP "Файл" {
    MENUITEM "Запуск потока",MI_NEW
    MENUITEM SEPARATOR
    MENUITEM "Выход",MI_EXIT
  }
}
```

```
/*Файл 12-3.cpp */
#include <windows.h>
#include <windowsx.h>
#include "12-3.h"
/*Глобальные переменные*/
char szClassName[]="MainWindow";//Имя класса главного окна
char szTitle[]="Программа 12-3";//Заголовок главного окна
char szThreadClassName[]="Thread";//Имя класса окна порождаемых потоков
char szThreadTitle[]="Дочерний поток";//Заголовок окон потоков
HINSTANCE hInstance;//Дескриптор экземпляра приложения
HANDLE hThread;//Дескриптор создаваемого потока
DWORD dwIDThread;//Идентификатор создаваемого потока
int nCount;//Счетчик потоков
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  MSG Msg;
  hInstance=hInst;//Сохраним дескриптор экземпляра приложения
/*Зарегистрируем класс главного окна первичного потока*/
  WNDCLASS wc;
  ZeroMemory(&wc,sizeof(wc));
  wc.lpfnWndProc=WndProc;//Оконная функция главного окна
  wc.hInstance=hInst;
  wc.lpszMenuName="Main";
```

```
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);//Цвет окна белый
  wc.lpszClassName=szClassName;//Имя класса главного окна
  RegisterClass(&wc);//Регистрируем класс главного окна
/*Зарегистрируем класс окна порожденного потока*/
  wc.lpfnWndProc=ThreadWndProc;//Общая оконная функция всех потоков
  wc.lpszMenuName=NULL;//Меню у потоков не будет
  wc.hbrBackground=GetStockBrush(LTGRAY_BRUSH);//Цвет окон светло-серый
  wc.lpszClassName=szThreadClassName;//Имя класса окон потоков
  wc.cbWndExtra=16;//Для номера потока и времени запуска
  RegisterClass(&wc);//Регистрируем класс окон
/*Создадим и покажем главное окно*/
  HWND hwnd=CreateWindow(szClassName,szTitle,WS_OVERLAPPEDWINDOW,
       10,10,320,150,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_NORMAL);
  while(GetMessage(&Msg,NULL,0,0))//Цикл обработки сообщений
    DispatchMessage(&Msg);//главного окна (первичного потока)
  return 0;
  }
/*Оконная процедура главного окна (первичного потока)*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg)  {
    HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);
    HANDLE_MSG(hwnd,WM_PAINT,OnPaint);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
    return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
/*Функция обработки сообщений WM_PAINT для главного окна*/
void OnPaint(HWND hwnd){
  char szT[30];
  PAINTSTRUCT ps;
  HDC hdc=BeginPaint(hwnd,&ps);
  wsprintf(szT,"Запущен поток #%d",nCount);
  if(hThread)//Если хоть один поток запущен
    TextOut(hdc,160,1,szT,strlen(szT));//Вывести текст в главное окно
  EndPaint(hwnd,&ps);
  }
/*Функция обработки сообщений WM_COMMAND для главного окна*/
void OnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id){
    case MI_NEW:
      nCount++;//Счетчик потоков
      hThread=CreateThread
              (NULL,0,Thread,(LPVOID)nCount,0,&dwIDThread);//Создаем поток
      CloseHandle(hThread);//Закрываем дескриптор потока
      InvalidateRect(hwnd,NULL,TRUE);//Обновить главное окно
      break;
    case MI_EXIT:
      DestroyWindow(hwnd);
    }
  }
/*Функция обработки сообщения WM_DESTROY для главного окна*/
void OnDestroy(HWND){
  PostQuitMessage(0);
  }
```

```
/*Общая для всех потоков рабочая функция потока*/
DWORD WINAPI Thread(LPVOID nCnt){
  MSG Msg;//Структурная переменная для сообщений потоков
  SYSTEMTIME SysTime;//Структурная переменная для местного времени
  HWND hwnd=CreateWindowEx(WS_EX_TOPMOST,"Thread",//Создадим окно потока
            "Порожденный поток",WS_OVERLAPPEDWINDOW,//Заголовок, стиль
             nCount*40+50,nCount*60+16,220,100,//Сдвиг окон потоков
             HWND_DESKTOP,NULL,hInstance,NULL);
  ShowWindow(hwnd,SW_NORMAL);//Покажем окно потока
  SetWindowLong(hwnd,0,(LONG)nCnt);//Сохраним номер потока
  GetLocalTime(&SysTime);//Получим местное время
  SetWindowLong(hwnd,4,(LONG)SysTime.wHour);//Сохраним часы
  SetWindowLong(hwnd,8,(LONG)SysTime.wMinute);//Сохраним минуты
  SetWindowLong(hwnd,12,(LONG)SysTime.wSecond);//Сохраним секунды
  while(GetMessage(&Msg,NULL,0,0))//Цикл обработки сообщений
    DispatchMessage(&Msg);//для окон всех потоков
  return 0;
  }
/*Оконная процедура для окон потоков*/
LRESULT CALLBACK ThreadWndProc
     (HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg)  {
    HANDLE_MSG(hwnd,WM_PAINT,ThreadOnPaint);
    HANDLE_MSG(hwnd,WM_DESTROY,ThreadOnDestroy);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
/*Функция обработки сообщений WM_PAINT для окон потоков*/
void ThreadOnPaint(HWND hwnd){
  char szT[100];
  PAINTSTRUCT ps;
  HDC hdc=BeginPaint(hwnd,&ps);
  SetBkMode(hdc,TRANSPARENT);//Сделаем прозрачным фон символов
  wsprintf(szT,"Выполняется поток #%d",
           GetWindowLong(hwnd,0));//Заберем номер потока
  TextOut(hdc,5,1,szT,strlen(szT));//Вывод строки
  wsprintf(szT,"Время запуска %d:%d:%d",
           GetWindowLong(hwnd,4),//Заберем часы
           GetWindowLong(hwnd,8),//Заберем минуты
           GetWindowLong(hwnd,12));//Заберем секунды
  TextOut(hdc,5,16,szT,strlen(szT));//Вывод в строки окно
  EndPaint(hwnd,&ps);
  }
/*Функция обработки сообщения WM_DESTROY для окон потоков*/
void ThreadOnDestroy(HWND){
  PostQuitMessage(0);
  }
```

В главном окне приложения предусмотрено меню с двумя пунктами: "Запуск потока" и "Выход" (рис. 12.4). С помощью команды Файл➤Запуск потока можно создать любое количество порожденных потоков, характеризующихся общей функцией потока Thread() и общей оконной процедурой ThreadWndProc(). Функция потока создает и делает видимым окно данного потока, после чего запускает цикл обработки сообщений. В случае создания нескольких потоков все они разделяют общий цикл обработки сообщений, который по очереди обслуживает все запущенные потоки.

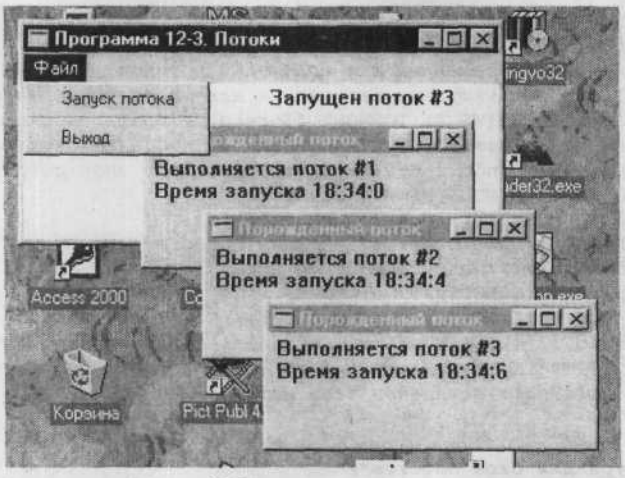


Рис. 12.4. Главное окно приложения 12-3 с раскрытым меню и окнами порожденных потоков

В рассматриваемом примере создаваемые потоки не выполняют никакой полезной работы. Однако каждый из них все же что-то делает, конкретно – выводит в собственное окно свой номер и время своего запуска. Нам предстоит, во-первых, выяснить, как создаются потоки и, во-вторых, каким образом отдельные потоки, имея общую оконную функцию и соответственно общие функции обработки сообщений, тем не менее могут выполнять индивидуальную работу.

Поток создается вызовом функции CreateThread(), имеющей следующий прототип:

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,//Указатель на атрибуты защиты потока
  DWORD dwStackSize,//Начальный размер стека потока
  LPTHREAD_START_ROUTINE lpStartAddress,//Адрес функции потока
  LPVOID lpParameter,//Аргумент, передаваемый новому потоку
  DWORD dwCreationFlags,//Флаги создания
  LPDWORD lpThreadId//Адрес переменной для возврата идентификатора потока
  );
```

Параметр lpThreadAttributes характеризует атрибуты защиты создаваемого потока. Как уже отмечалось, в прикладных программах индивидуального использования значение этого параметра может быть NULL.

Параметр dwStackSize определяет объем адресного пространства, выделяемый под стек потока. Каждому потоку выделяется свой стек, и по умолчанию (если значение параметра dwStackSize равно нулю) размер стека определяется из настроек компоновщика.

Параметр lpParameter поступает в функцию потока при ее активизации, и через него потоку можно передать любые данные в виде 32-разрядного числа. При необходимости передать потоку набор данных в качестве параметра lpParameter указывается адрес структуры или массива данных.

Параметр lpStartAddress задает адрес функции потока, которая, собственно, и определяет, какую работу будет выполнять данный поток. В нашем примере функция потока создает при каждом своем вызове новое окно потока, после чего входит в бесконечный цикл обработки сообщений. Фактически, поскольку потоков у нас несколько, а функция потока одна, ее совместно используют все созданные потоки по мере выделения им квантов процессорного времени.

Указав в качестве параметра dwCreationFlags значение CREATE_SUSPENDED, можно создать "замороженный" поток, который начнет выполняться только в результате вызова функции ResumeThread(). При нулевом значении параметра поток начинает выполняться немедленно после создания.

Последний параметр lpThreadId представляет собой адрес переменной типа DWORD, в которую функция CreateThread() в случае своего успешного завершения вернет назначенный потоку идентификатор.

При успешном выполнении функция CreateThread() возвращает дескриптор потока типа HANDLE, а в случае ошибки – значение NULL.

Функция CreateThread() в программе 12-3 вызывается в ответ на выбор пользователем пункта меню "Запуск потока". Через параметр lpParameter этой функции потоку передается предварительно инкрементированное значение счетчика потоков. Таким образом, при каждом вызове функции потока в нее поступает индивидуальный номер этого потока.

В нашем примере все потоки работают после своего создания самостоятельно, без какого-либо взаимодействия или синхронизации. Поэтому нам не нужен ни идентификатор потока, ни его дескриптор, который мы сразу же после создания потока закрываем, вызвав функцию CloseHandle(). Тем самым предотвращается размножение дескрипторов потоков по мере их создания. Далее вызовом функции InvalidateRect() в главное окно приложения посылается сообщение WM_PAINT. В инициируемой в этом случае функции OnPaint() в главное окно выводится строка с порядковым номером только что созданного потока. В целях благообразия эта строка выводится лишь в том случае, если глобальная переменная hThread, служащая для приема дескрипторов создаваемых потоков, приняла ненулевое значение, т. е. только после создания хотя бы одного потока.

Рассмотрим теперь вопрос о придании экземплярам потоков индивидуальных черт. Эта проблема возникает всякий раз, когда создается набор окон одного класса и требуется как-то различать эти окна. Если количество окон фиксировано, то возможным решением является сохранение в глобальном массиве их дескрипторов, которые для разных окон будут, разумеется, различаться. Однако у нас количество окон не определено заранее, и этот способ неудобен. Можно поступить по-другому. В Windows предусмотрена возможность при регистрации класса окна зарезервировать для *каждого* создаваемого окна этого класса определенное пространство во внутренней структуре Windows, относящейся к окну. Объем резервируемого пространства указывается в элементе cbWndExtra структуры WNDCLASS. В нашем примере резервируются 16 байт (для четырех 32-разрядных переменных).

Запись требуемых значений в резервную область окна осуществляется с помощью функции SetWindowLong(), а чтение – функцией GetWindowLong(). И той и другой функции указывается смещение интересующей нас переменной в резервной области. И записываются и считываются 32-разрядные числа.

Посмотрим, как выглядит функция потока Thread(). В ней прежде всего вызовом расширенной функции CreateWindowEx() создается окно очередного потока с дополнительным стилем WS_EX_TOPMOST, определяющим вывод окна на передний план (иначе новые окна прячутся под уже созданные и работать с ними неудобно). В качестве координат окна указываются значения, зависящие от номера окна nCount, что позволяет

сдвигать окна друг относительно друга. Затем вызовом функции ShowWindow() окно делается видимым.

При вызове функции потока Thread() через ее единственный параметр мы передаем номер очередного окна (и потока). Это значение (локальная переменная nCnt) сохраняется вызовом функции SetWindowLong() в первой переменной резервной области окна. Передаваемый параметр в соответствии с прототипом функции имеет обобщенный тип LPVOID, а функция SetWindowLong() принимает параметр типа LONG, поэтому приходится использовать явное преобразование типа. Далее читается системное время и значения часов, минут и секунд сохраняются в остальных трех переменных резервной области. Еще раз подчеркнем, что для *каждого* окна класса "Thread" создается своя резервная область и в каждой из них будет содержаться время создания конкретного окна.

Создав окно и заполнив его резервную область, функция Thread() входит в цикл обработки сообщений GetMessage() – DispatchMessage(), ожидая сообщений для любого из созданных к этому моменту окон.

Для того чтобы в окна создаваемых потоков можно было выводить какие-либо изображения, в частности текстовые строки, в оконной процедуре потока предусмотрена обработка сообщений WM_PAINT. В соответствующей этим сообщениям функции ThreadOnPaint() из резервной области окна извлекаются с помощью функции GetWindowLong() относящиеся к данном потоку значения и формируются две символьные строки – с номером данного потока и временем его создания.

В рассмотренном примере первичный поток с обычным интерфейсом (главным окном и циклом обработки сообщений) создавал некоторое количество однородных потоков, каждый из которых тоже имел главное окно и цикл обработки сообщений, при этом все порожденные потоки управлялись общей рабочей функцией. Структура такой программы изображена на рис. 12.5.

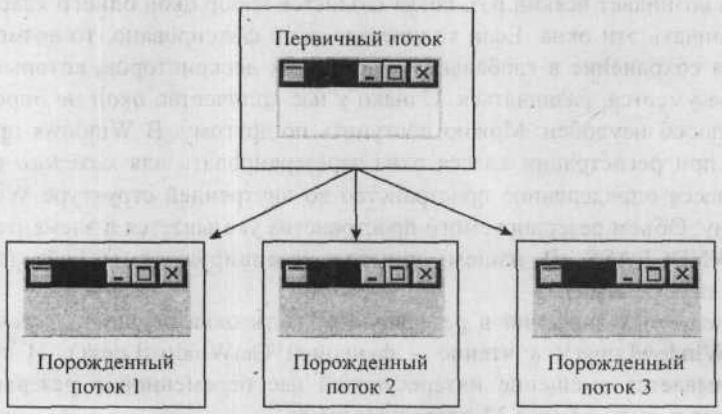


Рис. 12.5. Структура программы с множественными однородными потоками

Первичный поток не обязательно должен иметь главное окно; его интерфейс может, например, ограничиваться модальным диалогом. Интерфейсы порожденных потоков тоже могут принимать любую форму, и в частности, вполне можно представить себе порожденные потоки, носящие чисто вычислительный характер и не обладающие собственным интерфейсом (рис. 12.6). Рассмотрим пример программы такого рода.

Рис. 12.6. Другой вариант структуры программы с множественными однородными потоками

```
/*Программа 12-4. Создание однородных потоков без интерфейса*/
/*Файл 12-4.h*/
#define IDC_FILE 101
#define IDC_DATA 102
BOOL CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM);
BOOL DlgOnInitDialog(HWND,HWND,LPARAM);
void DlgOnCommand(HWND,int,HWND,UINT);
DWORD WINAPI ThreadProc(LPVOID);
struct DATA{//Структура для передачи данных в порожденные потоки
  char FileName[128];//Строка для имени файла
  char Data[50];//Строка для текстовых данных
  };
/*Файл 12-4.rc*/
#include "12-4.h"
INPUT DIALOG 0, 0, 185, 120
STYLE DS_MODALFRAME|WS_POPUP|WS_VISIBLE|WS_CAPTION|WS_SYSMENU
CAPTION "Ввод новых данных"
FONT 8, "MS Sans Serif" {
  CONTROL "OK",IDOK,"BUTTON",BS_PUSHBUTTON|BS_CENTER|WS_CHILD|
            WS_VISIBLE|WS_TABSTOP,64,90,50,14
  CONTROL "",IDC_FILE,"edit",ES_LEFT|WS_CHILD|WS_VISIBLE|WS_BORDER|
            WS_TABSTOP,20,26,140,13
  CONTROL "",IDC_DATA,"edit",ES_LEFT|WS_CHILD|WS_VISIBLE|WS_BORDER|
            WS_TABSTOP,19,65,140,13
  CONTROL "Имя файла",-1,"static",SS_LEFT|WS_CHILD|WS_VISIBLE,20,14,140,9
  CONTROL "Новые данные",-1,"static",SS_LEFT|WS_CHILD|WS_VISIBLE,20,53,140,9
  }
/*Файл 12-4.cpp*/
#include <windows.h>
#include <windowsx.h>
#include "12-4.h"
DATA dt;//Структура для обмена данными с потоками
```

```
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  DialogBox(hInst,"Input",NULL,DlgProc);//Открыть диалог
  return 0;
  }
/*Оконная функция диалогового окна*/
BOOL CALLBACK DlgProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg)  {
    HANDLE_MSG(hwnd,WM_INITDIALOG,DlgOnInitDialog);
    HANDLE_MSG(hwnd,WM_COMMAND,DlgOnCommand);
    default:
      return FALSE;
    }
  }
/*Функция обработки сообщения WM_INITDIALOG для диалогового окна*/
BOOL DlgOnInitDialog(HWND,HWND,LPARAM){
  return TRUE;
  }
/*Функция обработки сообщений WM_COMMAND для диалогового окна*/
void DlgOnCommand(HWND hwnd,int id,HWND,UINT){
  DWORD dwId;//Идентификатор потока
  switch(id){//Идентификатор элемента управления
    case IDOK://Нажата кнопка OK
      GetWindowText(GetDlgItem(hwnd,IDC_FILE),dt.FileName,128);//Получим имя файла
      GetWindowText(GetDlgItem(hwnd,IDC_DATA),dt.Data,50);//Получим данные
      SetWindowText(GetDlgItem(hwnd,IDC_FILE),"");//Очистим окно редактирования
      SetWindowText(GetDlgItem(hwnd,IDC_DATA),"");//Очистим окно редактирования
      CreateThread(NULL,0,ThreadProc,&dt,0,&dwId);//Создадим поток
      break;
    case IDCANCEL://Дана команда закрыть диалог через его системное меню
      EndDialog(hwnd,0);//Закрытие диалогового окна
    }
  }
/*Рабочая функция потока*/
DWORD WINAPI ThreadProc(LPVOID lpData){
  DWORD dwCnt;//Для работы с файлом
  HANDLE hFile=CreateFile(((DATA*)lpData)->FileName,//Откроем или
              GENERIC_WRITE,0,NULL,OPEN_ALWAYS,0,NULL);//создадим файл
  SetFilePointer(hFile,0,0,FILE_END);//Сдвинемся к концу файла
  WriteFile(hFile,((DATA*)lpData)->Data,//Дописывание введенной пользователем
           strlen(((DATA*)lpData)->Data),&dwCnt,NULL);//строки в файл
  CloseHandle(hFile);//Закроем дескриптор файла
  return 0;
  }
```

В файле ресурсов описан сценарий модального диалога с двумя окнами редактирования (рис. 12.7). Первое окно служит для ввода спецификации файла, второе – для строки данных, которая должна быть приписана в конец заданного файла. Общий ход программы выглядит следующим образом. Указав имя файла и записываемую в него строку, пользователь нажимает кнопку OK. Это приводит к созданию потока, рабочая функция которого открывает файл, записывает в его конец строку и завершается. Каждый новый ввод данных приводит к созданию очередного потока. Если предположить, что по тем или иным причинам доступ к файлам требует значительного времени или в действительности в файлы записывается большой объем информации, то пользователь успеет создать несколько потоков записи в разные файлы, которые будут, конкурируя друг с другом за процессорное время, выполняться одновременно.

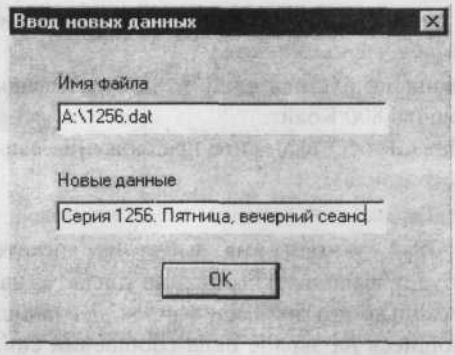


Рис. 12.7. Диалоговое окно первичного потока программы 12-4

В заголовочном файле 12-4.H, помимо констант и прототипов функций, описана структура DATA, включающая два символьных массива – для спецификации файла и для выводимой в файл строки. Глобальная переменная dt типа DATA, объявленная в начале программы до главной функции WinMain(), будет использована для приема данных, вводимых пользователем с помощью окон редактирования.

Главная функция WinMain() состоит, в сущности, из одного предложения – активизации модального диалога, описанного в файле ресурсов. После того как пользователь ввел в окна редактирования необходимые данные и нажал кнопку OK, управление передается функции DlgOnCommand(). Здесь с помощью функции GetWindowText() содержимое окон редактирования переносится в два элемента структурной переменной dt, сами окна редактирования очищаются и вызовом функции CreateThread() создается порожденный поток с рабочей функцией ThreadProc(). При создании потока через четвертый параметр функции CreateThread() потоку передается адрес структурной переменной dt, хранящей нужные потоку данные.

В функции ThreadProc() открывается файл с указанным пользователем именем. Флаг открытия (OPEN_ALWAYS) выбран так, что, если такой файл уже имеется, он открывается для записи. Если файла нет, он создается заново. Далее указатель в файле смещается к его концу и введенная пользователем строка из переменной dt дописывается в файл. После закрытия дескриптора файла данный экземпляр порожденного потока прекращает свое существование.

Предупреждение. *Система Windows 98 не поддерживает асинхронных операций с файлами. Описываемый ниже эксперимент следует выполнять в системах Windows NT или Windows 2000. В Windows 98 работа программы будет заблокирована до завершения записи на дискету, в результате чего наглядно наблюдать параллельную работу потоков не удастся.*

Для того чтобы убедиться в параллельном выполнении нескольких экземпляров порожденных потоков, мы рекомендуем провести некоторую модификацию приведенной программы, сделав ее еще менее осмысленной, но более наглядной.

В функцию ThreadProc() включите объявление относительно большого массива

```
char nBuf[200000];//Массив из 800000 байт
```

Для большей наглядности можно инициализировать массив какими-нибудь числами:

```
for(int i=0;i<200000;i++)
  nBuf[i]=1;//Заполним массив единицами
```

Предложение записи данных в файл перепишем следующим образом:

```
WriteFile(hFile,nBuf,800000,&dwCnt,NULL);
```

В этом случае в указанный пользователем файл будет выводиться содержимое массива nBuf, занимающего почти 800 Кбайт.

В начало функции ThreadProc() включите предложение вывода на экран окна сообщения с именем заказанного файла:

```
MessageBox(NULL,((DATA*)lpData)->FileName,"Info",MB_OK);
```

Запустив программу 12-4, укажем имя файла на дискете, как это показано на рис. 12.7. Пока система будет выполнять запись на дискету, на что потребуется значительное время, введем несколько других имен файлов для записи их *в текущий каталог текущего диска*. Появляющиеся на экране окна сообщений свидетельствуют о том, что все запущенные нами потоки выполняются одновременно. Можно даже успеть, пока еще не окончилась запись на дискету, посмотреть содержимое файлов, записанных на жесткий диск. Их имена были введены *после* имени файла на дискете, однако записаны на жесткий диск они будут *до* завершения работы с дискетой.

Потоки с индивидуальными рабочими функциями

В двух предыдущих примерах множество порожденных потоков имели общую рабочую функцию, хотя каждый поток выполнялся с некоторыми отличиями от других. Такая организация программ широко используется, например, при моделировании поведения объектов, имеющих одинаковые принципиальные свойства, но отличающихся деталями. Однако не менее важно и другое применение техники параллельных потоков, когда порожденные потоки принципиально отличаются друг от друга, выполняют разные задачи и имеют различные рабочие функции. Рассмотрим пример программы такого рода.

Пусть нам надо разработать программу управления некоторым измерительным оборудованием, например медицинского характера. Создадим в рамках процесса, помимо первичного потока, еще 3 порожденных, закрепив за этими четырьмя потоками следующие функции:

- первичный поток – вывод в главное окно информации о текущем времени;
- порожденный поток 1 – получение данных из измерительной установки в реальном времени и вывод их на экран в форме графика;
- порожденный поток 2 – просмотр файлов данных, относящихся к проводимым измерениям, например протоколов предыдущих сеансов;
- порожденный поток 3 – подача звуковых сигналов каждые 5 с для синхронизации работы оператора.

Разумеется, в нашем примере будет много искусственного. Так, данные из "измерительной установки" мы будем получать с помощью программного датчика случайных чисел; в потоке, организующем просмотр файлов, мы ограничимся поиском в текущем каталоге файлов с расширением .DAT и выводом перечня таких файлов в окно диалогового списка; в программе будет также отсутствовать вполне естественная операция по сохранению в файле результатов текущих измерений. На рис. 12.8 приведена иллюстрация работы описываемой ниже программы 12-5.

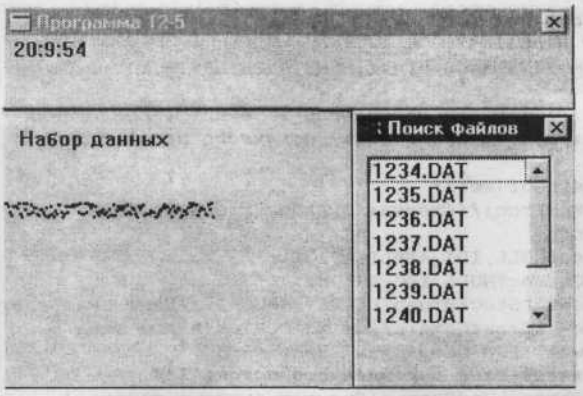


Рис. 12.8. Работа программы управления измерительной установкой

```
/*Программа 12-5. Потоки с различными рабочими функциями*/
/*Файл 12-5.h*/
#define ID_LIST 100
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
BOOL OnCreate(HWND,LPCREATESTRUCT);
void OnPaint(HWND);
void OnCommand(HWND,int,HWND,UINT);
void OnTimer(HWND,UINT);
void OnDestroy(HWND);
BOOL CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM);
void DlgOnCommand(HWND,int,HWND,UINT);
BOOL DlgOnInitDialog(HWND,HWND,LPARAM);
void OutTime(void);
DWORD WINAPI Thread1(LPVOID);
DWORD WINAPI Thread2(LPVOID);
DWORD WINAPI Thread3(LPVOID);
LRESULT CALLBACK Thread1WndProc(HWND,UINT,WPARAM,LPARAM);
void CALLBACK Thread1TmrProc(HWND,UINT,UINT,DWORD);
void CALLBACK Thread3TmrProc(HWND,UINT,UINT,DWORD);
void Thread3OnDestroy(HWND);
/*Файл 12-5.rc*/
#include "12-5.h"
Files DIALOG 120, 40, 69, 80
CAPTION "Поиск файлов" {
  CONTROL "",ID_LIST,"LISTBOX",WS_VSCROLL|WS_BORDER,3,4,61,60
  }
/*Файл 12-5.cpp*/
#include <windows.h>
#include <windowsx.h>
#include "12-5.h"
/*Глобальные переменные*/
char szClassName[]="MainWindow";//Имя класса главного окна
char szTitle[]="Программа 12-5";
HINSTANCE hInstance;//Дескриптор экземпляра приложения
char szCurrentTime[40];//Для формирования выводимой строки
DWORD dwIDThread;//Идентификатор создаваемого потока
HWND hwndMain;//Дескриптор главного окна
DWORD dwId;//Для идентификаторов потоков
int x;//Координата точки в окне потока 1
```

```
/*Функции первичного потока*/
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  MSG Msg;
  hInstance=hInst;//Сохраним дескриптор экземпляра приложения
/*Зарегистрируем класс главного окна первичного потока*/
  WNDCLASS wc;
  ZeroMemory(&wc,sizeof(wc));
  wc.lpfnWndProc=WndProc;//Оконная функция главного окна
  wc.hInstance=hInst;
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hbrBackground=GetStockBrush(LTGRAY_BRUSH);//Цвет окна светло-серый
  wc.lpszClassName=szClassName;//Имя класса главного окна
  RegisterClass(&wc);//Регистрируем класс окна
/*Зарегистрируем класс окна порожденного потока 1*/
  wc.lpfnWndProc=Thread1WndProc;//Оконная функция потока 1
  wc.hbrBackground=GetStockBrush(LTGRAY_BRUSH);//Цвет окна светло-серый
  wc.lpszClassName="Thread1";//Имя класса окна потока 1
  RegisterClass(&wc);//Регистрируем класс окон
/*Создадим и покажем главное окно*/
  hwndMain=CreateWindowEx(WS_EX_TOPMOST,szClassName,szTitle,
       WS_BORDER|WS_SYSMENU,//Стиль окна
       10,10,375,70,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwndMain,SW_NORMAL);
  while(GetMessage(&Msg,NULL,0,0))//Цикл обработки сообщений
    DispatchMessage(&Msg);//первичного потока
  return 0;
  }
/*Оконная процедура главного окна (первичного потока)*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg)  {
    HANDLE_MSG(hwnd,WM_CREATE,OnCreate);
    HANDLE_MSG(hwnd,WM_TIMER,OnTimer);
    HANDLE_MSG(hwnd,WM_PAINT,OnPaint);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
/*Функция обработки сообщения о создании главного окна*/
BOOL OnCreate(HWND hwnd,LPCREATESTRUCT){
/*Выведем время и запустим таймер даты-времени*/
  OutTime();//Первый немедленный вывод текущего времени
  SetTimer(hwnd,0,1000,NULL);//Установим таймер с периодом 1с
/*Создадим все три потока*/
  CreateThread(NULL,0,Thread1,NULL,0,&dwId);
  CreateThread(NULL,0,Thread2,NULL,0,&dwId);
  CreateThread(NULL,0,Thread3,NULL,0,&dwId);
  return TRUE;
  }
/*Функция обработки сообщения WM_PAINT для главного окна*/
void OnPaint(HWND hwnd){
  PAINTSTRUCT ps;
  HDC hdc=BeginPaint(hwnd,&ps);
  SetBkMode(hdc,TRANSPARENT);//Установим прозрачный фон символов
  TextOut(hdc,5,0,szCurrentTime,strlen(szCurrentTime));//Вывод времени
  EndPaint(hwnd,&ps);
  }
/*Функция обработки сообщений WM_TIMER для главного окна*/
void OnTimer(HWND,UINT){
```

```
  OutTime();//Периодический вывод текущего времени
  }
/*Функция обработки сообщения WM_DESTROY для главного окна*/
void OnDestroy(HWND){
  PostQuitMessage(0);
  }
/*Функция формирования строки с текущим временем для первичного потока*/
void OutTime(){
  char szT[20];
  SYSTEMTIME SystemTime;
  GetLocalTime(&SystemTime);//Получим текущее местное время
  wsprintf(szT,"%d",SystemTime.wHour);//Часы в символы
  strcpy(szCurrentTime,szT);//Запишем их в szCurrentTime
  strcat(szCurrentTime,":");//Разделяющее двоеточие
  wsprintf(szT,"%d",SystemTime.wMinute);//Минуты в символы
  strcat(szCurrentTime,szT);//Добавим их
  strcat(szCurrentTime,":");//Разделяющее двоеточие
  wsprintf(szT,"%d",SystemTime.wSecond);//Секунды в символы
  strcat(szCurrentTime,szT);//Добавим их
  InvalidateRect(hwndMain,NULL,TRUE);//Перерисовка окна
  }
/*Функции потока 1*/
/*Рабочая функция потока 1*/
DWORD WINAPI Thread1(LPVOID){
  MSG Msg;//Структурная переменная для сообщений потока 1
  HWND hwnd=CreateWindowEx(WS_EX_TOPMOST,"Thread1",NULL,
          WS_BORDER|WS_POPUP,10,80,230,185,
          HWND_DESKTOP,NULL,hInstance,NULL);//Создадим окно потока 1
  ShowWindow(hwnd,SW_NORMAL);//Покажем окно потока 1
  SetTimer(hwnd,0,100,Thread1TmrProc);//Установим таймер с периодом 0,1с
  while(GetMessage(&Msg,NULL,0,0))//Цикл обработки сообщений
    DispatchMessage(&Msg);//потока 1
  return 0;
  }
/*Оконная функция для окна потока 1*/
LRESULT CALLBACK Thread1WndProc
     (HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  return(DefWindowProc(hwnd,msg,wParam,lParam));
  }
/*Функция обработки сообщений WM_PAINT для окна потока 1*/
void Thread1OnPaint(HWND hwnd){
EndPaint(hwnd,&ps);
  }
/*Функция таймера для потока 1*/
void CALLBACK Thread1TmrProc(HWND hwnd,UINT,UINT,DWORD){
  char szText[]="Набор данных";//Надпись в окне
  RECT rect={0,30,230,155};//Прямоугольник, покрывающий график
  int y=60+random(10);//Получение "измерительных" данных
  HDC hdc=GetDC (hwnd);//Получим контекст устройства
  SetBkMode(hdc,TRANSPARENT);//Сделаем фон шрифта прозрачным
  TextOut(hdc,10,10,szText,strlen(szText));//Вывод надписи
  Rectangle(hdc,x,y,x+2,y+2);//Вывод большой квадратной точки
  x+=2;//К следующей точке графика (через 2 пиксела)
  if(x>=220){//Защита от выхода за пределы окна
    x=0;//Вернуться в начало графика
    InvalidateRect(hwnd,&rect,TRUE);//Очистить окно перерисовкой
    }
  ReleaseDC(hwnd,hdc);
  }
```

```
/*Функции потока 2*/
/*Рабочая функция потока 2*/
DWORD WINAPI Thread2(LPVOID){
  DialogBox(hInstance,"Files",NULL,DlgProc);//Откроем окно диалога
  return 0;
  }
/*Оконная функция диалога потока 2*/
BOOL CALLBACK DlgProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_INITDIALOG,DlgOnInitDialog);
    HANDLE_MSG(hwnd,WM_COMMAND,DlgOnCommand);
    default:
      return FALSE;
    }
  }
/*Функция обработки сообщения об инициализации диалога потока 2*/
BOOL DlgOnInitDialog(HWND hwnd,HWND,LPARAM){
  WIN32_FIND_DATA fd;
  HANDLE hFile=FindFirstFile("*.dat",&fd);//Поиск 1-го файла .DAT
  if(hFile==INVALID_HANDLE_VALUE)
    return TRUE;
  else
    SendDlgItemMessage(hwnd,ID_LIST,//Занесение найденного имени
          LB_ADDSTRING,0,(LPARAM)fd.cFileName);//в список
  while(FindNextFile(hFile,&fd)){//Поиск остальных файлов .DAT
    SendDlgItemMessage(hwnd,ID_LIST, //Занесение найденных имен
          LB_ADDSTRING,0,(LPARAM)fd.cFileName);//в список
    }
  return TRUE;
  }
/*Функция обработки сообщений WM_COMMAND для диалога*/
void DlgOnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id){
    case IDCANCEL:
      EndDialog(hwnd,0);
    }
  }
/*Функции потока 3*/
/*Рабочая функция потока 3*/
DWORD WINAPI Thread3(LPVOID){
  MSG Msg;
  SetTimer(NULL,0,5000,Thread3TmrProc);//Установим таймер с периодом 5с
  while(GetMessage(&Msg,NULL,0,0))//Цикл обработки сообщений
    DispatchMessage(&Msg);//для потока 3
  return 0;
  }
/*Функция таймера потока 3*/
void CALLBACK Thread3TmrProc(HWND,UINT,UINT,DWORD){
  sndPlaySound("ding.wav",SND_ASYNC);//Проиграем звуковой сигнал
  }
```

В главной функции WinMain() регистрируются классы окон первичного потока и потока 1. Каждому назначается своя оконная функция: WndProc() – для первичного потока и Thread1WndProc() – для потока 1. Создается и делается видимым окно первичного потока, после чего поток входит в цикл обработки сообщений.

В функции OnCreate() устанавливается таймер главного окна для периодического вывода текущего времени, после чего создаются все 3 порожденных потока с рабочими функциями Thread1, Thread2 и Thread3.

Процедура вывода в главное окно текущего времени уже описывалась ранее (см. пример 7-1) и не представляет особого интереса.

Функция потока 1 напоминает главную функцию приложения – в ней создается и показывается окно потока, затем устанавливается таймер для периодического опроса "измерительной установки" и, наконец, поток входит в цикл обработки сообщений. Хотя мы явным образом не будем обрабатывать какие-либо сообщения потока 1, однако в нем будет работать таймер Windows. Как уже отмечалось в гл. 7, приложения, использующие таймеры Windows, обязаны иметь цикл обработки сообщений. Теперь мы видим, что это утверждение относится не к приложению целиком, а к тем его потокам, в которых установлены таймеры.

Для окна потока 1 указан стиль WS_BORDER | WS_POPUP (вместо обычно используемого для главных окон стиля WS_OVERLAPPEDWINDOW). При таком наборе констант стиля в окне отсутствует верхняя строка с заголовком и нет системного меню. В результате у окна нельзя изменить размер; нельзя также перемещать его по экрану или завершить владеющий им поток. Для окна, в которое выводятся результаты измерений в определенном формате, это разумная предосторожность. Для того чтобы окно с результатами измерений всегда было на переднем плане, ему (как и главному окну приложения) назначен расширенный стиль WS_EX_TOPMOST.

Наличие в потоке 1 окна заставило нас ранее зарегистрировать класс этого окна, в характеристиках которого было, в частности, указано имя его оконной функции. Поскольку, однако, для этого окна никакие сообщения обрабатывать не надо, оконная функция вырождается в вызов функции DefWindowProc() обработки всех сообщений по умолчанию.

Вся содержательная работа в окне потока 1 выполняется в функции таймера Thread1TmrProc(), которая вызывается периодически с установленным ранее периодом (конкретно – 0,1 с). Предполагается, что в этой функции происходит получение текущего результата измерений из выходного порта измерительного оборудования и вывод этого результата на экран в виде точки графика. В нашей программе обращение к реальному порту заменено получением с помощью функции C++ random() случайного числа в диапазоне 0...9 и выводом на график точки с соответствующей ординатой и смещением от верхней границы окна на 60 пикселов.

Для использования каких-либо графических функций необходимо получить контекст устройства (фактически контекст окна). Если вывод в окно осуществляется в ответ на сообщение WM_PAINT, для получения контекста следует использовать функцию BeginPaint(). Во всех остальных случаях используется другая функция – GetDC(), которая тоже возвращает дескриптор контекста, но не для области вырезки, а для всего окна. В дальнейшем этот дескриптор используется обычным образом, однако для возврата его в Windows надо вместо функции EndPaint() использовать функцию ReleaseDC().

В нашем примере в контексте устройства устанавливается прозрачный фон под символами и в верхнюю часть окна выводится поясняющий текст "Набор данных" (вспомним, что в этом окне отсутствует строка заголовка и при отсутствии поясняющего текста оно было бы слишком безликим). Далее с помощью функции Rectangle() рисуется маленький квадратик с координатами x, y, где x пробегает четные значения от 0 до 220 (ширина окна в пикселах), а y представляет очередное значение "измеренной" величины. Когда координата точки доходит до правого края окна, переменной x присваивается нулевое значение и график начинает рисоваться заново от левого края, как это часто бывает в реальных приборах для медицинских измерений.

Следует подчеркнуть, что использованный в программе способ рисования в окне противоречит идеологии Windows. Для того чтобы при всяких манипуляциях с окном изображение в нем восстанавливалось, рисовать в окне следует в ответ на сообщение WM_PAINT. В нашем примере это правило не выдержано, и наложение на наше окно окна любого другого приложения приведет к стиранию изображения. Чтобы этого не происходило, окно сделано всплывающим на передний план (дополнительный стиль WS_EX_TOPMOST), так что заслонить его ничем не удастся. С другой стороны, традиционный вывод в окно в функции OnPaint() требует при каждом вызове этой функции прорисовки *всего* изображения; в нашем случае результаты измерений пришлось бы сохранять в зарезервированном для этого массиве и выводить в окно все его содержимое. В задачах реального времени, подобных нашей, когда частичные результаты измерений, выводимые на экран, недолговечны и периодически заменяются новыми, примененный способ (хранение отображаемых данных не в программной переменной, а непосредственно в видеопамяти экрана) имеет право на существование.

После того как координата выводимой точки достигнет правого края окна и снова примет нулевое значение, окно необходимо очистить. Для этого достаточно вызвать функцию InvalidateRect() с указанием в качестве последнего параметра значения TRUE (перерисовать фон). Windows пошлет в приложение сообщение WM_PAINT, которое, будучи обработано по умолчанию, просто зарисует все окно цветом фона. Для того чтобы эта операция была меньше заметна для глаза, в качестве второго параметра функции InvalidateRect() указан не NULL (перерисовать все окно), а координаты прямоугольника rect, смещенного на 30 пикселов от верхней границы окна вниз. Таким образом, перерисовка окна не затрагивает область надписи, что уменьшает мелькание экрана.

Рабочая функция потока 2 вызовом функции DialogBox() выводит на экран диалоговое окно со списком в соответствии со сценарием файла 12-5.RC. Программные фрагменты поиска файлов с расширением .DAT и заполнения окна списка их именами взяты без изменений из программы 6-2 и вряд ли нуждаются в пояснениях.

Последний из создаваемых в приложении потоков, поток 3 с рабочей функцией Thread3(), любопытен тем, что в нем отсутствует интерфейс с пользователем в виде обычного окна или диалога, однако имеется цикл обработки сообщений. В рабочей функции устанавливается таймер с периодом 5 с, после чего поток входит в бесконечный цикл GetMessage() – DispatchMessage(), который, как уже отмечалось, необходим для работы таймера. Функция таймера Thread3TmrProc(), активизируемая каждые 5 с, вызывает проигрывание короткого музыкального файла (файл DING.WAV в примере), создавая тем самым звуковые метки времени.

Глава 13
Обмен информацией между процессами и потоками

Как правило, потоки, сосуществующие в рамках одного процесса, выполняют взаимосвязанные задачи и работа одного потока может зависеть от хода или результатов работы другого. В этом случае необходимо организовать передачу информации из потока в поток. В качестве передаваемой информации часто выступают данные, подготовленные, например, в одном потоке, но используемые в другом. Однако из потока в поток (и из процесса в процесс) можно передавать не только данные, но и приказы на выполнение заданных действий. Остановимся сначала на межпоточной передаче данных. В самой передаче данных нет никакой сложности, так как все потоки процесса работают в едином адресном пространстве и всем им доступны глобальные переменные и функции процесса. Поэтому для передачи некоторых данных из одного потока в другой достаточно эти данные объявить глобальными. Реально, однако, межпоточный обмен данными часто требует решения задач синхронизации. Действительно, если один поток, например, формирует данные для обработки их другим потоком, то другой поток должен быть извещен о моменте готовности данных и не должен пытаться принимать их ранее этого момента. Вопросы синхронизации параллельно работающих потоков, как принадлежащих одному процессу, так и выполняемых в контексте разных процессов, будут рассмотрены в следующей главе. Настоящая глава посвящена способам передачи информации между процессами. Этих способов не так уж много. К ним относятся:

- передача в дочерний процесс при его запуске параметров командной строки;
- использование механизма сообщений;
- совместное использование файлов данных;
- совместное использование страничного файла.

Передача в дочерний процесс командной строки рассматривалась в предыдущей главе (см. программу 12-2). Здесь мы остановимся на оставшихся трех способах.

Обмен сообщениями

Как видно из многочисленных примеров, приведенных в этой книге, механизм сообщений относится к числу основополагающих концепций системы Windows. Заполнение окон приложения какими-либо изображениями, отклик на выбор пунктов меню, управление диалоговыми окнами – все это осуществляется с помощью инициирования и обработки тех или иных сообщений Windows. При этом до сих пор мы сталкивались с сообщениями либо поступающими из Windows в окно нашего приложения, либо инициированных в приложении с целью влияния на элементы управления (например, для занесения новой строки в элемент управления – список). Как будет показано в этом разделе, сообщения можно посылать также из одного приложения в другое (из процесса в процесс), создавая между процессами определенную связь.

Посылка сообщения в другой процесс осуществляется вызовом функции SendMessage(), в числе параметров которого указывается дескриптор окна, которому посылается

данное сообщение, код сообщения и соответствующие конкретному сообщению параметры. Внутри разрабатываемого приложения не составляет труда сохранить дескрипторы всех созданных окон; дескрипторы окон других процессов мы, конечно, не знаем. Однако для их определения предусмотрена функция Windows FindWindow() со следующим прототипом:

```
HWND FindWindow(
  LPCTSTR  lpClassName,//Имя класса искомого окна
  LPCTSTR  lpWindowName//Заголовок искомого окна
  );
```

Таким образом, получить дескриптор окна активного приложения Windows не составляет труда – надо лишь знать имя класса этого окна или его заголовок. Если функция FindWindow() находит среди активных окон искомое, она возвращает значение дескриптора этого окна; если окно не найдено, возвращается FALSE.

Получив дескриптор окна приложения, с которым мы хотим установить связь, можно посылать в него любые сообщения, имеющие для приложения-приемника смысл (т. е. такие, которые в нем обрабатываются, хотя бы и по умолчанию). Разумеется, посылка сообщения в приложение подразумевает, что в приложении-приемнике существует очередь сообщений и цикл обработки сообщений, т. е. что оно является интерфейсным.

Разработаем программный комплекс, состоящий из двух приложений – 13-1a и 13-1b. Первое будет представлять собой процесс-отправитель, второе – процесс-приемник. Пусть оба приложения обладают обычным интерфейсом с главным окном и меню. В приложении-приемнике с помощью меню можно открыть модальный диалог с информацией о программе; в приложении-отправителе предусмотрим возможность выполнения следующих действий:

- создание дочернего процесса-приемника и определение дескриптора его главного окна;
- посылка в процесс-приемник сообщения WM_SETTEXT, изменяющее заголовок его главного окна;
- посылка в процесс-приемник сообщения WM_COMMAND, имитирующее выбор в этом приложении определенного пункта меню;
- посылка в процесс-приемник сообщения WM_DESTROY "насильственного" завершения этого приложения.

Заметим, что пока мы рассматриваем только передачу в процесс-приемник стандартных сообщений Windows; пересылке прикладных данных будет посвящен следующий пример.

Приведем сначала текст программы-приемника 13-1b. Она представляет собой вполне законченное автономное приложение, которое запускается с Рабочего стола Windows или через меню "Пуск", после чего с помощью команд меню можно управлять его работой – вызвать окно модального диалога с информацией о программе или завершить ее работу. Программа 13-1b почти полностью повторяет рассмотренную ранее программу 6-1.

```
/*Программа 13-1b. Приложение-приемник для демонстрации межпроцессной связи*/
/*Файл 13-1b.h*/
#define MI_ABOUT 100//Идентификаторы пунктов меню
#define MI_EXIT 101
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
void OnCommand(HWND,int,HWND,UINT);
void OnPaint(HWND);
void OnDestroy(HWND);
BOOL CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM);
BOOL DlgOnInitDialog(HWND,HWND,LPARAM);
void DlgOnCommand(HWND,int,HWND,UINT);
```

```
/*Файл 13-1b.rc*/
#include "13-1b.h"
/*Сценарий меню*/
Main MENU {
  POPUP "Файл"{
    MENUITEM "О программе",MI_ABOUT
    MENUITEM "Выход",MI_EXIT
    }
  }
/*Сценарий диалога*/
About DIALOG 65, 0, 93, 40
STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU | DS_MODALFRAME
CAPTION "О программе"{
  CTEXT "Демонстрация связи процессов", -1, 12, 4, 72, 25
  PUSHBUTTON "Закрыть", IDOK, 27, 24, 43, 12
  }
/*Файл 13-1b.cpp*/
#include <windows.h>
#include <windowsx.h>
#include "13-1b.h"
char szClassName[]="WinRcv";//Имя класса окна-приемника
char szTitle[]="Программа 13-1b";//Заголовок окна-приемника
HINSTANCE hInstance;//Дескриптор приложения
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  MSG Msg;
  WNDCLASS wc;
  ZeroMemory(&wc,sizeof (WNDCLASS));
  wc.lpfnWndProc=WndProc;
  wc.hInstance=hInst;
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
  wc.lpszMenuName="Main";
  wc.lpszClassName=szClassName;
  RegisterClass(&wc);
  HWND hwnd=CreateWindow(szClassName,szTitle, //Создадим окно
            WS_OVERLAPPEDWINDOW,210,10,200,100,
            HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_NORMAL);//Покажем окно
  while(GetMessage(&Msg,NULL,0,0))//Цикл обработки сообщений
    DispatchMessage(&Msg);
  return 0;
  }
/*Оконная функция главного окна*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_PAINT,OnPaint);
    HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
/*Функция обработки сообщений WM_PAINT*/
void OnPaint(HWND hwnd){
  PAINTSTRUCT ps;
  char szText[]="Процесс-приемник";
  HDC hdc=BeginPaint(hwnd,&ps);
  TextOut(hdc,5,10,szText,strlen(szText));
```

```
  EndPaint(hwnd,&ps);
  }
/*Функция обработки сообщений WM_COMMAND от пунктов меню*/
void OnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id){//Идентификатор пункта меню
    case MI_ABOUT://Открыть диалог
      DialogBox(hInstance,"About",hwnd,DlgProc);
      break;
    case MI_EXIT://Завершение процесса
      DestroyWindow(hwnd);
    }
  }
/*Функция обработки сообщения WM_DESTROY*/
void OnDestroy(HWND){
  PostQuitMessage(0);
  }
/*Оконная функция диалога*/
BOOL CALLBACK DlgProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg)  {
    HANDLE_MSG(hwnd,WM_INITDIALOG,DlgOnInitDialog);
    HANDLE_MSG(hwnd,WM_COMMAND,DlgOnCommand);
    default:
      return FALSE;
      }
   }
/*Функция обработки сообщения WM_INITDIALOG для диалогового окна*/
BOOL DlgOnInitDialog(HWND,HWND,LPARAM){
  return TRUE;
  }
/*Функция обработки сообщений WM_COMMAND для диалогового окна*/
void DlgOnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id){//Код элемента управления (кнопки)
    case IDOK://Нажата кнопка "Закрыть"
    case IDCANCEL://Дана команда закрыть диалог через его системное меню
      EndDialog(hwnd,0);//Закрытие диалогового окна
    }
  }
```

Приведенный выше текст программы ничем не замечателен и не нуждается в комментариях. Рассмотрим теперь текст программы 13-1a, которая будет служить процессом-отправителем.

```
/*Программа 13-1a. Приложение-отправитель для демонстрации связи между
  процессами*/
//Файл 13-1a.h
#define MI_ABOUT 100 //Идентификатор пункта меню процесса-приемника
#define MI_NEW 100 //Идентификаторы
#define MI_LINK1 101 //пунктов
#define MI_LINK2 102 //меню
#define MI_LINK3 103 //данного
#define MI_EXIT 104 //процесса
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
void OnCommand(HWND,int,HWND,UINT);
void OnDestroy(HWND);
BOOL OnCreate(HWND,LPCREATESTRUCT);
void OnPaint(HWND);
void OnCopyData(HWND,HWND,COPYDATASTRUCT*);
/*Файл 13-1a.rc*/
#include "13-1a.h"
Main MENU {
  POPUP "Файл"{
```

```
    MENUITEM "Запуск процесса",MI_NEW
    MENUITEM "Смена заголовка",MI_LINK1
    MENUITEM "Выбор пункта меню",MI_LINK2
    MENUITEM "Завершение процесса",MI_LINK3
    MENUITEM SEPARATOR
    MENUITEM "Выход",MI_EXIT
    }
  }
/*Файл 013-1a.cpp*/
#include <windows.h>
#include <windowsx.h>
#include "13-1a.h"
char szClassName[]="WinSnd";//Имя класса главного окна
char szTitle[]="Программа 013-1a";//Заголовок главного окна
HWND hwndRcv;//Дескриптор главного окна процесса-приемника
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  MSG Msg;
  WNDCLASS wc;
  ZeroMemory(&wc,sizeof(wc));
  wc.lpfnWndProc=WndProc;
  wc.hInstance=hInst;
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
  wc.lpszMenuName="Main";
  wc.lpszClassName=szClassName;
  RegisterClass(&wc);
  HWND hwnd=CreateWindow(szClassName,szTitle,//Создаем главное окно
      WS_OVERLAPPEDWINDOW,10,10,200,100,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_NORMAL);//Показываем окно
  while(GetMessage(&Msg,NULL,0,0))//Цикл обработки сообщений
    DispatchMessage(&Msg);
  return 0;
  }
/*Оконная функция главного окна*/
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg)  {
    HANDLE_MSG(hwnd,WM_PAINT,OnPaint);
    HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
/*Функция обработки сообщений WM_COMMAND от пунктов меню*/
void OnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id){
    case MI_NEW:{//Запустим процесс-приемник
      STARTUPINFO si;
      PROCESS_INFORMATION pi;
      ZeroMemory(&si,sizeof(si));
      si.cb=sizeof(si);
      hwndRcv=FindWindow("WinRcv",NULL);
      if(!hwndRcv){
        CreateProcess(NULL,"13-1b",NULL,NULL,FALSE,0,NULL,NULL,&si,&pi);
        while(!hwndRcv)//Дождемся запуска процесса
          hwndRcv=FindWindow("WinRcv",NULL);//Получим дескриптор окна
        }
      else
```

```
        MessageBox(NULL,"Процесс уже запущен","Info",MB_OK);
      break;
      }//Конец case MI_NEW
    case MI_LINK1://Изменим заголовок процесса-приемника
      SendMessage(hwndRcv,WM_SETTEXT,0,(LPARAM)"Новый заголовок");
      break;
    case MI_LINK2://Выберем пункт меню процесса-приемника
      SendMessage(hwndRcv,WM_COMMAND,(WPARAM)MI_ABOUT,(LPARAM)0);
      break;
    case MI_LINK3://Завершим процесс-приемник
      SendMessage(hwndRcv,WM_DESTROY,0,0);
      break;
    case MI_EXIT://Завершим процесс-отправитель
      DestroyWindow(hwnd);
    }
  }
/*Функция обработки сообщений WM_PAINT*/
void OnPaint(HWND hwnd){
  char szT[]="Процесс-отправитель";
  PAINTSTRUCT ps;
  HDC hdc=BeginPaint(hwnd,&ps);
  TextOut(hdc,5,10,szT,strlen(szT));
  EndPaint(hwnd,&ps);
 }
/*Функция обработки сообщения WM_DESTROY*/
void OnDestroy(HWND){
  PostQuitMessage(0);
  }
```

При запуске приложения 13-1a на экран выводится его главное окно со строкой текста, которая на рис. 13.1 заслонилась открытым меню.

Все содержательные действия программы сосредоточены в функции OnCommand(). При выборе пункта меню "Запуск процесса" (идентификатор MI_NEW) прежде всего осуществляется попытка найти среди активных процессов окно с именем класса "WinRcv".

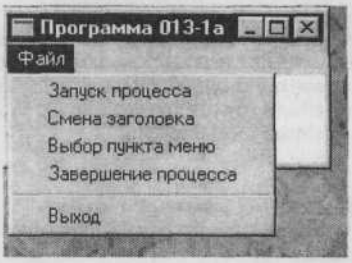


Рис. 13.1. Главное окно процесса-отправителя 13-1a с открытым меню

Если такое окно найдено (функция FindWindow() возвращает ненулевой дескриптор), на экран выводится предупреждающее сообщение и обслуживание данного пункта меню завершается; если окна нет, создается процесс 13-1b. Указанная проверка защищает программный комплекс от запуска повторных экземпляров одного и того же процесса-приемника.

После создания процесса необходимо получить дескриптор его главного окна. Здесь мы опять сталкиваемся с проблемой синхронизации множественных процессов одного программного комплекса. Дело в том, что фактическое создание процесса требует времени; если после вызова функции CreateProcess() сразу же попытаться получить дескриптор его окна, то функция FindWindow() вернет FALSE, так как к этому моменту новый процесс еще не будет создан. Поэтому в программе выполняются циклические попытки получить дескриптор окна процесса-приемника до тех пор, пока функция FindWindow() не вернет ненулевое значение. Полученное значение дескриптора сохраняется в глобальной переменной hwndRcv для дальнейшего использования. На рис. 13.2 показано изображение на Рабочем столе после запуска процесса-приемника.

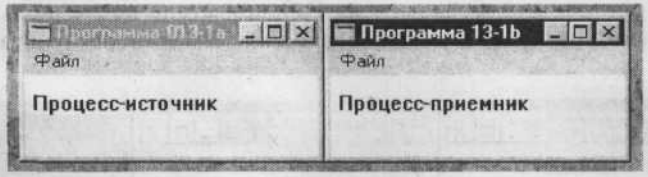


Рис. 13.2. Из родительского процесса запущен процесс-приемник

Дальнейшие действия пользователя заключаются в выборе различных пунктов меню родительского процесса. При выборе пункта "Смена заголовка" (идентификатор MI_LINK1) в процесс-приемник посылается сообщение WM_SETTEXT, которое позволяет изменить заголовок его окна (не будем утверждать, что это очень полезное действие, однако оно наглядно демонстрирует пересылку сообщений между процессами).

При выборе пункта "Завершение процесса" (идентификатор MI_LINK3) в процесс-приемник посылается сообщение WM_DESTROY, которое естественным образом приводит к завершению дочернего процесса.

Наиболее интересна команда "Выбор пункта меню" (идентификатор MI_LINK2). В этом случае в процесс-приемник посылается сообщение WM_COMMAND, которое имитирует выбор в этом процессе какого-либо пункта его меню. Для того чтобы правильно сформировать сообщение WM_COMMAND, надо посмотреть справку об этом сообщении в интерактивном справочнике:

```
WM_COMMAND
wNotifyCode = HIWORD(wParam);//Код уведомления
wID = LOWORD(wParam);//Идентификатор пункта или элемента управления
hwndCtl = (HWND) lParam;//Дескриптор элемента управления
```

и сопоставить параметры сообщения с параметрами функции SendMessage(), назначение которых следует из прототипа функции:

```
LRESULT SendMessage(
  HWND hwnd,//Дескриптор окна-приемника сообщения
  UINT uMsg,//Код сообщения (у нас WM_COMMAND)
  WPARAM wParam,//Первый параметр сообщения
  LPARAM lParam//Второй параметр сообщения
  );
```

Из приведенных справок видно, что в старшей половине параметра wParam функции SendMessage() должен содержаться код уведомления, а в младшей половине – идентификатор пункта меню (вспомним, что в 32-разрядных приложениях оба параметра, и wParam и lParam, имеют размерность 32 бита). Кода уведомления в нашем случае нет, а идентификатор активизируемого нами пункта меню "О программе" процесса-приемника равен MI_ABOUT. В файл 13-1A.H нами был предусмотрительно включен макрос #define с определением значения этого идентификатора, и мы можем воспользоваться этим символическим обозначением при вызове функции SendMessage():

```
SendMessage(hwndRcv,WM_COMMAND,(WPARAM)MI_ABOUT,(LPARAM)0);
```

Разумеется, важно здесь не символическое обозначение пункта меню MI_ABOUT, а его значение (100), которое должно быть таким же, как и в процессе-приемнике. Функция SendMessage() ожидает получить в качестве третьего и четвертого параметров переменные типа WPARAM и LPARAM соответственно. Поэтому мы для наглядности преобразуем символическую константу MI_ABOUT в тип WPARAM, а последний 0 – в LPARAM, хотя в данном случае явные преобразования необязательны.

На рис. 13.3 показано изображение на Рабочем столе после посылки в процесс-приемник сообщений об изменении заголовка окна и о выборе пункта меню "О программе".

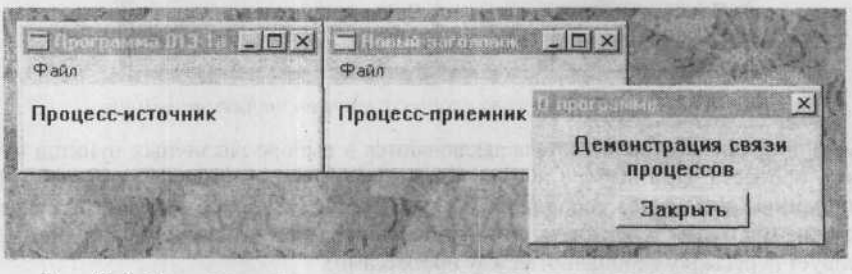


Рис. 13.3. Состояние процесса-приемника после замены заголовка его окна и активизации модального диалога из процесса-отправителя

Передача данных с помощью механизма сообщений

Рассмотрим теперь вопрос о передаче в другой процесс массива данных. Непосредственно это сделать нельзя, так как, хотя все процессы работают в одном и том же виртуальном (и линейном) адресном пространстве, их физические адреса не перекрываются и один процесс не имеет доступа к переменным другого. Однако для передачи данных можно воспользоваться специально для этого предназначенным сообщением WM_COPYDATA, в качестве косвенного параметра которого может выступать адрес любой коллекции данных.

Разработаем программный комплекс, в котором родительский процесс-отправитель (программа 13-2a) подготавливает значительный по объему массив данных и передает его запущенному им же процессу-приемнику (программе 13-2b), а тот занимается пересылкой этого массива по назначению (на удаленный компьютер, принтер или просто диск; в нашем примере массив записывается на диск). Для простоты в родительском процессе не будем связываться с меню или даже с главным окном; пусть этот процесс носит безынтерфейсный, чисто вычислительный характер. С дочерним процессом-приемником так поступить не удастся, так как для приема сообщения ему нужны и оконная функция с циклом обработки сообщений, и само окно.

```
/*Программа 13-2a. Передача данных процессу с помощью сообщения WM_COPYDATA*/
/*Файл 13-2a.cpp*/
#include <windows.h>
HWND hwndRcv;//Дескриптор окна процесса-приемника сообщения
int nData[0x100000];//Пересылаемый массив данных
/*Главная функция приложения WinMain*/
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
/*Создадим дочерний процесс и получим дескриптор его окна*/
  STARTUPINFO si;
  PROCESS_INFORMATION pi;
  ZeroMemory(&si,sizeof(si));
  si.cb=sizeof(si);
  hwndRcv=FindWindow("WinRcv",NULL);
  if(!hwndRcv){
    CreateProcess(NULL,"13-2b",NULL,NULL,FALSE,0,NULL,NULL,&si,&pi);
    while(!hwndRcv)
      hwndRcv=FindWindow("WinRcv",NULL);
    }
```

```
  else
    MessageBox(NULL,"Процесс уже запущен","Info",MB_OK);
/*Подготовим передаваемый в дочерний процесс массив*/
  for(int i=0;i<0x100000;i++)
    nData[i]=i*10;
/*Передадим данные в другой процесс*/
  COPYDATASTRUCT cds;
  cds.dwData=0;
  cds.cbData=sizeof(nData);
  cds.lpData=(PVOID)nData;
  SendMessage(hwndRcv,WM_COPYDATA,NULL,(LPARAM)&cds);
  return 0;//Завершим приложение
  }
/*Программа 13-2b. Прием данных из процесса с помощью сообщения WM_COPYDATA*/
/*Файл 13-2b.h*/
#define HANDLE_WM_COPYDATA(hwnd,wParam,lParam,fn)\ //Макрос для сообщения
  ((fn)((hwnd),(HWND)(wParam),(COPYDATASTRUCT*)(lParam))?1L:0L) //WM_COPYDATA
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
void OnDestroy(HWND);
BOOL OnCopyData(HWND,HWND,COPYDATASTRUCT*);
/*Файл 13-2b.cpp*/
#include <windows.h>
#include <windowsx.h>
#include "13-2b.h"
char szClassName[]="WinRcv";
char szTitle[]="Программа 13-2b";
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  MSG Msg;
  WNDCLASS wc;
  ZeroMemory(&wc,sizeof(wc));
  wc.lpfnWndProc=WndProc;
  wc.hInstance=hInst;
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
  wc.lpszClassName=szClassName;
  RegisterClass(&wc);
  HWND hwnd=CreateWindow(szClassName,szTitle,WS_OVERLAPPEDWINDOW,
       210,10,200,100,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_NORMAL);
  while(GetMessage(&Msg,NULL,0,0))
    DispatchMessage(&Msg);
  return 0;
  }
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg)  {
    HANDLE_MSG(hwnd,WM_COPYDATA,OnCopyData);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
/*Функция обработки сообщения WM_COPYDATA */
void OnCopyData(HWND hwnd,HWND,COPYDATASTRUCT* pcds){
  DWORD dwCnt;
  HANDLE hFile=CreateFile("1.1",GENERIC_WRITE,//Создадим файл
              0,NULL,CREATE_ALWAYS,0,NULL);
  WriteFile(hFile,pcds->lpData, pcds->cbData,&dwCnt,NULL);//Запись данных
  CloseHandle(hFile);//Закроем дескриптор файла
```

```
  DestroyWindow(hwnd);//Завершим приложение
  return TRUE;//После обработки WM_COPYDATA следует вернуть TRUE
  }
/*Функция обработки сообщения WM_DESTROY*/
void OnDestroy(HWND){
  PostQuitMessage(0);
  }
```

Как уже отмечалось, приложение-отправитель данных состоит из единственной функции WinMain(). В ней делается попытка найти окно другого активного процесса с именем класса "WinRcv" и, если такого окна нет, создается процесс-приемник 13-2b (запускается приложение 13-2B.EXE). В любом случае в переменной hwndRcv сохраняется дескриптор окна процесса-приемника.

Далее подготавливаются данные для передачи в процесс-приемник (в примере целочисленный массив nData заполняется последовательным рядом чисел, кратных 10), после чего выполняются операции по пересылке данных в составе сообщения WM_COPYDATA окну с дескриптором hwndRcv.

Использование сообщения WM_COPYDATA требует наличия в программе структурной переменной типа COPYDATASTRUCT. В файле WINUSER.H определен состав этой структуры:

```
typedef struct tagCOPYDATASTRUCT {
  DWORD dwData;//Произвольный параметр, передаваемый приложению-приемнику
  DWORD cbData;//Размер в байтах передаваемых данных
  PVOID lpData;//Адрес данных, передаваемых приложению-приемнику
  } COPYDATASTRUCT;
```

Видно, что через элемент lpData процессу-приемнику можно передать адрес любой коллекции данных (поскольку этот элемент имеет тип обобщенного адреса PVOID), через элемент cbData – объем передаваемых данных, а через элемент dwData – любой дополнительный параметр. Инициализировав эти элементы структурной переменной должным образом, мы вызовом функции SendMessage() посылаем в процесс-приемник сообщение WM_COPYDATA. В качестве параметров этого сообщения указывается дескриптор окна передающего процесса (но у нас окна в передающем процессе нет, и этот параметр имеет значение NULL) и адрес структурной переменной типа COPYDATASTRUCT.

Если функция SendMessage() используется для передачи данных в другой процесс, т. е. передает сообщения WM_SETTEXT или WM_COPYDATA, то она работает довольно сложным образом. Для того чтобы данные процесса-отправителя были доступны процессу-приемнику, функция выделяет в адресном пространстве процесса-приемника область линейных адресов, накладываемых на пересылаемые данные. Базовый адрес этой области передается в процесс-приемник в качестве элемента lpData структуры COPYDATASTRUCT, что дает возможность процессу-приемнику читать пересылаемые данные.

Далее функция SendMessage() осуществляет *синхронную* пересылку сообщения. Это означает, что управление вернется в процесс-отправитель (на предложение программы, следующее за вызовом функции SendMessage()) только после того, как это сообщение будет полностью обработано в процессе-приемнике, т. е. когда в функции OnCopyData() будет выполнен последний оператор

```
return TRUE;
```

(в соответствии с документацией Microsoft, если сообщение SendMessage() обрабатывается в прикладной программе, она должна вернуть значение TRUE). Таким образом, про-

цесс-отправитель, отправив сообщение WM_COPYDATA, *останавливается* до возврата из функции, обрабатывающей это сообщение в процессе-приемнике. Поэтому желательно, чтобы обработка сообщения не занимала слишком много времени.

Наконец, последнее. Адрес пересылаемых данных, передаваемый в процесс-приемник вместе со структурой COPYDATASTRUCT, действителен, лишь пока длится обработка сообщения WM_COPYDATA. Как только функция, обрабатывающая это сообщение, завершается, система уничтожает отображение выделенного участка адресов на пересылаемые данные. Учитывая предыдущее замечание, разумно в функции обработки сообщения WM_COPYDATA не обрабатывать данные, а только скопировать их в специально предусмотренный для этого (или динамически выделенный) буфер. Вернув управление процессу-отправителю, процесс-приемник может теперь уже без спешки заняться обработкой полученных данных, что, впрочем, потребует введения в него какого-либо механизма синхронизации.

Приемная программа 13-2b выглядит традиционно. В оконную функцию включены два макроса HANDLE_MSG для обработки сообщений WM_COPYDATA и WM_DESTROY. Однако в файле WINDOWSX.H отсутствует макрос HANDLE_WM_COPYDATA, поэтому его пришлось составить по образцу остальных макросов такого рода и включить в файл 13-2B.H. Использованный в макросе оператор "?" осуществляет возврат значения 1L (интерпретируемое далее как TRUE), если функция fn (в нашем случае OnCopyData()) вернула TRUE; если же OnCopyData() возвращает FALSE, то и в Windows поступает значение 0L, равносильное FALSE.

В функции OnCopyData() выполняется создание нового файла 1.1 и запись в него всего массива данных, полученного через структуру COPYDATASTRUCT. Своим последним действием функция OnCopyData() завершает приложение-приемник.

Обмен данными через файлы

Совместное использование файлов данных

В тех случаях, когда скорость обработки данных не имеет решающего значения, обмен данными между процессами можно выполнять через разделяемые файлы. Для того чтобы сделать файл разделяемым, достаточно при его открытии указать флаги FILE_SHARE_READ или FILE_SHARE_WRITE (или их комбинацию). Рассмотрим пример совместного использования файла двумя процессами. Проще всего в качестве двух процессов просто использовать два экземпляра одного приложения. Для того чтобы не осложнять себе жизнь процедурами синхронизации, будем считать, что процессы обращаются к общему файлу только с целью чтения из него отдельных участков. В качестве рабочего файла воспользуемся файлом STOCK.DAT, созданным с помощью программы 11-1. В этом файле размером 8000000 байт будто бы записана информация о номерах и стоимостях неких образцов, причем значение стоимости всегда в 10 раз больше номера образца, что удобно для контроля правильности чтения файла.

Организуем программу 13-3 на базе модального диалога с элементом управления – списком. Пусть добавление в список очередной выборки из файла осуществляется по нажатии кнопки, а номер выборки определяется случайным образом с помощью датчика случайных чисел.

```
/*Программа 13-3. Совместное использование файла несколькими процессами*/
/*Файл 13-3.h*/
#define ID_LIST 100
#define ID_CHOICE 101
```

```
struct GOODS{ //Структура, отражающая содержимое файла
  int Nmb;
  int Price;
  };
BOOL CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM);
void DlgOnCommand(HWND,int,HWND,UINT);
BOOL DlgOnInitDialog(HWND,HWND,LPARAM);
/*Файл 13-3.rc*/
#include "13-3.h"
Files DIALOG 41, 23, 84, 115
STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Выборки из файла" {
  LISTBOX ID_LIST, 5, 4, 74, 88, WS_VSCROLL |WS_BORDER
  PUSHBUTTON "Выборка", ID_CHOICE, 24, 98, 36, 14, BS_PUSHBUTTON
  }
/*Файл 13-3.cpp*/
#include <windows.h>
#include <windowsx.h>
#include "13-3.h"
HANDLE hFile;//Дескриптор файла
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  DialogBox(hInst,"Files",NULL,DlgProc);
  return 0;
  }
/*Оконная функция диалогового окна*/
BOOL CALLBACK DlgProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_INITDIALOG,DlgOnInitDialog);
    HANDLE_MSG(hwnd,WM_COMMAND,DlgOnCommand);
    default:
      return FALSE;
    }
  }
/*Функция обработки сообщения WM_INITDIALOG*/
BOOL DlgOnInitDialog(HWND,HWND,LPARAM){
  hFile=CreateFile("stock.dat",GENERIC_READ,//Откроем файл для чтения
              FILE_SHARE_READ,//с разделяемым доступом
              NULL,OPEN_EXISTING,0,NULL);
  randomize();//Задаем случайное начальное значение датчику случайных чисел
  return TRUE;
  }
/*Функция обработки сообщений от элементов управления диалогом*/
void DlgOnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id){
    case ID_CHOICE:{
      GOODS Stock;//Переменная для чтения из файла
      DWORD dwCount;//Число прочитанных байтов
      char szUnit[20];//Поле для формирования строк списка
      int numb=random(1000000);//Получим случайное число
      SetFilePointer(hFile,numb*sizeof(GOODS),0,FILE_BEGIN);//Сдвинем указатель
      ReadFile(hFile,&Stock,sizeof(Stock),&dwCount,NULL);//Прочитаем
      wsprintf(szUnit,"%d   %d",Stock.Nmb,Stock.Price);//
      SendDlgItemMessage(hwnd,ID_LIST,LB_ADDSTRING,0,(LPARAM)szUnit);//В список
      break;
      }
    case IDCANCEL:
      EndDialog(hwnd,0);
    }
  }
```

В функции DlgOnInitDialog() файл STOCK.DAT открывается для чтения с флагом разделяемого доступа FILE_SHARE_READ. Для того чтобы при запуске нескольких экземпляров приложения каждый из них осуществлял выборку из файла случайным образом, вызовом функции C++ randomize() датчик случайных чисел инициализируется случайным значением (получаемым как некоторая производная от текущего времени).

В ответ на каждое нажатие кнопки "Выборка" случайное число, полученное от генератора случайных чисел, умножается на размер структуры GOODS и используется в качестве смещения для установки файлового указателя. Из файла читается участок, соответствующий по размеру структуре GOODS (фактически два целых числа). Далее эти числа преобразуются в символьную форму и полученная строка добавляется в список.

На рис. 13.4 приведены результаты сеанса работы с двумя экземплярами этого приложения. При желании можно было запустить и большее количество экземпляров.

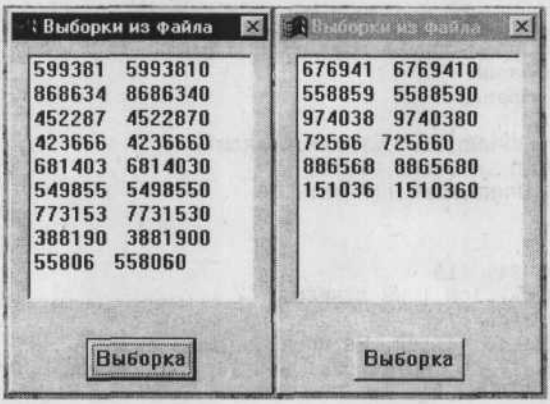


Рис. 13.4. Два экземпляра приложения работают с одним разделяемым файлом

Легко убедиться в том, что совместная работа двух процессов с одним файлом стала возможно только потому, что файл открыт в режиме разделения. Достаточно заменить при открытии файла флаг FILE_SHARE_READ на 0 – и последующие экземпляры приложения обращаться к файлу не смогут.

Обмен данными через проекцию файла в памяти

Выше (см. гл. 11) мы познакомились с методикой проецирования файлов данных в память. Использование проекций файлов позволяет упростить программу и повысить эффективность ее работы за счет перекладывания на Windows задач загрузки в память требуемых участков файлов и, при необходимости, модификации их на диске. Сущность методики заключается в том, что после создания в физической памяти специального объекта "проекция файла" выполняется отображение этой проекции на адресное пространство процесса, что и обеспечивает обращение к файлу по адресам памяти, как если бы файл был обычной программной переменной.

Использование проекций файлов в памяти для передачи данных между процессами основано на том, что объект "проекция файла" после своего создания оказывается доступным для всех процессов, выполняемых в системе. Исходный процесс создает проекцию файла с помощью функции CreateFileMapping() (см. программу 11-3). Процедура доступа к созданной проекции со стороны других процессов включает в себя два этапа:

- открытие имеющейся проекции с помощью функции OpenFileMapping();

- отображение открытой проекции на адресное пространство процесса с помощью функции MapViewOfFile().

Модифицируем предыдущий пример, оставив его содержательную часть – случайные выборки записей из файла STOCK.DAT и помещение их в список, но вместо работы с файлом на диске используем проекцию файла в памяти. Для демонстрации передачи данных из процесса в процесс организуем программный комплекс из двух весьма близких по алгоритму приложений. Программа 13-4a открывает файл с данными и создает его проекцию в памяти, после чего запускает дочерний процесс (программу 13-4b). Дочерний процесс получает доступ к той же проекции файла. После этого оба процесса параллельно и независимо работают с одним и тем же файлом.

```
/*Программа 13-4a. Передача проекции файла в другой процесс*/
/*Файл 13-4a.h*/
#define ID_LIST 100
#define ID_CHOICE 101
struct GOODS{//Структура, описывающая записи файла
  int Nmb;//Номер образца
  int Price;//Цена образца
  };
BOOL CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM);
void DlgOnCommand(HWND,int,HWND,UINT);
BOOL DlgOnInitDialog(HWND,HWND,LPARAM);
/*Файл 13-4a.rc*/
#include "13-4a.h"
Files DIALOG 41, 23, 84, 115
STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Программа 13-4a" {
  LISTBOX ID_LIST, 5, 4, 74, 88, WS_VSCROLL |WS_BORDER
  PUSHBUTTON "Выборка", ID_CHOICE, 24, 98, 36, 14, BS_PUSHBUTTON
  }
/*Файл 13-4a.cpp*/
#include <windows.h>
#include <windowsx.h>
#include "13-4a.h"
HANDLE hFile;//Дескриптор файла
GOODS *ptr;//Указатель на переменную типа GOODS
HANDLE hMem;//Дескриптор проекции файла в памяти
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  DialogBox(hInst,"Files",NULL,DlgProc);//Откроем модальный диалог
  return 0;
  }
/*Оконная функция модального диалога*/
BOOL CALLBACK DlgProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_INITDIALOG,DlgOnInitDialog);
    HANDLE_MSG(hwnd,WM_COMMAND,DlgOnCommand);
    default:
      return FALSE;
    }
  }
/*Функция обработки сообщения WM_INITDIALOG*/
BOOL DlgOnInitDialog(HWND,HWND,LPARAM){
  STARTUPINFO si;
  PROCESS_INFORMATION pi;
  ZeroMemory(&si,sizeof(si));
```

```
  hFile=CreateFile("stock.dat",GENERIC_READ,0,//Откроем имеющийся файл
              NULL,OPEN_EXISTING,0,NULL);
/*Создание в памяти именованного объекта "проекция файла"*/
  hMem=CreateFileMapping(hFile,NULL,PAGE_READONLY,0,0,"MyFile");
/*Отображение проекции файла на адресное пространства процесса*/
  ptr=(GOODS*)MapViewOfFile(hMem,FILE_MAP_READ,0,0,0);
/*Создадим  дочерний процесс - приложение 13-4b*/
  CreateProcess(NULL,"13-4b",NULL,NULL,FALSE,0,NULL,NULL,&si,&pi);
  return TRUE;
  }
/*Функция обработки сообщений WM_COMMAND от диалога*/
void DlgOnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id){
    case ID_CHOICE:{//Нажата кнопка "Выборка"
      char szUnit[20];//Поле для формирования строк списка
      int numb=random(1000000);//Получим случайное число
/*Читаем выборочно из проекции файла и заполняем список*/
      wsprintf(szUnit,"%d   %d",ptr[numb].Nmb,ptr[numb].Price);
      SendDlgItemMessage(hwnd,ID_LIST,LB_ADDSTRING,0,(LPARAM)szUnit);
      break;
      }
    case IDCANCEL://Закрываем диалог через системное меню
      UnmapViewOfFile(ptr);//Освободим адресное пространство процесса
      CloseHandle(hMem);//Закроем дескриптор проекции файла
      CloseHandle(hFile);//Закроем дескриптор файла
      EndDialog(hwnd,0);//Закроем диалог
    }
  }
/*Приложение 13-4b. Получение проекции файла из другого процесса*/
/*Файл 13-4b.h*/
...//Полностью повторяет файл 13-4a.h
/*Файл 13-4b.rc*/
#include "13-4b.h"
Files DIALOG 128, 23, 84, 115 //Другие координаты для удобства
STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Программа 13-4b" { //Другой заголовок для ясности
  LISTBOX ID_LIST, 5, 4, 74, 88, WS_VSCROLL |WS_BORDER
  PUSHBUTTON "Выборка", ID_CHOICE, 24, 98, 36, 14, BS_PUSHBUTTON
  }
/*Файл 13-4b.cpp*/
#include <windows.h>
#include <windowsx.h>
#include "13-4b.h"
HANDLE hFile;//Дескриптор файла
HANDLE hMem;//Дескриптор проекции файла в памяти
GOODS *ptr;//Указатель на переменную типа GOODS
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  DialogBox(hInst,"Files",NULL,DlgProc);//Откроем модальный диалог
  return 0;
  }
/*Оконная функция модального диалога*/
BOOL CALLBACK DlgProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_INITDIALOG,DlgOnInitDialog);
    HANDLE_MSG(hwnd,WM_COMMAND,DlgOnCommand);
    default:
      return FALSE;
    }
  }
```

```
/*Функция обработки сообщения WM_INITDIALOG*/
BOOL DlgOnInitDialog(HWND,HWND,LPARAM){
  randomize();//Инициализируем датчик случайных чисел случайным числом
/*Отображение имеющейся уже проекции файла
  на адресное пространство процесса*/
  hMem=OpenFileMapping(FILE_MAP_READ,FALSE,"MyFile");
  ptr=(GOODS*)MapViewOfFile(hMem,FILE_MAP_READ,0,0,0);
  return TRUE;
  }
/*Функция обработки сообщений WM_COMMAND от диалога*/
void DlgOnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id){
    case ID_CHOICE:{//Нажата кнопка "Выборка"
      char szUnit[20];//Поле для формирования строк списка
      int numb=random(1000000);//Получим случайное число
/*Читаем выборочно из проекции файла и заполняем список*/
      wsprintf(szUnit,"%d   %d",ptr[numb].Nmb,ptr[numb].Price);
      SendDlgItemMessage(hwnd,ID_LIST,LB_ADDSTRING,0,(LPARAM)szUnit);
      break;
      }
    case IDCANCEL://Закрываем диалог через системное меню
      UnmapViewOfFile(ptr);//Освободим адресное пространство процесса
      CloseHandle(hMem);//Закроем дескриптор проекции файла
      EndDialog(hwnd,0);//Закроем диалог
    }
  }
```

Оба приложения рассматриваемого программного комплекса построены на основе модального диалога. В программе 14-3a в процессе инициализации диалогового окна выполняются подготовительные действия для работы с проекцией файловых данных: открытие файла STOCK.DAT, создание в физической памяти объекта "проекция файла" и отображение ее на виртуальное адресное пространство процесса с возвратом указателя ptr на выделенный регион памяти. После этого вызовом функции CreateProcess() создается дочерний процесс – запускается приложение 14-3b.

Для того чтобы объект "проекция файла" мог использоваться другим процессом, ему назначается имя ("MyFile" в данном примере), которое должно быть известно процессу-приемнику. Имя объекта является такой же принципиальной характеристикой объекта, как и его дескриптор (значение типа HANDLE).

Дескриптор объекта возвращается функцией создания данного объекта и используется в дальнейшем в *том же* процессе для взаимодействия с объектом. Так, полученный в примере дескриптор файла hFile передается в качестве параметра в функцию CreateFileMapping(), а полученный из этой функции дескриптор проекции файла hMem используется далее при вызове функции MapViewOfFile() отображения проекции на адресное пространство процесса.

В отличие от дескриптора, *имя объекта* используется главным образом для передачи информации об объекте *другому* процессу, который, работая в собственном адресном пространстве, не может знать значений переменных первого процесса. Таким образом, имя объекта, в отличие от его дескриптора, является *общесистемной* характеристикой объекта, видимой не только в процессе-создателе объекта, но и во всех остальных.

Функция MapViewOfFile() возвращает указатель на созданный регион с отображением проекции файла. Поскольку данные, составляющие файл, могут иметь самую разную организацию, функция возвращает указатель обобщенного типа LPVOID, который в каждом конкретном случае следует явно преобразовывать в требуемый тип. Наш пример характерен в том отношении, что содержимое файла составляют записи структурного

типа GOODS. Поэтому возвращаемое функцией MapViewOfFile() значение следует преобразовать в указатель на эту структуру:

```
ptr=(GOODS*)MapViewOfFile(hMem,FILE_MAP_READ,0,0,0);
```

(указатель ptr уже был объявлен ранее как GOODS*, т. е. как указатель на структуру GOODS).

Дальнейшие действия в программе 13-4a инициируются нажатием кнопки "Выборка". В этом случае оконная функция диалогового окна передает управление на блок case ID_CHOICE функции DlgOnCommand(). Здесь вызовом функции C++ random() генерируется случайное число, используемое далее как индекс записи в отображенном на память массиве данных. Запись, соответствующая этому индексу, преобразуется в символьную форму и заносится в элемент управления – список, имеющийся в диалоговом окне.

При закрытии диалога (через системное меню или щелчком по кнопке) происходит переход на блок case IDCANCEL. Здесь вызовом соответствующих функций освобождается регион адресного пространства процесса, на который была отображена проекция файла, далее закрывается дескриптор самой проекции и, наконец, закрывается дескриптор дискового файла. Все эти действия в данном примере не обязательны, так как при завершении приложения система освобождает все выделенные ему ресурсы.

Необходимо иметь в виду, что, если некоторый объект используется несколькими процессами (как в нашем случае объект "проекция файла"), он не будет уничтожен, пока не закроются все его дескрипторы. Поэтому, несмотря на закрытие дескриптора проекции файла при завершении приложения 13-4a, второе приложение, 13-4b, будет продолжать правильно работать до *своего* завершения.

Вторая составляющая нашего программного комплекса, приложение 13-4b, отличается от первой лишь парой деталей. Для того чтобы в этом приложении генерировалась *другая* последовательность случайных чисел, в функции инициализации диалога выполняется вызов функции C++ randomize(), назначающей первому генерируемому числу случайное значение (зависящее от текущего времени). Другое отличие носит более принципиальный характер. Для получения доступа к созданному уже объекту "проекция файла" он открывается повторно вызовом функции OpenFileMapping() с указанием (что очень важно) имени этого объекта. В дальнейших примерах этой книги мы не раз еще столкнемся с использованием имени объекта для межпроцессной передачи информации разного рода.

На рис. 13.5 приведен результат одного из сеансов работы программного комплекса 13-4.

Рис. 13.5. Работа программного комплекса 13-4

Обмен данными через страничный файл

Как уже говорилось, страничный файл входит в общую физическую память вычислительной системы, заметно увеличивая ее объем. В гл. 11 был приведен пример неявного использования страничного файла при работе с массивами данных, превышающими по своим размерам наличную оперативную память. Здесь мы поговорим о другой возможности – явном использовании страничного файла для обмена данными между процессами. Идея этого метода заключается в том, что в физической памяти создается проекция страничного файла, которая затем отображается на виртуальное адресное пространство процесса. Поскольку, как было показано в предыдущем разделе, *именованную* проекцию файла могут использовать другие процессы, предварительно открыв ее, создается удобная возможность иметь участок физической памяти, отображенный на виртуальные пространства нескольких процессов (рис. 13.6). Кстати, согласно документации Microsoft это единственный способ передать физическую память из процесса в процесс.

Рис. 13.6. Отображение части страничного файла на адресные пространства двух процессов

Приведенный рисунок иллюстрирует некую казуистичность используемой терминологии. Ведь физическая память и так состоит из оперативной памяти и страничного файла. Получается, что страничный файл отображается на самого себя. В действительности создание объекта "проекция файла" для страничного файла обозначает лишь то, что физическую память для этого объекта следует выделить не из поименованного файла, а из страничного.

Рассмотрим формальный пример обмена данными через страничный файл. Пусть первый процесс (приложение 13-5a), создав в памяти проекцию страничного файла и отобразив ее на свое адресное пространство, записывает в эту память некоторые данные. Второй процесс (приложение 13-5b), отобразив ту же проекцию страничного файла на

свое адресное пространство, читает эти данные. Разумеется, в задачах такого рода необходима синхронизация работы процессов, чтобы второй процесс не начал читать из памяти раньше, чем первый процесс запишет туда что-то полезное. Вопросы синхронизации будут подробно рассмотрены в следующей главе; здесь ради простоты мы обойдемся без синхронизации процессов.

```
/*Программа 13-5a. Обмен данными через страничный файл*/
/*Файл 13-5.cpp*/
#include <windows.h>
const int nSize=0x4000000;//Размер массива данных в байтах = 64 Мбайт
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
/*Создание в памяти объекта "проекция файла" для 16 М целых чисел*/
  char szMapName[]="13-5=Mapping";//Произвольное имя проекции
  HANDLE hMem=CreateFileMapping((HANDLE)-1,NULL,PAGE_READWRITE,
              0,nSize,szMapName);
/*Отображение выделенной физической памяти на виртуальные адреса приложения*/
  int *ptr=(PINT)MapViewOfFile(hMem,FILE_MAP_ALL_ACCESS,0,0,nSize);
  for(int i=0;i<nSize/4;i++)//Создадим тестовые данные
    ptr[i]=i;
/*Запустим дочерний процесс (13-5b.exe)*/
  STARTUPINFO si;
  PROCESS_INFORMATION pi;
  ZeroMemory(&si,sizeof(si));
  si.cb=sizeof(si);
  CreateProcess(NULL,"13-5b",NULL,NULL,FALSE,0,NULL,NULL,&si,&pi);
/*Выведем окно сообщения и тем самым приостановим программу*/
  char szTxt[80];
  wsprintf(szTxt,
      "Элементы из начала,\nсередины и конца:\n0x%X\n0x%X\n0x%X",
      ptr[0],ptr[nSize/8],ptr[nSize/4-1]);
  MessageBox(NULL,szTxt,"13-5a. Запись",MB_OK);
  return 0;
  }
/*Программа 13-5b. Чтение данных из страничного файла*/
#include <windows.h>
const int nSize=0x4000000;//Размер массива данных в байтах = 64 Мбайт
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
  char lpszName[]="13-5=Mapping";//Произвольное имя проекции
/*Откроем уже созданную проекцию и отобразим ее на адреса приложения*/
  HANDLE hMem=OpenFileMapping(PAGE_READWRITE,FALSE,lpszName);
  int *ptr=(PINT)MapViewOfFile(hMem,FILE_MAP_ALL_ACCESS,0,0,nSize);
/*Выведем окно сообщения для контроля данных*/
  char szTxt[80];
  wsprintf(szTxt,"Последние элементы:\n0x%X\n0x%X\n0x%X",
      ptr[nSize/4-3],ptr[nSize/4-2],ptr[nSize/4-1]);
  MessageBox(NULL,szTxt,"13-5b. Чтение",MB_OK);
  return 0;
  }
```

При создании проекции файла в предыдущем примере мы в качестве первого параметра функции CreateFileMapping() указывали дескриптор открытого дискового файла. Для того чтобы создать проекцию страничного файла, первый параметр этой функции должен быть равен –1 (и иметь к тому же тип HANDLE). Предпоследний параметр этой функции задает размер проекции, а последний – произвольное имя объекта, которое, очевидно, будет использовано и во второй программе комплекса. Для убедительности при-

мера размер создаваемого массива данных и, соответственно, проекции файла для его хранения задан довольно большим – 16 М целых чисел, т. е. 64 Мбайт.

С помощью функции MapViewOfFile() в виртуальной памяти процесса выделяется регион того же размера (последний параметр этой функции), и проекция страничного файла отображается на этот регион. Как мы уже видели в предыдущем примере, функция MapViewOfFile() возвращает указатель на данное обобщенного типа LPVOID, который мы преобразуем в тип "указатель на целое" (PINT).

Выделенная физическая память заполняется тестовыми данными (натуральный ряд чисел от 0 до 0xFFFFFF), после чего создается дочерний процесс 13-5b. Для контроля созданного тестового массива, а также для предотвращения преждевременного завершения программы на экран выводится окно сообщения с тремя элементами массива (первым, средним и последним).

В программе 13-5b открывается проекция файла с известным именем, выполняется ее отображение на виртуальное пространство процесса и читаются 3 последних элемента массива с выводом результата в окно сообщения.

На рис. 13.7 показан результат выполнения программного комплекса.

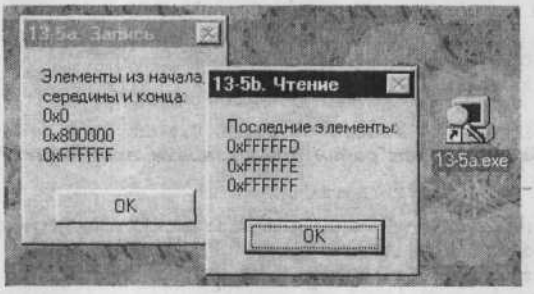


Рис. 13.7. Использование страничного файла для межпроцессной передачи данных

Глава 14
Синхронизация процессов и потоков

Общие характеристики объектов Windows

Выше уже отмечалось, что организация программного комплекса, состоящего из нескольких процессов или потоков, неминуемо приводит к необходимости синхронизации выполняемых фрагментов программы. Синхронизация может иметь различные аспекты: сообщение потоку о завершении другого потока или его фрагмента, запрет потоку обращаться к некоторым данным, пока с ними работает другой поток, наложение ограничений на количество потоков, использующих одни и те же данные, и др.

Синхронизация процессов и входящих в них потоков осуществляется с помощью различных объектов Windows, к которым относятся как сами процессы и потоки, так и программные средства, созданные специально для целей синхронизации: критические секции, мьютексы, семафоры, события и некоторые другие. В настоящей главе будет рассмотрена методика использования всех этих средств для решения различных задач синхронизации.

Как уже вскользь упоминалось ранее, для объектов, которые могут служить целям синхронизации, в системе Windows предусматриваются два состояния – свободное, или сигнальное (signaled), и занятое, или несигнальное (nonsignaled). Занятое состояние объекта используется для запрета тех или иных программных действий; свободное состояние, наоборот, снимает запрет и разрешает эти действия.

Имеются различные способы как изменения состояния синхронизирующего объекта, так и анализа его состояния. Для некоторых объектов имеются функции, позволяющие переводить объект в требуемое (свободное или занятое) состояние. Например, для объекта "событие" (event) функция SetEvent() устанавливает объект в свободное (сигнальное) состояние, а функция ResetEvent(), наоборот, сбрасывает его в занятое состояние. В соответствии с именами этих функций свободное состояние объекта называют еще установленным (set), а занятое – сброшенным (reset).

Анализ состояния объекта осуществляется с помощью двух основных функций синхронизаций – WaitForSingleObject() и WaitForMultipleObjects(). Суть их одинаковая – они останавливают выполнение потока, или "усыпляют" его, если анализируемый объект занят, и разрешают потоку дальнейшее выполнение ("будят" поток), как только объект освобождается. При этом первая из двух упомянутых выше функций позволяет анализировать состояние только одного объекта Windows, а вторая функция "чувствительна" к состоянию группы объектов.

Важно понимать, что усыпление потока является высокоэффективной системной операцией, которая почти не требует процессорного времени. Поэтому в то время как один или несколько потоков спят в ожидании некоторого события, остальные действующие в системе потоки забирают себе практически все процессорное время. Как только ожидаемое событие произойдет, спящий поток мгновенно пробуждается и продолжает свое выполнение.

Функция WaitForSingleObject() имеет следующий прототип:

```
DWORD WaitForSingleObject(
  HANDLE hObject,//Дескриптор синхронизирующего объекта
  DWORD dwTimeout//Лимит времени ожидания в миллисекундах
  );
```

Если нам нужно, чтобы функция ожидала событие неограниченное время, в качестве второго параметра следует задать константу INFINITE.

Функция WaitForMultipleObjects() выглядит сложнее:

```
DWORD WaitForMultipleObjects(
  DWORD  cObjects,//Число синхронизирующих объектов
  HANDLE *lphObjects,//Адрес массива дескрипторов синхронизирующих объектов
  BOOL  fWaitAll,//Флаг ожидания
  DWORD  dwTimeout //Лимит времени ожидания в миллисекундах
  );
```

Любопытно отметить, что массив дескрипторов синхронизирующих объектов может состоять из *разнородных* дескрипторов. Таким образом, функция WaitForMultipleObjects() может, например, усыпить процесс до того момента, как завершится определенный поток некоторого процесса и, кроме того, какой-то другой поток установит в сигнальное состояние обусловленный объект "событие".

Флаг ожидания функции WaitForMultipleObjects() может принимать два значения: TRUE и FALSE. В первом случае функция будет ожидать установки в свободное состояние *всех* объектов, указанных вторым параметром; во втором случае спящий поток пробудится, если *хотя бы один* из этих объектов окажется свободным.

Последний параметр функции WaitForMultipleObjects() имеет тот же смысл, что и для функции WaitForSingleObject().

Мы видим, что функции ожидания освобождения объектов требуют указания дескрипторов этих объектов. Вообще объекты Windows отличаются тремя характеристиками: дескриптором, именем и состоянием. Как мы уже видели, дескриптор объекта возвращается функциями, которые этот объект либо создают, либо открывают; обычно дескриптор используется для обращения к объекту внутри процесса (хотя, возможно, и из другого потока). Имя объекта, если оно необходимо, также задается объекту на этапе его создания; оно является общесистемной характеристикой и служит для обращения к объекту из другого процесса. Что касается состояния, то для объектов, которые будут рассматриваться в этой книге, характерны следующие состояния:

- Критическая секция – состояние в явной форме отсутствует. Можно с некоторой натяжкой считать, что состояние критической секции сбрасывается при ее захвате функцией EnterCriticalSection() и устанавливается при выходе из секции с помощью функции LeaveCriticalSection().
- Процесс – состояние сбрасывается при создании процесса и устанавливается при его завершении.
- Поток – состояние сбрасывается при создании потока и устанавливается при его завершении.
- Событие – состояние события задается при его создании функцией CreateEvent(). Кроме того, событие может быть установлено в процессе выполнения потока явным образом функцией SetEvent() или сброшено функцией ResetEvent(). В некоторых случаях, когда события используются в асинхронных операциях, состояние события задается системой.
- Мьютекс – это объект, схожий по смыслу с критической секцией. Мьютекс переходит в свободное состояние, когда он не занят (не захвачен) ни одним потоком. Поток, создающий мьютекс с помощью функции CreateMutex(), может потребовать его немедленного захвата, и тогда мьютекс окажется в занятом состоянии.

В противном случае для захвата мьютекса поток должен использовать одну из функций ожидания. Эти функции ставят системе запрос на захват мьютекса и в случае предоставления права на захват переводят мьютекс в занятое состояние. Для освобождения мьютекса и перевода его в свободное состояние используется функция ReleaseMutex().

- Семафор – синхронизирующий объект, используемый для учета ресурсов. Он содержит счетчик, определяющий число доступных ресурсов. Максимальное значение счетчика задается при создании семафора функцией CreateSemafor(). Семафор находится в свободном состоянии, если значение его счетчика больше нуля, и в занятом, если значение счетчика равно нулю (все ресурсы выбраны). Обычно при создании семафор оказывается установленным, хотя можно создать и занятый первоначально семафор. Выполнение функции ожидания, т. е. освобождение потока, анализирующего состояние семафора, уменьшает значение счетчика семафора на единицу, фиксируя захват потоком единицы доступного ресурса. Увеличение значения счетчика семафора на заданную величину осуществляется вызовом функции ReleaseSemafor().

Еще раз отметим, что объект, находящийся в свободном (установленном, сигнальном) состоянии, разрешает выполнение потока; объект, находящийся в занятом (сброшенном, несигнальном) состоянии, запрещает выполнение.

Критические секции и защита данных

Критической секцией называется произвольный фрагмент программы, который должен обладать монопольным доступом к некоторым данным любого содержания и объема. Особенностью критических секций является то, что их можно использовать только для синхронизации потоков, входящих в единый процесс. Если требуется обеспечить поочередный доступ к общим данным потоков, входящих в разные процессы, то в этом случае надо вместо критических секций использовать мьютексы, о которых речь будет идти ниже.

Для использования в программе критических секций следует прежде всего объявить среди глобальных данных структурную переменную типа CRITICAL_SECTION. Эта переменная внешне никак не связана ни с защищаемыми данными, ни с тем фрагментом программы, который к этим данным обращается; она, можно сказать, носит организационный характер: с ее помощью Windows определяет, можно ли в данный момент предоставить доступ к общим данным тому или иному потоку.

В первичном потоке программы, еще до создания дочерних потоков, следует выполнить инициализацию объекта Windows "критическая секция". Эта инициализация выполняется вызовом функции InitializeCriticalSection() с указанием в качестве параметра адреса структурной переменной типа CRITICAL_SECTION.

Фрагменты потоков, в которых осуществляется обращение к общим данным и которые, собственно, и называются критическими секциями, защищаются функциональными скобками

```
EnterCriticalSection()
...
LeaveCriticalSection()
```

с указанием в качестве параметра адреса той же структурной переменной типа CRITICAL_SECTION.

Какие изменения в ход выполнения потоков вносит такая организация? Обычно каждому потоку система по очереди выделяет кванты времени, в течение которых потоки и выполняются. "Отнять" время процессора у потока система может в любой момент, на любой его команде. При наличии в потоке критической секции алгоритм квантования времени усложняется. Если первый поток вошел в свою критическую секцию, система по-прежнему может в какой-то момент времени приостановить его выполнение и передать время процессора другому потоку, в том числе и второму потоку, связанному с той же переменной типа CRITICAL_SECTION. Однако как только второй поток подойдет к своей критической секции и вызовет функцию EnterCriticalSection(), система прервет его выполнение и отдаст время процессора либо снова первому потоку, либо другим потокам, действующим в системе. Пока первый поток не выйдет из *своей* критической секции (вызовом функции LeaveCriticalSection()), второй поток войти в *свою* критическую секцию не может. В результате, пока первый поток не закончит работу с общими данными, они будут недоступны второму потоку.

Разумеется, в процессе может существовать любое количество потоков, работающих с общими данными. В любом случае обращаться к общим данным они смогут только поочередно.

Структурная схема программы, в которой монопольный доступ к общим данным организуется с помощью критических секций, показана на рис. 14.1.

Первичный поток

```
CRITICAL_SECTION CritSect;      //Объявление "организационной" переменной

INT nArray[10000];              // Произвольные общие данные,
char szTitle[]="График занятости"; // подлежащие защите
PINT ptr;

InitializeCriticalSection(&CritSec); //Инициализация объекта "критическая секция"
CreateThread(..., Thread1, ...);
CreateThread(..., Thread2, ...);     //Запуск двух (или большего числа) потоков
.
. //Продолжение первичного потока
.
```

Поток 1

```
DWORD WINAPI Thread1(...){
.
. // Строки рабочей функции
. // потока 1

EnterCriticalSection(&CritSect);
. // Фрагмент рабочей функции
. // потока 1, в котором выполняется
. // обращение к общим данным
LeaveCriticalSection(&CritSect)

. // Продолжение рабочей
. // функции потока 1
}
```

Поток 2

```
DWORD WINAPI Thread2(...){
.
. // Строки рабочей функции
. // потока 2

EnterCriticalSection(&CritSect);
. // Фрагмент рабочей функции
. // потока 2, в котором выполняется
. // обращение к общим данным
LeaveCriticalSection(&CritSect)

. // Продолжение рабочей
. // функции потока 2
}
```

Рис. 14.1. Организация программы с критическими секциями

Мы рассмотрели формальную сторону дела. А почему вообще надо защищать данные от другого потока? Если общие для нескольких потоков данные только читаются, то к таким данным может обращаться любое число потоков. Проблемы возникают в том случае, если потоки как-то видоизменяют данные – модифицируют отдельные элементы, уничтожают их или добавляют. Пусть общие данные олицетворяют собой счет в банке, доступный нескольким клиентам – мужу и жене, например, или нескольким агентам по закупке некоторой продукции. Клиенты снимают деньги в банке с помощью программы, в которой каждому клиенту отведен отдельный поток. Представим себе, что все клиенты имеют право снимать зараз только по одному рублю. Такая операция в программе на языке C++ описывается оператором

```
Account--
```

после трансляции которого в загрузочном модуле появится команда процессора

```
dec [Account]
```

Такое действие, которое требует единственной команды процессора, иногда называется *атомарным*. Сколько бы потоков-клиентов ни обращалось к общему счету, и как бы система ни прерывала выполнение потоков, сумма счета будет убывать правильным образом, потому что передача управления от потока к потоку осуществляется *между* командами процессора. Влезть внутрь атомарной операции и исказить ее выполнение невозможно.

Между прочим, ограничение снимаемой суммы одним рублем мы ввели только для наглядности. Снятие большей суммы выражается на языке C++ оператором

```
Account-=10000
```

а в машинных кодах – парой команд

```
mov eax,10000
sub [Account],eax
```

Даже если система прервет выполнение потока между этими командами, то после возврата управления в прерванный поток операция вычитания, которая по-прежнему является атомарной, выполнится правильно: текущая сумма счета Account уменьшится на хранящуюся в eax величину.

Другая ситуация возникнет, если перед снятием со счета указанной клиентом суммы выполняется проверка наличия такой суммы, что, разумеется, на практике всегда делается. Будем считать, что требуемая клиентом сумма поступает сначала в переменную Delta. В программе на языке C++ операции проверки и снятия денег будут выглядеть примерно так:

```
Delta=10000;
if(Account>=Delta)
  Account-=Delta;
```

После трансляции эти предложения преобразуются приблизительно в такую последовательность машинных команд:

```
(1) mov [Delta],10000;Снимаемая сумма сохраняется в ячейке Delta
(2) mov eax,[Account];Сумма на счету отправляется в регистр eax
(3) cmp eax,[Delta];Содержимое eax сравнивается со снимаемой суммой в Delta
(4) jl m1;Если на счету меньше Delta, обход операции снятия со счета
(5) mov edx,[Delta];Снимаемая сумма поступает в регистр edx
(6) sub [Account],edx;Сумма Delta снимается со счета
(7) m1:...;Метка m1. Продолжение программы
```

Представим теперь, что где-либо между командами (2) и (6), когда поток первого клиента уже прочитал сумму счета из памяти в регистр, но еще не успел уменьшить ее на запрашиваемую величину, система передала управление потоку второго клиента. Поскольку в переменной Account еще прежняя сумма счета, второй поток ее проанализирует и, обнаружив, что она достаточно велика, уменьшит в соответствии с запросом второго клиента (например, до нуля). После того как управление вернется в первый поток, он продолжит анализ старой суммы, хранящейся в регистре eax, и тоже снимет деньги со счета, хотя их там, возможно, уже и нет.

Разумеется, вероятность такого события чрезвычайно мала. Время выполнения четырех команд процессора составляет порядок $10^{-8}...10^{-7}$ с, и практически невероятно, чтобы именно на этот интервал времени попало переключение потоков. Однако такое событие все же возможно, а его реализация может привести к трагическим последствиям. Защитив приведенный выше программный фрагмент с помощью критической секции, мы устраним возможность краха банка.

Из приведенного примера должно быть ясно, что вероятность сбоя при обращении нескольких потоков к общим данным возрастает, с одной стороны, при увеличении количества потоков, а с другой – при усложнении операций с общими данными. В нашем примере обращение к общим данным уложилось в 5 машинных команд. Если же потоки выполняют с данными сложные преобразования (например, сортируют их каждый по своим критериям), то время выполнения этих преобразований может стать весьма значительным, а вероятность сбоя ощутимой.

Рассмотрим программную реализацию описанного выше примера с двумя клиентами, снимающими деньги с общего счета.

```
/*Программа 14-1. Критические секции*/
#include <windows.h>
DWORD WINAPI Thread1(LPVOID);//Рабочая функция потока 1
DWORD WINAPI Thread2(LPVOID);//Рабочая функция потока 2
/*Глобальные переменные*/
CRITICAL_SECTION CritSec;//Служебная структура
int nBalance;//Защищаемая переменная: текущий остаток средств
const int nAccount=500000;//Начальная сумма счета
const int nDelta=1;//Сумма, снимаемая каждым клиентом зараз
BOOL bGo,bTerminate;//Булевы переменные для управления процессом
int nDeals1,nDeals2;//Число сделок первого и второго клиента
/*Главная функция приложения WinMain*/
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
  HANDLE hThreads[2];//Массив двух дескрипторов потоков
  DWORD dwIDThread1,dwIDThread2;//Идентификаторы потоков
  int nExpences;//Снятая сумма
  char szText[100];//Строка для вывода в окно сообщения
  nBalance=nAccount;//Начальное значение текущего остатка средств
  InitializeCriticalSection(&CritSec);//Инициализируем критические секции
  hThreads[0]=CreateThread(NULL,0,Thread1,NULL,0,&dwIDThread1);//Создаем поток 1
  hThreads[1]=CreateThread(NULL,0,Thread2,NULL,0,&dwIDThread2);//Создаем поток 2
  MessageBox(NULL,"Пуск","Команда",MB_OK);//Подождем создания потоков
  bGo=TRUE;//Одновременный пуск обоих потоков
  while(nBalance>=nDelta);//Подождем снятия всех денег
  bTerminate=TRUE;//Завершим оба потока
  WaitForMultipleObjects(2,hThreads,TRUE,INFINITE);//Ждем завершения потоков
  CloseHandle(hThreads[0]);//Закроем дескрипторы потоков
  CloseHandle(hThreads[1]);
  DeleteCriticalSection(&CritSec);//Удалим объект "критическая секция"
  nExpences=nDelta*(nDeals1+nDeals2);//Подсчитаем полную снятую сумму
  wsprintf(szText,"Начальная сумма=%d\nСделок 1=%d\nСделок 2=%d\n"
```

```
                "Остаток средств=%d\nСнято со счета %d\n",
                 nAccount,nDeals1,nDeals2,nBalance,nExpences);
  MessageBox(NULL,szText,"Results",MB_OK);//Вывод итогового отчета
  return 0;
  }
/*Рабочая функция потока 1*/
DWORD WINAPI Thread1(LPVOID){
  while(!bTerminate){//Выполнять, пока не дана команда прекратить
    if(bGo){//Если была дана команда пуска
      EnterCriticalSection(&CritSec);//Войдем в критическую секцию
      int nLocalBalance=
                  nBalance;//Прочитаем в локальную переменную остаток средств
      if(nLocalBalance>=nDelta) {//Если есть что снимать и была команда пуска
        nLocalBalance-=nDelta;//Снимаем сумму
        nDeals1++;//Отсчитываем сделку
        nBalance=nLocalBalance;//Заносим новый текущий остаток на счет
        }//Конец if(nLocalBalance>=nDelta)
      LeaveCriticalSection(&CritSec);//Выйдем из критической секции
      }//Конец if(bGo)
    }//Конец цикла while
  return 0;//Завершающий оператор рабочей функции потока
  }//Конец рабочей функции потока
/*Рабочая функция потока 2*/
DWORD WINAPI Thread2(LPVOID){
  while(!bTerminate){//Выполнять, пока не дана команда прекратить
    if(bGo){//Если была дана команда пуска
      EnterCriticalSection(&CritSec);//Войдем в критическую секцию
      int nLocalBalance=
                  nBalance;//Прочитаем в локальную переменную остаток средств
      if(nLocalBalance>=nDelta){//Если есть что снимать и была команда пуска
        nLocalBalance-=nDelta;//Снимаем сумму
        nDeals2++;//Отсчитываем сделку
        nBalance=nLocalBalance;//Заносим новый текущий остаток на счет
        }//Конец if(nLocalBalance>=nDelta)
      LeaveCriticalSection(&CritSec);//Выйдем из критической секции
      }//Конец if(bGo)
    }//Конец цикла while
  return 0;//Завершающий оператор рабочей функции потока
  }//Конец рабочей функции потока
```

В секции глобальных переменных определена структурная переменная CritSec типа CRITICAL_SECTION, необходимая для реализации критических секций, защищаемая переменная nBalance, а также ряд константных и обычных переменных, которые будут использоваться как в главной функции приложения, так и в рабочих функциях потоков. Остальные переменные описываются как локальные.

В главной функции WinMain() прежде всего выполняется инициализация объекта "критическая секция" и создание двух рабочих потоков Thread1 и Thread2. Создание потоков требует значительного времени, и, если не предусмотреть никакой синхронизации, первый запущенный поток успеет выполнить всю работу (снятие с счета всех имеющихся там денег) еще до запуска второго потока. В примере используется простейший вид синхронизации или, точнее, управления программой – окно сообщения, которое, как известно, представляет собой модальный диалог, блокирующий выполнение программы до нажатия предусмотренной кнопки OK.

После нажатия кнопки и разблокирования программы устанавливается флаг пуска, в качестве которого используется булева переменная bGo. Ее значение проверяется в самом начале рабочей функции потока, и, пока эта переменная сброшена, все действия по

снятию со счета денег пропускаются. Установка флага пуска приводит к одновременному разрешению "банковских операций" для обоих потоков.

Действие рабочих функций потоков заключается в циклическом снятии со счета одной и той же суммы до его исчерпания. На это время следует приостановить работу главной функции, чтобы она не завершила работу всего приложения слишком рано; остановка реализуется с помощью цикла while, в котором мы ожидаем уменьшения остатка средств до величины, меньшей, чем снимаемые суммы. Между прочим, учитывая возможности Win32, такой метод ожидания чего-то – худшее, что можно было придумать. Действительно, каждый раз, когда первичному потоку будет предоставляться квант времени, он будет добросовестно его использовать, не выполняя при этом никакой полезной работы. Существенно лучшим решением вопроса синхронизации в данном случае было бы использование механизма событий. Как это сделать, будет показано ниже в разделе этой главы, посвященной событиям.

Как только остаток средств (переменная nBalance) станет меньше nDelta, программа устанавливает булеву переменную bTerminate, которая служит флагом завершения потоков. Рабочие функции потоков организованы в виде бесконечного цикла while(), который выполняется, пока флаг bTerminate находится в исходном, сброшенном состоянии. Установка флага приводит к завершению рабочей функции потока переходом на оператор return.

Далее в главной функции закрываются дескрипторы потоков и удаляется объект "критическая секция". Однако удалить критическую секцию можно лишь в том случае, если ни один поток не владеет ею. Удаление критической секции, занятой каким-либо потоком или, наоборот, выход потока из удаленной уже критической секции приведет к серьезному системному сбою. Для того чтобы удаление критической секции было выполнено заведомо после завершения обоих порожденных потоков, в первичном потоке вызывается синхронизирующая функция WaitForMultipleObjects(), которая приостанавливает первичный поток до перевода в свободное состояние обоих указанных в ней дескрипторов объектов (массив дескрипторов потоков hThreads).

Перед своим завершением главная функция приложения формирует и выводит в окно сообщения итоговую сводку по выполненным операциям. Вывод программы для конкретного прогона приведен на рис. 14.2. Видно, что первый клиент совершил 240 159 операций снятия денег, второй клиент – 259 841 операцию; всего, таким образом, были выполнены точно 500 000 операций. Поскольку в каждой операции снималась 1 денежная единица, а начальная сумма составляла 500 тыс. единиц, операции в итоге привели к обнулению вклада.

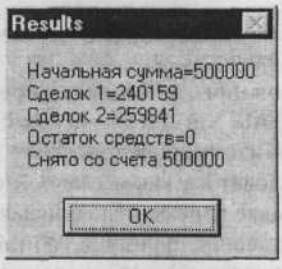


Рис. 14.2. Работа двух потоков с общими защищенными данными

Рис. 14.2 демонстрирует работу программы, но не убеждает в необходимости использования критических секций. Для того чтобы наблюдать роль критических секций, следует исключить из программы строки их активизации. Достаточно поставить знаки комментария в текстах рабочих функций потоков перед предложениями входа в критическую секцию и выхода из нее (и разумеется, перетранслировать программу). На рис. 14.3 показаны два варианта вывода программы при выключенном механизме критических секций. Видно, что в обоих случаях сумма снятых со счета денег превышает (и сущест-

венно!) исходную сумму вклада, т. е. учет текущего остатка (глобальная переменная nBalance) велся с ошибками.

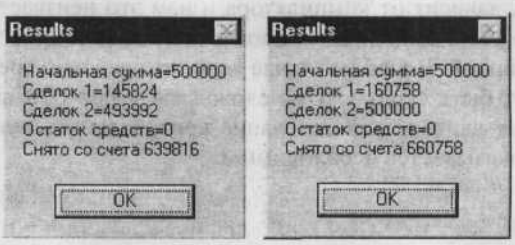


Рис. 14.3. Возможные результаты работы программы 14-1 при отсутствии защиты данных

Неправильный ход программы легко объяснить. Действительно, в рабочих функциях потоков текущая сумма вклада сначала читается в локальную переменную nLocalBalance, затем выполняется коррекция *этой* переменной на величину снимаемых средств, а также инкремент счетчика сделок nDeals*i*, и лишь после этого новое значение текущего остатка заносится назад в общую для обоих потоков переменную nBalance:

```
int nLocalBalance=nBalance;//Прочитаем в локальную переменную остаток средств
    if((nLocalBalance>=nDelta)&&bGo){//Если есть что снимать и была команда пуска
      nLocalBalance-=nDelta;//Снимаем сумму
      nDeals2++;//Отсчитываем сделку
      nBalance=nLocalBalance;//Заносим новый текущий остаток на счет
      ...
```

Если после чтения текущей суммы вклада в локальную переменную nLocalBalance выполнение потока будет прервано с передачей управления второму потоку, тот прочитает в свою локальную переменную nLocalBalance *то же* значение остатка вклада и будет с ним работать в неведении того, что первый поток уже снял со вклада некоторую сумму. В результате кажущаяся сумма вклада увеличивается.

Рассмотрим некоторые особенности программы, влияющие на ее работоспособность. Для того, чтобы продемонстрировать функционирование критических секций, требуется, чтобы в программе происходило переключение потоков, т. е. чтобы каждый поток в течение времени своего выполнения получил хотя бы 2–3 кванта времени процессора. С этой целью начальная величина вклада выбрана относительно большой, а снятие денег осуществляется минимальными порциями. При испытаниях программы 14-1 на компьютере среднего быстродействия на выполнение заданного числа операций расходовалось 6–10 квантов времени, что и позволило наблюдать неправильный учет средств (при выключенных критических секциях) почти в каждом прогоне. При использовании более быстрого процессора значение константной переменной nAccount (и соответственно, начальное значение переменной nBalance) следует увеличить.

Легко видеть, что роль критических секций нам удалось показать столь наглядно лишь потому, что алгоритм рабочей функции потока составлен далеко не оптимальным образом. Все неприятности возникли из-за того, что функция потока читает остаток средств в свою локальную переменную, затем с ней работает и лишь после этого записывает новое значение остатка в общую переменную nBalance, давая возможность системе переключить потоки между этими действиями. Если устранить локальную переменную и выполнять коррекцию остатка одним предложением непосредственно в переменной nBalance

```
nBalance-=nDelta;
```

то ошибок учета средств станет гораздо меньше. Возможно, они даже устранятся полностью, если операция вычитания из ячейки памяти является атомарной. Однако то, будет она атомарной или нет, зависит от компилятора и нам это неизвестно. Кроме того, если перед снятием денег со счета выполняется проверка их наличия, то эта пара операций уж точно не является атомарной. В любом случае ясно, что реально требуемые действия над общими данными могут быть сколь угодно сложными и при обращении нескольких потоков к одним и тем же данным использование критических секций (или других механизмов защиты) часто оказывается необходимым.

Мьютексы

Мьютексы, как уже отмечалось, по смыслу своего применения весьма похожи на критические секции: они служат для предотвращения одновременного использования какого-либо ресурса несколькими процессами. Сам термин "мьютекс" (mutex) является сокращением слов mutual exclusion (взаимное исключение). От критических секций мьютексы отличаются главным образом тем, что их можно использовать для синхронизации не только потоков, но и процессов. Для демонстрации техники применения мьютексов воспользуемся предыдущим примером с коллективным счетом в банке, заменив критические секции мьютексом.

```
/*Программа 14-2. Мьютексы в потоках*/
#include <windows.h>
DWORD WINAPI Thread1(LPVOID);//Рабочая функция потока 1
DWORD WINAPI Thread2(LPVOID);//Рабочая функция потока 1
/*Глобальные переменные*/
► HANDLE hMutex;//Дескриптор мьютекса
int nBalance;//Текущий остаток средств
const int nAccount=500000;//Начальная сумма счета
const int nDelta=1;//Сумма, снимаемая каждым клиентом зараз
BOOL bGo,bTerminate;//Булевы переменные для управления процессом
int nDeals1,nDeals2;//Число сделок первого и второго клиента
/*Главная функция приложения WinMain*/
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
  HANDLE hThreads[2];//Массив двух дескрипторов потоков
  DWORD dwIDThread1,dwIDThread2;//Идентификаторы потоков
  int nExpences;//Снятая сумма
  char szText[100];//Строка для вывода в окно сообщения
  nBalance=nAccount;//Начальное значение текущего остатка средств
► CreateMutex (NULL,FALSE,NULL);//Создаем мьютекс
  hThreads[0]=CreateThread(NULL,0,Thread1,NULL,0,&dwIDThread1);//Создаем поток 1
  hThreads[1]=CreateThread(NULL,0,Thread2,NULL,0,&dwIDThread2);//Создаем поток 2
  MessageBox(NULL,"Пуск","Команда",MB_OK);//Подождем создания потоков
  bGo=TRUE;//Одновременный пуск обоих потоков
  while(nBalance>=nDelta);//Подождем снятия всех денег
  bTerminate=TRUE;//Завершим оба потока
  WaitForMultipleObjects(2,hThreads,TRUE,INFINITE);//Ждем завершения потоков
   CloseHandle(hThreads[0]);//Закроем дескрипторы потоков
   CloseHandle(hThreads[1]);
►  CloseHandle(hMutex);//Закроем дескриптор мьютекса
   nExpences=nDelta*(nDeals1+nDeals2);//Подсчитаем полную снятую сумму
   wsprintf(szText,"Начальная сумма=%d\nСделок 1=%d\nСделок 2=%d\n"
                   "Остаток средств=%d\nСнято со счета %d\n",
                    nAccount,nDeals1,nDeals2,nBalance,nExpences);
  MessageBox(NULL,szText,"Results",MB_OK);
  return 0;
  }
```

```
/*Рабочая функция потока 1*/
DWORD WINAPI Thread1(LPVOID){
  while(!bTerminate){//Выполнять, пока не дана команда прекратить
    if(bGo){//Если дана команда пуска
      WaitForSingleObject(hMutex,INFINITE);//Будем ждать освобождения мьютекса
      int nLocalBalance=
              nBalance;//Прочитаем в локальную переменную остаток средств
      if(nLocalBalance>=nDelta){//Если есть что снимать и была команда пуска
        nLocalBalance-=nDelta;//Снимаем сумму
        nDeals1++;//Отсчитываем сделку
        nBalance=nLocalBalance;//Заносим новый текущий остаток на счет
        }//Конец if(nLocalBalance>=nDelta)
      ReleaseMutex(hMutex);//Освободим мьютекс
      }//Конец if(bGo)
    }//Конец цикла while
  return 0;//Завершающий оператор рабочей функции потока
  }//Конец рабочей функции потока
/*Рабочая функция потока 2*/
DWORD WINAPI Thread2(LPVOID){
  while(!bTerminate){//Выполнять, пока не дана команда прекратить
    if(bGo){//Если дана команда пуска
      WaitForSingleObject(hMutex,INFINITE);//Будем ждать освобождения мьютекса
      int nLocalBalance=
              nBalance;//Прочитаем в локальную переменную остаток средств
      if(nLocalBalance>=nDelta){//Если есть что снимать и была команда пуска
        nLocalBalance-=nDelta;//Снимаем сумму
        nDeals2++;//Отсчитываем сделку
        nBalance=nLocalBalance;//Заносим новый текущий остаток на счет
        }//Конец if(nLocalBalance>=nDelta)
      ReleaseMutex(hMutex);//Освободим мьютекс
      }//Конец if(bGo)
    }//Конец цикла while
  return 0;//Завершающий оператор рабочей функции потока
  }//Конец рабочей функции потока
```

Легко заметить, что программа с мьютексом отличается от предыдущей программы с критическими секциями лишь несколькими предложениями (все эти предложения помечены в тексте программы флажками). Вместо объявления структурной переменной типа CRITICAL_SECTION в глобальные данные включена переменная hMutex, в которую поступит дескриптор созданного мьютекса. Предложение с инициализацией объекта "критическая секция" заменено на вызов функции CreateMutex() создания мьютекса. Прототип этой функции выглядит следующим образом:

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,//Адрес структуры с атрибутами защиты
  BOOL  bInitialOwner,//Флаг начального владения
  LPCTSTR  lpName//Имя объекта "мьютекс"
  );
```

Второй параметр этой функции определяет исходный режим владения нового мьютекса. Если значение параметра равно TRUE, право владения мьютексом отдается породившему его потоку; в нашем случае первичный поток мьютексом владеть не должен, а порожденные потоки еще не созданы, потому значение этого флага должно быть равно FALSE.

Третий параметр функции позволяет задать имя создаваемого объекта, что, как уже отмечалось, используется для обращения к объекту из другого *процесса*. Нам имя не понадобится, так как в примере 14-2 демонстрируется использование мьютекса для синхронизации двух потоков одного процесса.

Перед завершением главной функции выполняется закрытие дескриптора ненужного уже мьютекса.

Как выглядит собственно синхронизация потоков с помощью объекта "мьютекс"? Пока мьютексом не владеет ни один поток (а именно таково исходное состояние мьютекса в нашем примере), он находится в свободном состоянии. При выделении первого кванта времени порожденному потоку (какой из двух наших потоков первым получит квант времени – дело случая) в рабочей функции потока выполняется функция ожидания WaitForSingleObject() с указанием в качестве анализируемого объекта дескриптора мьютекса. Поскольку мьютекс находится в свободном состоянии, поток продолжает выполнение, а функция ожидания переводит мьютекс в занятое состояние. Когда рабочая функция потока выполнит все действия по снятию со счета суммы Delta, вызовом функции ReleaseMutex() она переведет мьютекс в свободное состояние и даст возможность выполняться второму потоку. Все время, пока идет обработка общих данных (в данном случае переменной nBalance) первым потоком, второй поток, даже при выделении ему квантов времени, будет находиться в спящем состоянии.

На рис. 14.4 приведен результат работы программы 14-2. Практически он не отличается от результатов примера 14-1, если не считать того, что время между потоками распределилось несколько более равномерно.

Выполняя на одном и том же компьютере программы 14-1 и 14-2, легко заметить, что второй вариант работает существенно (по меньшей мере на порядок) медленнее первого.

Рис. 14.4. Возможный результат работы программы с синхронизацией потоков с помощью мьютекса

Это объясняется тем, что организация и обслуживание мьютексов, которые относятся к объектам ядра Windows, требует от системы значительно большего времени, чем работа с критическими секциями. Функции Windows EmterCriticalSection() и LeaveCriticalSection() выполняются обычно в режиме пользователя, и их вызов требует приблизительно 9 машинных команд, в то время как функции WaitForSingleObject() и ReleaseMutex() требуют перехода из режима пользователя в режим ядра, на что расходуется около 600(!) машинных команд. Если учесть, что в нашем примере защищенный мьютексом участок программы имеет небольшую длину и содержит всего лишь около тридцати машинных команд (что можно определить, рассмотрев эту часть программы в отладчике), то станет ясно, что использование мьютекса может замедлить выполнение программы в несколько десятков раз, что мы и наблюдаем на практике.

События

Объект "событие" относится к числу весьма эффективных, но, как мы сейчас увидим, не очень простых средств синхронизации потоков и процессов. С помощью этого объекта поток уведомляет другой поток об окончании любой операции или вообще о возникновении любой оговоренной заранее ситуации, могущей повлиять на выполнение другого потока. Другими словами, события не привязаны к определенным действиям, например завершению процесса, окончанию операции ввода-вывода и др. Программист имеет возможность устанавливать и сбрасывать события в любой точке программы, организуя тем

самым взаимосвязь потоков нужным ему образом. Заметим, что, поскольку при создании события ему можно дать произвольное имя и затем открыть это событие в другом процессе, события, как и мьютексы, можно использовать для синхронизации не только потоков одного процесса, но и самостоятельных процессов, точнее, потоков, работающих в контексте самостоятельных процессов.

Рассмотрим сначала формальный аппарат работы с событиями. Разумеется, перед тем, как использовать событие в целях синхронизации, его надо создать. Событие создается с помощью функции CreateEvent() со следующим прототипом:

```
HANDLE CreateEvent(
  LPSECURITY_ATTRIBUTES lpEventAttributes,//Адрес атрибутов защиты
  BOOL bManualReset,//Флаг ручного сброса события
  BOOL bInitialState,//Флаг начального состояния события
  LPCTSTR lpName//Имя события
  );
```

Существует два типа событий: со сбросом вручную и с автоматическим сбросом (автосбросом). События со сбросом вручную требуют явной установки их в свободное состояние функцией SetEvent() и столь же явного переключения в занятое состояние функцией ResetEvent(); очевидно, что эти переключения можно выполнять в произвольных точках программы. События с автосбросом точно так же устанавливаются явным образом функцией SetEvent(), однако их сброс происходит автоматически, как только в управляемом потоке функция ожидания события обнаружила, что событие установлено и, соответственно, разбудила поток. Событиями с автосбросом удобно пользоваться в циклических фрагментах программ, так как такое событие, автоматически сброшенное в начале фрагмента, предотвращает повторное его выполнение до новой установки этого события в управляющем потоке.

Если в функции CreateEvent() параметр bManualReset равен TRUE, создается событие со сбросом вручную; если этот параметр равен FALSE, создается событие с автосбросом.

Параметр bInitialState задает начальное состояние создаваемого события. Если он равен TRUE, создается установленное событие; если этот параметр равен FALSE, созданное событие будет сброшено.

Функция CreateEvent, как и другие функции создания объектов Windows, возвращает дескриптор созданного события, который затем используется в функциях переустановки и ожидания.

Для установки события в свободное (сигнальное) состояние используется функция SetEvent(), в качестве параметра которой указывается дескриптор устанавливаемого события. Если событие удалось установить, функция возвращает значение TRUE.

Для сброса события в занятое (несигнальное) состояние используется функция ResetEvent(), в качестве параметра которой указывается дескриптор сбрасываемого события. Если событие удалось сбросить, функция возвращает значение TRUE.

Для того чтобы открыть в некотором процессе событие *с именем*, созданное в другом процессе, используется функция OpenEvent(), в качестве последнего параметра которой используется имя созданного ранее события.

Наконец, предусмотрена функция PulseEvent(), которая устанавливает указанное в качестве параметра событие в свободное состояние, и после того, как *все* ожидающие потоки проснулись, тут же снова сбрасывает его, предотвращая повторное выполнение защищенных участков потоков до явной установки этого события в управляющем потоке.

Процедуры взаимодействия потоков с помощью событий, в зависимости от поставленной задачи, могут принимать самые разные формы. В простейшем случае взаимодей-

ствие носит односторонний и однократный характер: исходный поток, выполнив некоторые действия, например подготовив данные, устанавливает событие и тем самым извещает другой, спящий поток о том, что тот может проснуться и начать обработку этих данных.

Схема такого взаимодействия потоков приведена на рис. 14.5.

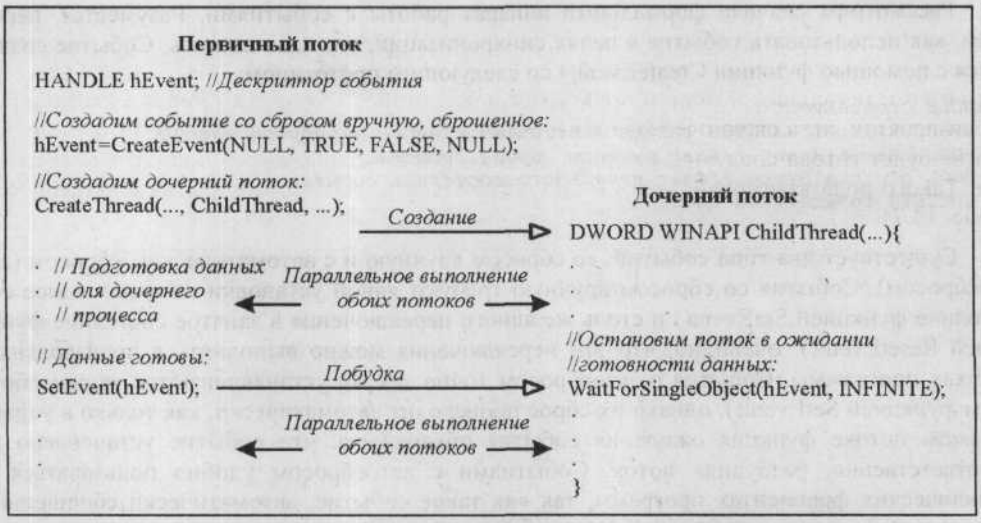


Рис. 14.5. Однократное одностороннее взаимодействие потоков

Если результат обработки нужен исходному потоку, то поток-приемник, завершив обработку, в свою очередь устанавливает (другое) событие, сообщая таким образом исходному потоку, что тот может использовать результаты обработки (рис. 14.6).

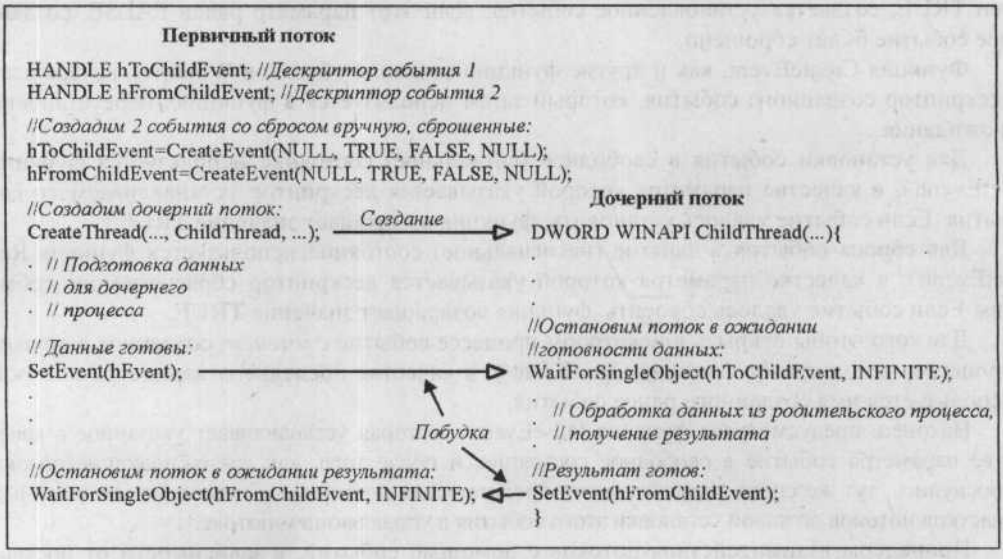


Рис. 14.6. Однократное двухстороннее взаимодействие потоков

Часто действия, выполняемые в двух или большем числе потоков, носят циклический, повторяющийся характер. Например, исходный поток, подготовив очередную порцию данных и записав ее в общий глобальный буфер, оповещает остальные (спящие) потоки о том, что данные готовы и можно приступить к работе с ними, а сам тем временем приступает к подготовке очередной порции данных. Однако глобальный буфер пока занят предыдущей порцией и записывать в него нельзя. Поэтому исходный поток засыпает до завершения операций остальных потоков. Когда все потоки закончат свои операции, исходный поток просыпается, записывает готовые данные в глобальный буфер, оповещает об этом остальные потоки и приступает в подготовке следующей порции данных. Потоки-приемники, в свою очередь, обработав данные, приостанавливаются до тех пор, пока не будет готова следующая порция данных, после чего процесс обработки повторяется. Такого рода взаимодействие удобно осуществлять, используя события с автосбросом (рис. 14.7).

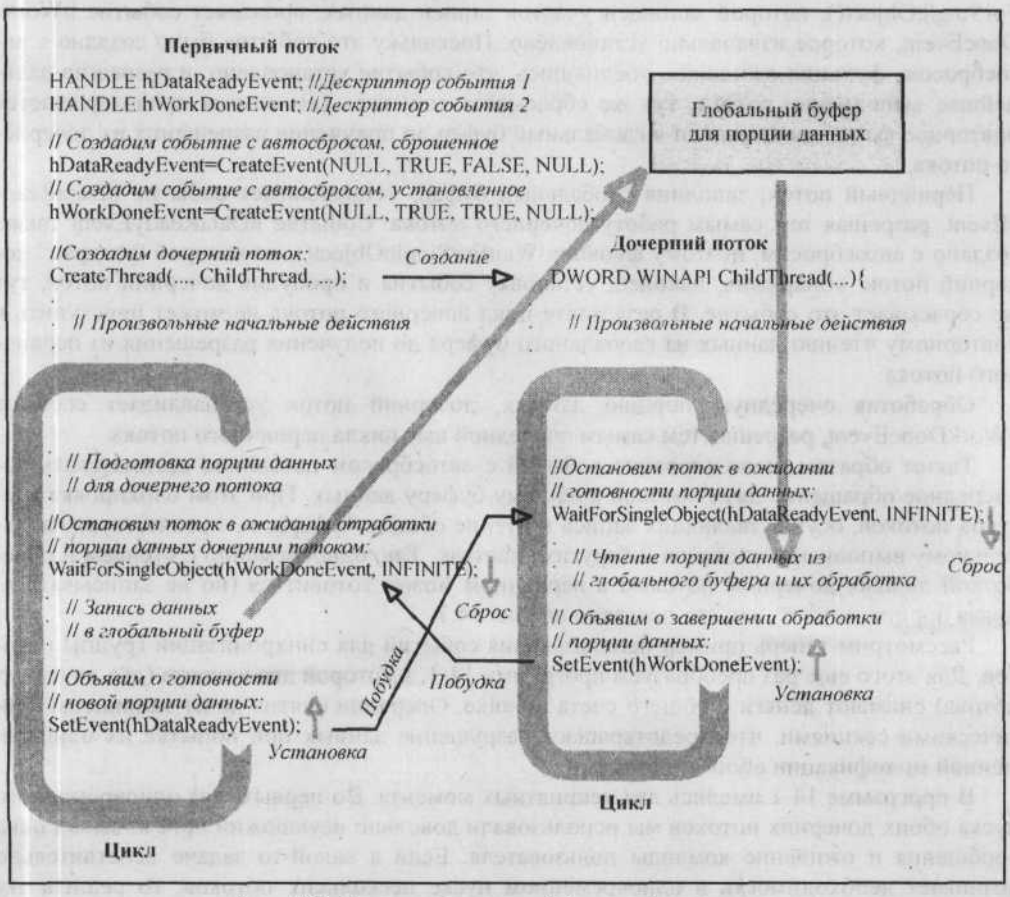


Рис. 14.7. Циклическое взаимодействие двух потоков

Первичный поток создает два события с автосбросом. Первое, с дескриптором hDataReadyEvent, будет сигнализировать дочернему потоку о том, что первичный поток подготовил очередную порцию данных. Поскольку в начале работы программы этих данных

пока нет, событие hDataReadyEvent создается в сброшенном, занятом состоянии. Второе событие, с дескриптором hWorkDoneEvent, будет сигнализировать из дочернего потока в первичный о том, что поток выполнил обработку порции данных и, следовательно, первичный поток может заняться подготовкой новой порции. Поскольку в начале работы программы дочернему потоку обрабатывать нечего и родительскому ничто не должно мешать записывать свои данные в глобальный буфер, событие hWorkDoneEvent создается в установленном состоянии, разрешающем работу первичному потоку.

Следующая операция первичного потока – создание дочернего. Дочерний поток может выполнить необходимые инициализирующие действия, после чего переводится в спящее состояние функцией ожидания события hDataReadyEvent, которое при его создании было сброшено.

Родительский поток, наоборот, после инициализации и подготовки первой порции данных может приступить к записи их в глобальный буфер, так как функция WaitForSingleObject(), которой защищен участок записи данных, проверяет событие hWorkDoneEvent, которое изначально установлено. Поскольку это событие было создано с автосбросом, функция ожидания, убедившись, что событие установлено, и разрешив дальнейшее выполнение потока, тут же сбрасывает событие. Тем самым предотвращается повторное выполнение записи в глобальный буфер до получения разрешения из дочернего потока.

Первичный поток, заполнив глобальный буфер, устанавливает событие hDataReadyEvent, разрешая тем самым работу дочернего потока. Событие hDataReadyEvent также создано с автосбросом, поэтому функция WaitForSingleObject(), на которой "застрял" дочерний поток, обнаружив, наконец, установку события и пробудив дочерний поток, тут же сбрасывает это событие. В результате цикл дочернего потока не может приступить к повторному чтению данных из глобального буфера до получения разрешения из первичного потока.

Обработав очередную порцию данных, дочерний поток устанавливает событие hWorkDoneEvent, разрешая тем самым очередной шаг цикла первичного потока.

Таким образом, использование событий с автосбросом позволило организовать поочередное обращение двух потоков к общему буферу данных. При этом блокировка участков потоков, осуществляющих запись и чтение общего буфера, не препятствует параллельному выполнению потоков на других участках. Так, одновременно с чтением и обработкой данных дочерним потоком в первичном может готовиться (но не записываться) новая порция данных, как это показано на рис. 14.7.

Рассмотрим теперь пример использования событий для синхронизации группы потоков. Для этого еще раз преобразуем программу 14-1, в которой два клиента (два дочерних потока) снимают деньги с общего счета в банке. Операции снятия денег защищены критическими секциями, что предотвращает разрушение данных при попытке их одновременной модификации обоими потоками.

В программе 14-1 имелись два неприятных момента. Во-первых, для одновременного пуска обоих дочерних потоков мы использовали довольно неуклюжий прием: вывод окна сообщения и ожидание команды пользователя. Если в какой-то задаче действительно возникает необходимость в одновременном пуске нескольких потоков, то решить эту проблему лучше с помощью аппарата событий.

Во-вторых, исчерпание счета в банке отслеживалось с помощью пустого цикла

```
while(nBalance>=nDelta);
```

Как уже отмечалось, это крайне неэффективный способ остановки программы, бессмысленно потребляющий время процессора. Останов программы в ожидании некоторого события следует реализовывать также с помощью аппарата событий.

В программе 14-3 события используются для достижения трех целей: оповещения первичного потока о том, что дочерние потоки созданы и готовы к работе; одновременного пуска дочерних потоков; завершения всей программы после того, как дочерние потоки исчерпали возможности снятия денег со счета. Поскольку предметов синхронизации в программе три, а дочерних потоков два, используются 3 *пары* событий и, соответственно, 3 массива дескрипторов событий, каждый из которых включает два элемента.

```
/*Программа 14-3. События*/
#include <windows.h>
DWORD WINAPI Thread1(LPVOID);//Рабочая функция потока 1
DWORD WINAPI Thread2(LPVOID);//Рабочая функция потока 2
/*Глобальные переменные*/
CRITICAL_SECTION CritSec;//Служебная структура
HANDLE hEventsToChild[2];//События оповещения дочерних потоков
HANDLE hEventsFromChild[2];//События оповещения из дочерних потоков
HANDLE hEventsAboutEnd[2];//События завершения работы дочерних потоков
int nBalance;//Текущий остаток средств
const int nAccount=500000;//Начальная сумма счета
const int nDelta=3;//Для разнообразия изменим снимаемую сумму
BOOL bGo,bTerminate;//Булевы переменные для управления процессом
int nDeals1,nDeals2;//Число сделок первого и второго клиента
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
  HANDLE hThreads[2];//Массив двух дескрипторов потоков
  DWORD dwIDThread1,dwIDThread2;//Идентификаторы потоков
  int nExpences;//Снятая сумма
  char szText[100];//Строка для вывода в окно сообщения
  nBalance=nAccount;//Начальное значение текущего остатка средств
  InitializeCriticalSection(&CritSec);//Инициализируем критические секции
  hThreads[0]=CreateThread(NULL,0,Thread1,NULL,0,&dwIDThread1);//Создаем поток 1
  hThreads[1]=CreateThread(NULL,0,Thread2,NULL,0,&dwIDThread2);//Создаем поток 2
/*Создаем массивы событий. Все события со сбросом вручную, сброшенные*/
/*События для оповещения первичного потока о готовности дочерних*/
  hEventsFromChild[0]=CreateEvent(NULL,TRUE,FALSE,NULL);
  hEventsFromChild[1]=CreateEvent(NULL,TRUE,FALSE,NULL);
/*События для пуска дочерних потоков*/
  hEventsToChild[0]=CreateEvent(NULL,TRUE,FALSE,NULL);
  hEventsToChild[1]=CreateEvent(NULL,TRUE,FALSE,NULL);
/*События для оповещения первичного потока о завершении дочерних*/
  hEventsAboutEnd[0]=CreateEvent(NULL,TRUE,FALSE,NULL);
  hEventsAboutEnd[1]=CreateEvent(NULL,TRUE,FALSE,NULL);
/*Создаем оба дочерних потока*/
  hThreads[0]=CreateThread(NULL,0,Thread1,NULL,0,&dwIDThread1);
  hThreads[1]=CreateThread(NULL,0,Thread2,NULL,0,&dwIDThread2);
/*Ждем извещений из обоих дочерних потоков о том, что они запущены*/
  WaitForMultipleObjects(2,hEventsFromChild,TRUE,INFINITE);
  SetEvent(hEventsToChild[0]);//Разбудим поток 1
  SetEvent(hEventsToChild[1]);//Разбудим поток 2
/*Ждем извещений из обоих дочерних потоков о том,
  что в банке нет денег*/
  WaitForMultipleObjects(2,hEventsAboutEnd,TRUE,INFINITE);
  bTerminate=TRUE;//Завершим оба дочерних потока
  WaitForMultipleObjects(2,hThreads,TRUE,INFINITE);//Ждем завершения потоков
/*Закроем дескрипторы потоков и событий*/
  for(int i=1;i<2;i++){
    CloseHandle(hThreads[i]);
```

```
    CloseHandle(hEventsToChild[i]);
    CloseHandle(hEventsFromChild[i]);
    CloseHandle(hEventsAboutEnd[i]);
    }
  DeleteCriticalSection(&CritSec);//Удалим объект "критическая секция"
  nExpences=Delta*(nDeals1+nDeals2);//Подсчитаем полную снятую сумму
  wsprintf(szText,"Начальная сумма=%d\nСделок 1=%d\nСделок 2=%d\n"
                  "Остаток средств=%d\nСнято со счета %d\n",
                   nAccount,nDeals1,nDeals2,nBalance,nExpences);
  MessageBox(NULL,szText,"Results",MB_OK);//Вывод итогового отчета
  return 0;
  }
/*Рабочая функция потока 1*/
DWORD WINAPI Thread1(LPVOID){
  SetEvent(hEventsFromChild[0]);//Поток создан, установим событие
  WaitForSingleObject(hEventsToChild[0],INFINITE);//Ждем команды пуска
  while(!bTerminate){//Выполнять, пока не дана команда прекратить
    EnterCriticalSection(&CritSec);//Войдем в критическую секцию
    int nLocalBalance=
          nBalance;//Прочитаем в локальную переменную остаток средств
    if(nLocalBalance>=nDelta){//Если есть что снимать
      nLocalBalance-=nDelta;//Снимаем сумму
      nDeals2++;//Отсчитываем сделку
      nBalance=nLocalBalance;//Заносим новый текущий остаток на счет
      }//Конец if(nLocalBalance>=nDelta)
    else SetEvent(hEventsAboutEnd[0]);//Денег больше нет, установим событие
      LeaveCriticalSection(&CritSec);//Выйдем из критической секции
    }
  return 0;
  }
/*Рабочая функция потока 2*/
DWORD WINAPI Thread2(LPVOID){
  SetEvent(hEventsFromChild[1]);//Поток создан, установим событие
  WaitForSingleObject(hEventsToChild[1],INFINITE);//Ждем команды пуска
  while(!bTerminate){//Выполнять, пока не дана команда прекратить
    EnterCriticalSection(&CritSec);//Войдем в критическую секцию
    int nLocalBalance=
          nBalance;//Прочитаем в локальную переменную остаток средств
    if(nLocalBalance>=Delta){//Если есть что снимать
      nLocalBalance-=Delta;//Снимаем сумму
      nDeals2++;//Отсчитываем сделку
      nBalance=nLocalBalance;//Заносим новый текущий остаток на счет
      }
    else SetEvent(hEventsAboutEnd[1]);//Денег больше нет, установим событие
      LeaveCriticalSection(&CritSec);//Выйдем из критической секции
    }
  return 0;
  }
```

Первичный поток, создав все необходимые события и оба дочерних потока, приостанавливается вызовом функции ожидания установки событий hEventsFromChild[i] из *обоих* дочерних потоков:

```
WaitForMultipleObjects(2,hEventsFromChild,TRUE,INFINITE);
```

Получив извещение о фактическом пуске дочерних потоков, первичный поток устанавливает события hEventsToChild[i], снимая блокировку дочерних потоков и позволяя им выполняться по мере выделения им квантов процессорного времени. Далее первичный поток впадает в спячку в ожидании завершения обоими дочерними потоками процедур снятия денег. Дочерние потоки тем временем в цикле снимают деньги со счета до тех пор, пока оставшаяся сумма не станет меньше снимаемый величины. В этом случае каж-

дый дочерний поток устанавливает свое событие, сигнализируя родительскому о завершении своей работы (не о завершении функции потока, которая продолжает выполняться, а поток, соответственно, существовать, а о том, что поток не сможет более выполнять содержательную работу). Первичный поток, получив эти сообщения из обоих дочерних, закрывает все дескрипторы, выводит на экран итоговый отчет и завершается.

На рис. 14.8 приведен результат работы программы при той же, что и раньше, начальной сумме, но при значении nDelta=3.

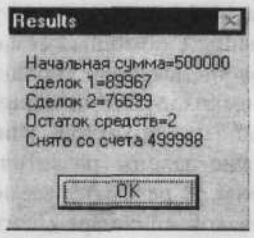


Рис. 14.8. Результат работы программы 14-3

Семафоры

Семафоры отличаются от остальных синхронизирующих объектов наличием у них не только двух состояний – установленного и сброшенного, но и счетчика ресурса, показывающего "степень установленности" семафора. Состояние семафора считается сброшенным, если значение его счетчика равно нулю. При любом значении счетчика, отличном от нуля, семафор установлен. Функции ожидания, примененные к семафору, не сбрасывают семафор в состояние "занято", а лишь уменьшают на единицу значение его счетчика. Таким образом, к семафору можно несколько раз применить функцию ожидания, последовательно уменьшая значение его счетчика, при этом семафор будет оставаться в сигнальном состоянии, разрешая использование защищаемого им ресурса. Лишь когда значение счетчика достигнет 0, семафор перейдет в сброшенное состояние (рис. 14.9).

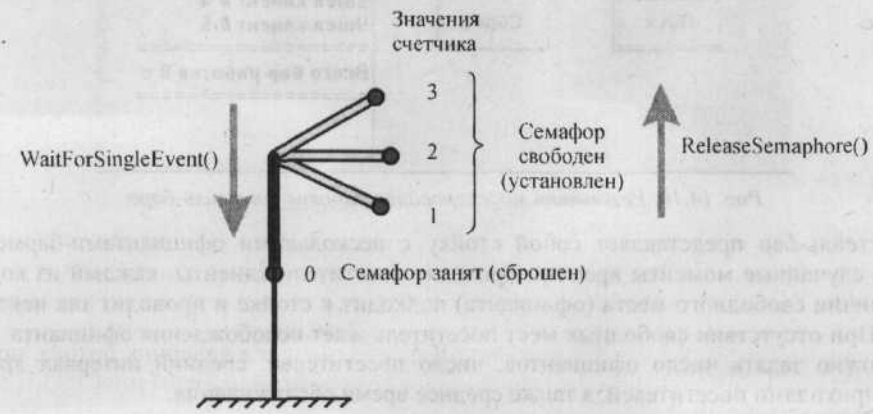


Рис. 14.9. Состояния семафора

Аналогично функции SetEvent(), переводящей событие в свободное состояние, для семафора предусмотрена функция ReleaseSemaphore(), которая дискретно повышает значение счетчика семафора, делая его все "более и более" установленным.

Семафоры удобно использовать в тех случаях, когда в программе имеется ограниченное количество экземпляров определенного ресурса. Создав для них семафор с начальным значением счетчика, равным количеству имеющихся экземпляров ресурса, мы

можем обеспечить одновременное использование любого числа экземпляров от одного до максимально возможного. Каждый запрос ресурса (с помощью функции ожидания) будет уменьшать значение счетчика семафора, и, как только все экземпляры исчерпаются, семафор перейдет в занятое состояние, блокируя дальнейшие попытки получить отсутствующий ресурс.

Программа 14-4 демонстрирует управление многопоточной моделью системы массового обслуживания с помощью семафоров (и событий). Вообще надо заметить, что концепция потоков чрезвычайно упрощает моделирование всякого рода процессов – физических, биологических, экономических и др., в которых принимает участие большое количество однотипных объектов (микрочастицы, биологические особи, предприятия), имеющих схожие законы развития, но различающихся числовыми характеристиками своего поведения. В рассматриваемой ниже программе моделируется процесс обслуживания клиентов коктейль-бара (бензоколонки, магазина самообслуживания, библиотеки или любой другой схожей системы) при различных значениях входных параметров. При запуске программы на экран выводится диалоговое окно, в котором можно задать требуемые значения параметров модели (рис. 14.10).

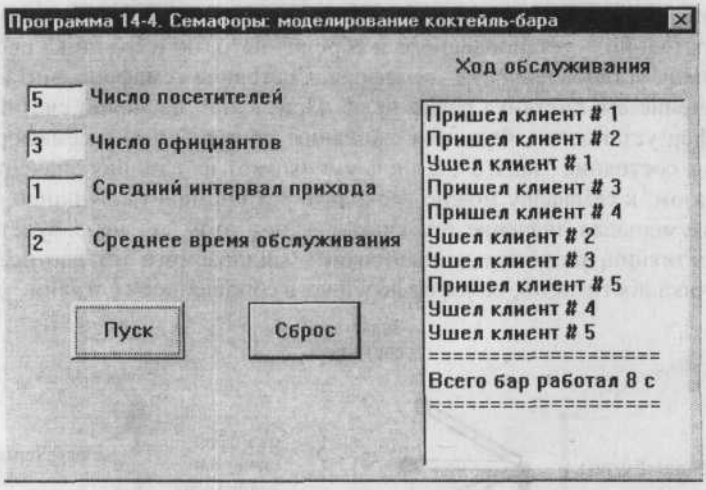


Рис. 14.10. Результат моделирования работы коктейль-бара

Коктейль-бар представляет собой стойку с несколькими официантами-барменами. В бар в случайные моменты времени приходят посетители-клиенты, каждый из которых при наличии свободного места (официанта) подходит к стойке и проводит так некоторое время. При отсутствии свободных мест посетитель ждет освобождения официанта. В модели можно задать число официантов, число посетителей, средний интервал времени между приходами посетителей, а также среднее время обслуживания.

Как известно, распределение интервалов времени между приходами отдельных клиентов в системах такого рода в идеале описывается экспоненциальным законом: с наибольшей вероятностью реализуются малые интервалы времени, а с увеличением интервалов их вероятность монотонно снижается. Для многих процессов (например, регистрации микрочастиц в ядерной физике) экспоненциальный закон выполняется с большой точностью; для коктейль-бара он, конечно, не очень применим, хотя бы потому что в за-

ведения такого рода посетители склонны приходить парами. Мы будем считать, что наши посетители приходят поодиночке.

Распределение времени обслуживания может носить самый разный характер, однако для многих систем можно принять, что время обслуживания каждого конкретного клиента колеблется в определенных пределах вокруг некоторой величины – среднего времени обслуживания $t_{ср}$ (на рис. 14.11 – сплошная линия). Размытость этой кривой определяется тем, насколько разные действия совершает каждый клиент. Например, для кафе быстрого обслуживания эта кривая будет относительно узкой, так как всем посетителям предлагается одно и то же меню. Для ресторана или коктейль-бара кривая будет шире, учитывая разброс вкусов и возможностей клиентов.

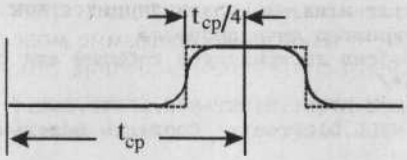


Рис. 14.11. Возможное распределение времени обслуживания

В рассматриваемой модели для простоты принято прямоугольное распределение с шириной, равной половине от среднего значения, как это показано на рис. 14.11 пунктиром.

```
/*Программа 14-4. Семафоры*/
/*Файл 14-4.h*/
#define ID_NCLIENTS 101
#define ID_NBARMEN 102
#define ID_TENTER 103
#define ID_TSERVICE 104
#define ID_LIST 105
#define ID_GO 106
#define ID_CLEAR 107
BOOL CALLBACK DlgProc(HWND,UINT,WPARAM,LPARAM);
void DlgOnCommand(HWND,int,HWND,UINT);
BOOL DlgOnInitDialog(HWND,HWND,LPARAM);
DWORD WINAPI ThreadBar(LPVOID);
DWORD WINAPI ThreadClient(LPVOID);
/*Файл 14-4.rc*/
#include "14-4.h"
Main DIALOG 6, 10, 226, 146
STYLE DS_MODALFRAME |WS_CAPTION |WS_SYSMENU |WS_VISIBLE
CAPTION "Программа 14-4. Семафоры: моделирование коктейль-бара" {
  LTEXT "Число посетителей", -1, 28, 16, 68, 10
  LTEXT "Число официантов", -1, 28, 31, 68, 10
  LTEXT "Средний интервал прихода", -1, 28, 46, 104, 10
  LTEXT "Среднее время обслуживания", -1, 28, 63, 108, 10
  LISTBOX ID_LIST, 136, 21, 87, 123, WS_VSCROLL
  EDITTEXT ID_NCLIENTS, 7, 16, 19, 10, WS_TABSTOP
  EDITTEXT ID_NBARMEN, 7, 31, 19, 10, WS_TABSTOP
  EDITTEXT ID_TENTER, 7, 46, 19, 10, WS_TABSTOP
  EDITTEXT ID_TSERVICE, 7, 63, 19, 10, WS_TABSTOP
  DEFPUSHBUTTON "Пуск", ID_GO, 21, 88, 37, 19
  LTEXT "Ход обслуживания ", -1, 148, 6, 66, 12
  DEFPUSHBUTTON "Сброс", ID_CLEAR, 79, 88, 37, 19
  }
```

```
/*Файл 14-4.cpp*/
#include <windows.h>
#include <windowsx.h>
#include <math.h>//Ради функции log()
#include "14-4.h"
HWND hListBox;//Дескриптор списка
HANDLE hSemBarmen;//Дескриптор семафоров
DWORD dwIDThread;//Идентификаторы создаваемых потоков
int nClients=5;//Число клиентов
int nBarmen=3;//Число официантов
int nTimeEnter=1;//Средний интервал между клиентами
int nTimeService=2;//Среднее время обслуживания клиента
DWORD dwOpenTime;//Время открытия бара
BOOL BtnDisable=FALSE;//Флаг нажатия кнопки "Пуск" (TRUE - нажата)
HANDLE hProcessHeap;//Дескриптор кучи процесса
HANDLE* hEventHandles;//Массив дескрипторов событий для потоков клиентов
/*Главная функция WinMain*/
int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int){
  DialogBox(hInst,"Main",NULL,DlgProc);//Создадим модальный диалог
  return 0;
  }
/*Диалоговая функция модального диалога*/
BOOL CALLBACK DlgProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_INITDIALOG,DlgOnInitDialog);
    HANDLE_MSG(hwnd,WM_COMMAND,DlgOnCommand);
    default:
      return FALSE;
    }
  }
/*Функция инициализации диалога*/
BOOL DlgOnInitDialog(HWND hwnd,HWND,LPARAM){
  char szEdit[17];
  hListBox=GetDlgItem(hwnd,ID_LIST);//Сохраним дескриптор списка
  itoa(nClients,szEdit,10);
  SendDlgItemMessage(hwnd,ID_NCLIENTS,EM_REPLACESEL,0,(LPARAM)szEdit);
  itoa(nBarmen,szEdit,10);
  SendDlgItemMessage(hwnd,ID_NBARMEN,EM_REPLACESEL,0,(LPARAM)szEdit);
  itoa(nTimeEnter,szEdit,10);
  SendDlgItemMessage(hwnd,ID_TENTER,EM_REPLACESEL,0,(LPARAM)szEdit);
  itoa(nTimeService,szEdit,10);
  SendDlgItemMessage(hwnd,ID_TSERVICE,EM_REPLACESEL,0,(LPARAM)szEdit);
  return TRUE;
  }
/*Функция обработки команд от диалога*/
void DlgOnCommand(HWND hwnd,int id,HWND,UINT Notify){
  char szLine[10];
  switch(id){
    case ID_NCLIENTS://Ввели число клиентов
      if(Notify==EN_CHANGE){
      GetDlgItemText(hwnd,ID_NCLIENTS,szLine,9);
      nClients=atoi(szLine);//Сохраним в nClients
      }
    break;
    case ID_NBARMEN://Ввели число официантов
      if(Notify==EN_CHANGE){
      GetDlgItemText(hwnd,ID_NBARMEN,szLine,9);
      nBarmen=atoi(szLine);//Сохраним в nBarmen
      }
    break;
    case ID_TENTER://Ввели средний интервал прихода клиентов
```

```
      if(Notify==EN_CHANGE){
      GetDlgItemText(hwnd,ID_TENTER,szLine,9);
      nTimeEnter=atoi(szLine);//Сохраним в nTimeEnter
      }
    break;
    case ID_TSERVICE://Ввели среднее время обслуживания
      if(Notify==EN_CHANGE){
      GetDlgItemText(hwnd,ID_TSERVICE,szLine,9);
      nTimeService=atoi(szLine);//Сохраним в nTimeService
      }
    break;
    case ID_GO:{//Нажали кнопку "Пуск"
/*Инициализация модели*/
      hProcessHeap=GetProcessHeap();//Получим дескриптор кучи процесса
      SendDlgItemMessage(hwnd,ID_GO,BM_SETSTATE,1,0L);//Нажали кнопку
      if(BtnDisable==FALSE){//Защита от преждевременного повторного пуска
        hSemBarmen=CreateSemaphore(NULL,nBarmen,nBarmen,NULL);//Создаем семафор
        HANDLE hThreadBar=CreateThread
              (NULL,0,ThreadBar,hwnd,0,&dwIDThread);//Поток-бар
        hEventHandles=(HANDLE*)HeapAlloc//Выделим память под дескрипторы
              (hProcessHeap,HEAP_ZERO_MEMORY,sizeof(HANDLE)*nClients);
        for(int i=0;i<nClients;i++)//Создадим события по числу клиентов
          hEventHandles[i]=CreateEvent
              (NULL,TRUE,FALSE,NULL);//Сброс вручную, сброшены
        dwOpenTime=GetTickCount();//Время пуска модели
        CloseHandle(hThreadBar);//Закроем дескриптор потока
        BtnDisable=TRUE;//Кнопка "Пуск" нажата (идет моделирование)
        }
      break;
      }
    case ID_CLEAR://Очистим список
      while(SendMessage(hListBox,LB_DELETESTRING,0,0)!=LB_ERR);
      break;
    case IDCANCEL://Закрываем приложение. Освободим ресурсы
      CloseHandle(hSemBarmen);
      for(int i=0;i<nClients;i++)
        CloseHandle(hEventHandles[i]);//Закроем дескрипторы событий
      HeapFree(hProcessHeap,0,hEventHandles);//Освободим память дескрипторов
      EndDialog(hwnd,0);
    }
  }
/*Рабочая функция потока стойки*/
DWORD WINAPI ThreadBar(LPVOID hwnd){
  char szT[80];
  char szSeparator[]="====================";
  randomize();//Внесем дополнительную случайность
  for(int i=0;i<nClients;i++){//В цикле для всех будущих клиентов
    float x=(float)random(1000)/1000;//Равномерное распределение, 0...1
    float ex=-nTimeEnter*log(1-x);//Экспоненциальное распределение, 0...∞
    if(ex>=10) ex=10;//Ограничим время ожидания для удобства отладки
    Sleep(ex*1000);//Ждем прихода следующего клиента
    HANDLE hThreadClient=CreateThread//Создадим поток очередного клиента
        (NULL,0,ThreadClient,(LPVOID)i,0,&dwIDThread);//i - его номер
    CloseHandle(hThreadClient);
    wsprintf(szT,"Пришел клиент # %d",i+1);
    SendMessage(hListBox,LB_ADDSTRING,0,(LPARAM)szT);//Вывод сообщения в список
    }
  WaitForMultipleObjects
      (nClients,hEventHandles,TRUE,INFINITE);//Ждем завершения всех клиентов
```

```
/*Все потоки клиентов завершились. Выведем в список итоги*/
    SendMessage(hListBox,LB_ADDSTRING,0,(LPARAM)szSeparator);
    wsprintf(szT,"Всего бар работал %i с",
              (int)(GetTickCount()-dwOpenTime+999)/1000);
    SendMessage(hListBox,LB_ADDSTRING,0,(LPARAM)szT);//Вывод сообщения в список
    SendMessage(hListBox,LB_ADDSTRING,0,(LPARAM)szSeparator);
    BtnDisable=FALSE;//Запомним, что кнопка отпущена
    SendDlgItemMessage((HWND)hwnd,ID_GO,BM_SETSTATE,0,0L);//Отпустим кнопку
  return 0;
  }
/*Рабочая функция потока клиента*/
DWORD WINAPI ThreadClient(LPVOID lpvParam){//Параметр - номер клиента
  char szT[80];
  randomize();//Внесем случайность
  WaitForSingleObject(hSemBarmen,INFINITE);//Декремент семафора
  float fTime=nTimeService*1000;//Среднее обслуживания в миллисекундах
  float fTimeServ=fTime-fTime*0.25+random(fTime*0.5);//Равномерное
       //распределение в границах nTimeService+-nTimeService*0.25
  Sleep((DWORD)fTimeServ);//Клиент обслуживается
  wsprintf(szT,"Ушел клиент # %d",(int)lpvParam+1);
  SendMessage(hListBox,LB_ADDSTRING,0,(LPARAM)szT);//Вывод сообщения в список
  ReleaseSemaphore(hSemBarmen,1,NULL);//Инкремент семафора
  SetEvent(hEventHandles[(int)lpvParam]);//Поток завершен, установим событие
  return 0;//Данный клиент ушел
  }
```

Главная функция приложения состоит всего из трех строк – в ней создается модальный диалог, который и представляет собой рабочее окно приложения.

Оконная функция диалога предусматривает обработку двух сообщений – WM_INITDIALOG об инициализации диалога и WM_COMMAND в ответ на работу пользователя с элементами управления – окнами ввода и кнопками.

На этапе инициализации диалога в функции DlgOnInitDialog() происходит заполнение окон редактирования начальными значениями параметров модели. Если пользователь вводит свои значения параметров, эти действия приводят к посылке в приложение сообщений WM_COMMAND (с идентификаторами окон ввода ID_CLIENTS, ID_BARMEN и др.), обработка которых заключается в приеме введенной символьной строки, преобразовании ее в числовое значение и помещении в соответствующую глобальную переменную.

Собственно моделирование начинается при нажатии пользователем кнопки "Пуск", когда управление передается на фрагмент программы, начинающийся с оператора

```
case ID_GO:
```

После перевода изображения кнопки в нажатое состояние выполняется целый ряд необходимых инициализирующих действий.

Прежде всего создается семафор hSemBarmen, управляющий доступом клиентов к официантам. Счетчик семафора устанавливается в соответствии с заданным пользователем числом официантов. При каждом создании нового клиента счетчик будет уменьшаться на единицу, и, когда число клиентов сравняется с числом официантов (счетчик семафора принял значение 0), новые клиенты создаваться не будут до освобождения хотя бы одного официанта.

Функцией CreateThread() создается поток стойки-бара с рабочей функцией ThreadBar(), который будет управлять созданием потоков клиентов. Этот поток призван также наблюдать за поведением клиентов, чтобы после завершения обслуживания последнего клиента вывести на экран итоговую запись. Оповещение потока стойки о завершении потоков клиентов осуществляется с помощью группы событий. Поскольку

число клиентов заранее неизвестно, память под дескрипторы событий выделяется динамически из кучи процесса функций HeapAlloc(). Объем выделяемой памяти определяется исходя из размера дескрипторов и их количества.

Далее в цикле создается требуемое число событий, дескрипторы которых помещаются в виде массива в выделенную память. Легко сообразить, что в данном случае нужны события со сбросом вручную, и в исходном состоянии все они сброшены.

Для определения полного времени обслуживания (именно оно является основным критерием оптимизации системы) в переменной dwOpenTime запоминается текущее системное время в миллисекундах от загрузки Windows. После закрытия дескриптора потока стойки, который нам более не понадобится, и занесения в булеву переменную BtnDisable значения TRUE, обработка сообщения WM_COMMAND завершается.

Собственно моделирование осуществляется в потоке стойки и в создаваемых в реальном времени потоках клиентов.

Создание потоков клиентов осуществляется в цикле, содержащем функцию Sleep() усыпления потока на заданное время. Это чрезвычайно эффективный способ создания временно́й задержки, поскольку спящий поток практически не потребляет процессорного времени, которое, таким образом, передается другим, активным потокам. Как уже отмечалось выше, интервалы времени между клиентами распределены приблизительно по экспоненциальному закону. Встроенная функция random() генерирует случайные числа в заданном диапазоне, распределенные равномерно. Для преобразования равномерного распределения в экспоненциальное используется формула

$ex=-t_{ср}*\ln(1-x)$

где x – величина, распределенная равномерно, а ex – экспоненциально. Величина $t_{ср}$ представляет собой среднее значение интервала времени между клиентами, задаваемое пользователем.

Создание каждого нового потока клиента сопровождается выводом в список диалогового окна строки с порядковым номером клиента (в программе клиенты нумеруются, естественно, от нуля, а при выводе в список – от единицы). Следует обратить внимание на то, что этот порядковый номер через параметр функции CreateThread() передается каждому создаваемому потоку, что дает возможность потокам выводить в список диалога свои сообщения с указанием номера потока-источника сообщения. Образовав заданное количество потоков клиентов, поток стойки впадает в спячку в ожидании установки *всеми* созданными потоками своих событий.

Все потоки клиентов представлены одной рабочей функцией ThreadClient(). В ней прежде всего функцией WaitForSingleObject() проверяется состояние семафора официантов. Если семафор установлен, что свидетельствует о наличии свободных официантов, функция WaitForSingleObject() уменьшает на единицу счетчик семафора, и функция потока продолжает свое выполнение. Если семафор сброшен (все официанты на данных момент заняты), поток приостанавливается, символизируя ожидание посетителем места у стойки.

Время обслуживания клиента моделируется той же функцией Sleep(), в качестве аргумента которой используется случайное число fTimeServe, распределенное так, как это показано на рис. 14.11 пунктиром. Для его вычисления от среднего времени обслуживания, заданного пользователем и переведенного в число миллисекунд, вычитается его четвертая часть и к результату прибавляется случайное число, равномерно распределенное в диапазоне от 0 до $t_{ср}/2$.

Пробудившись после завершения функции Sleep(), поток клиента выводит в список диалога сообщение об уходе клиента, выполняет инкремент семафора (один официант освободился), вызовом функции SetEvent() устанавливает свое событие и завершается оператором return.

Пока потоки клиентов возникают и завершаются, поток стойки заблокирован функцией WaitForMultipleObjects(). После завершения последнего из запланированных потоков клиентов поток стойки пробуждается и приступает к завершающим операциям. В список диалога последовательно выводятся: разделительная линия szSeparator, сообщение о полном времени обслуживания всех клиентов и еще раз разделительная линия. Переменная BtnDisable сбрасывается в 0, что является признаком завершения данного сеанса моделирования, а кнопка "Пуск" отпускается. Пользователь может, установив новые параметры модели, снова нажать кнопку "Пуск" и провести новый сеанс моделирования.

Для очистки окна списка предусмотрена кнопка "Сброс". При ее нажатии выполняется фрагмент программы

```
case ID_CLEAR://Очистим список
  while(SendMessage(hListBox,LB_DELETESTRING,0,0)!=LB_ERR);
  break;
```

в котором в цикле в окно списка посылается сообщение LB_DELETESTRING, удаляющее из списка строку с указанным номером. Казалось бы, что удалить надо строки со всеми номерами, однако после удаления первой строки с номером 0 (номер строки передается через предпоследний параметр функции SendMessage) список перенумерует свои строки, так что при следующем вызове функции SendMessage() надо снова удалять строку с номером 0. Число строк в списке нам неизвестно, поэтому вызов функции SendMessage() повторяется до тех пор, пока она не вернет код ошибки LB_ERR, свидетельствующий об отсутствии в списке строк.

При завершении приложения (путем нажатия кнопки  в диалоговом окне) освобождаются использованные нами ресурсы: закрываются дескрипторы семафора и всех событий, после чего вызовом функции HeapFree() освобождается выделенная память кучи.

Предметный указатель

П

Оглавление

Гусева А. И.
Учимся программировать: PASCAL 7.0
Задачи и методы их решения

Гусева А. И.
УЧИМСЯ ИНФОРМАТИКЕ

Гусева А. И.
СЕТИ И МЕЖСЕТЕВЫЕ КОММУНИКАЦИИ. Windows 2000

Дубейковский В. И.
Практика функционального моделирования с ALLFUSION PROCESS MODELER

Дунаев Сергей.
JAVA для Internet в Windows и Linux

Епанешников А., Епанешников В.
Программирование в среде TURBO PASCAL 7.0

Епанешников А., Епанешников В.
DELPHI. Проектирование СУБД

Епанешников А., Епанешников В.
Локальные вычислительные сети

Куправа Т. А. EXCEL. ***Практическое руководство***

Лавров К. Н., Цыплякова Т. П.
Финансовая аналитика.
MATLAB 6. (Под общ. ред. к. т. н. В. Г. Потемкина)

Лукин С. Н.
ТУРБО-ПАСКАЛЬ 7.0. ***Самоучитель для начинающих.***

Лукин С. Н.
VISUAL BASIC. ***Самоучитель для начинающих***

Лукин С. Н.
WORD и WINDOWS. ***Самоучитель для начинающих***

Маклаков С. В.
Моделирование бизнес-процессов с ALLFUSION PROCESS MODELER (BPWIN 4.1)

Маклаков С. В.
Создание информационных систем с ALLFUSION MODELING SUITE

Моргун А. Н.
MS WORD. Руководство к действию

Низаметдинова Н. Н.
СОВРЕМЕННАЯ РУССКАЯ ПУНКТУАЦИЯ

Пильщиков В. Н.
Программирование на языке АССЕМБЛЕРА

Потемкин В. Г.
Вычисления в среде MATLAB

Потемкин В. Г.
Инструментальные средства MATLAB 5.x

Потемкин В. Г.
MATLAB 6: среда проектирования инженерных расчетов

Рашевская М. А.
CORELDRAW. Практическое руководство

Федоров А., Елманова Н.
Введение в OLAP-технологии Microsoft

Фомичев В. М.
Дискретная математика и криптология

Фролов А. В., Фролов Г. В.
ЯЗЫК С #. Самоучитель

Хейфец А. Л.
Инженерная компьютерная графика. AUTOCAD

Цисарь И. Ф.,
Нейман В. Г. Компьютерное моделирование экономики

Черных И. В.
SIMULINK: среда создания инженерных приложений

Шикин Е. В., Боресков А. В.
КОМПЬЮТЕРНАЯ ГРАФИКА.
Полигональные модели

Шумаков П. В.
ADO.NET и создание приложений баз данных
в среде Microsoft Visual Studio.NET