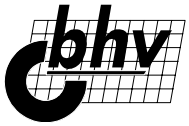
Ч. Петзолд

Программирование для Windows® 95

в двух томах

Том II



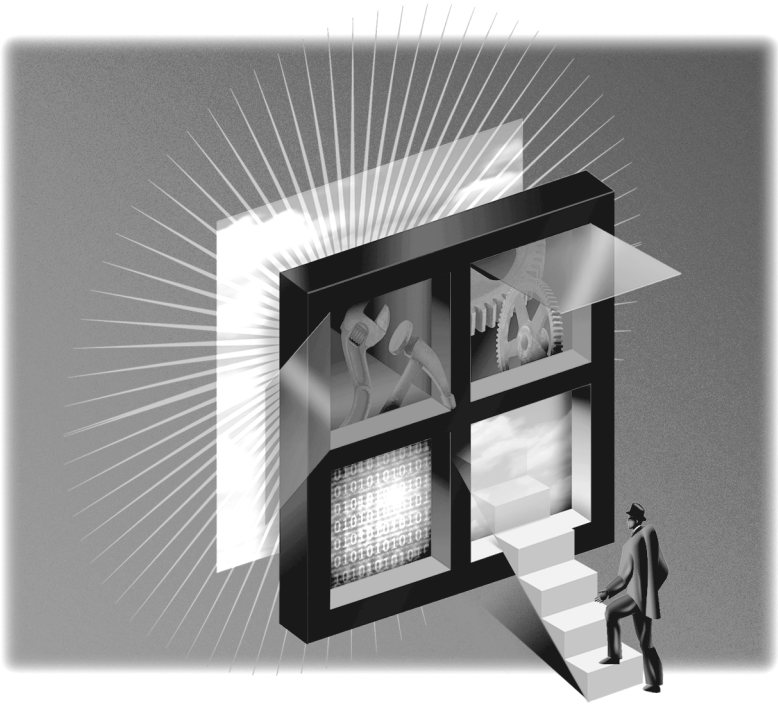
«BHV — Санкт-Петербург»

Дюссельдорф Киев
Москва Санкт-Петербург

Содержание

Часть IV

Ядро и принтер



Глава 13 Управление памятью и файловый ввод/вывод

13

Если вы впервые учитесь программированию под Windows и используете для этого Windows 95, то вы — счастливый человек. Вы даже не представляете себе, как вам повезло. В самом деле, основной урок, который вы извлечете из этой главы, можно сформулировать кратко следующим образом: при работе с памятью или файлами вам редко (а, может быть, и никогда) придется использовать что-либо кроме функций из стандартной библиотеки времени выполнения языка C.

Причина, по которой рекомендуется использовать библиотечные функции C (такие как *malloc, free, fopen, fclose* и т. д.), состоит в том, что они просты и понятны, и, кроме того, вероятно, вам хорошо знакомы. Но самое главное заключается в том, что у вас не возникнет никаких проблем при использовании этих функций в программах, написанных для Windows 95. Как будет показано ниже, так было далеко не всегда.

Управление памятью и файловый ввод/вывод являются очень старыми услугами, которые предоставляет программам операционная система (например, старая добрая неграфическая MS DOS). Третьей услугой является подсистема выполнения, которая в MS DOS поддерживала простейшую загрузку из файла в память и запуск на выполнение одной задачи. Кроме этих трех услуг, четвертой важной, может считаться отслеживание даты и времени.

Набор системных услуг, поддерживаемых ядром Windows 95, гораздо более широк. Он включает в себя динамическое связывание (оно будет рассмотрено в главе 19), многозадачность, многопоточность и синхронизацию потоков (глава 14), связь между процессами (главы 16, 17 и 20), а также некоторые другие услуги, которые не включены в данную книгу.

Хотя использовать библиотечные функции языка C удобно, возможно, в принципе, написание программы для Windows 95 вообще без использования этих функций. Каждая библиотечная функция, которая требует обращения к операционной системе (такие как функции управления памятью или файлового ввода/вывода) имеет соответствующую, и, как правило, более развитую и гибкую функцию операционной системы. Какой путь выбрать — использование функций библиотеки языка C или функций операционной системы — дело ваше. Можете испробовать оба варианта и сравнить.

Управление памятью: хорошо, плохо и ужасно

Для того чтобы увидеть, насколько далеко продвинулась вперед Windows за последние десять лет, достаточно интересно и полезно ознакомиться с минимальными требованиями к компьютеру для работы Windows 1.0 выпуска ноября 1985 года: 320 Кбайт памяти, операционная система MS DOS 2.0 и выше, два дисковода, графическая видеокарта. Эти требования отражают тип компьютера, на котором в то время работало большинство пользователей. Оглядываясь назад, можно сказать, что Microsoft добилась совершенно уникального результата, заставив Windows работать в такой ограниченной среде. Управление памятью в Windows 1.0 было очень странным, часто даже ужасным, но *оно работало*, по крайней мере, большую часть времени.

Сегментированная память

Windows 1.0 была разработана для микропроцессоров Intel 8086 и Intel 8088. Это были 16-разрядные микропроцессоры, способные адресовать 1МБ памяти. В компьютерах, совместимых с IBM PC, верхние 384 КБ этой памяти резервировались для памяти видеоадаптера и системного BIOS. При этом для программ и данных оставалось ничтожно мало — всего 640 КБ памяти.

Для того чтобы адресовать 1 МБ памяти требуется 20-разрядный адрес (2^{20} = 1048576). В процессорах 8086 и 8088 этот 20-разрядный адрес формировался из двух 16-разрядных значений: компоненты сегмента и компоненты смещения внутри сегмента. Микропроцессор имел четыре сегментных регистра: кода, данных, стека и дополнительный. 20-разрядный физический адрес получался сдвигом сегмента влево на 4 разряда и добавлением к полученной величине смещения:

	Сегмент:	ssssssssssssssss0000
+		
	Смещение:	0000oooooooooooooo
=	Адрес:	aaaaaaaaaaaaaaaaaaa

Таким образом строится 20-разрядный адрес, с помощью которого можно адресовать до 1 МБ памяти (2^{20}).

Если сегментные регистры содержат константы, то программа использует только 16-разрядные смещения для доступа к коду и данным. (В соответствии с архитектурой языка С сегмент стека устанавливался равным сегменту данных, используемому для хранения статических данных.) Каждый из этих двух сегментов давал возможность адресации 64 КБ памяти. Для однозадачных операционных систем, где программам требовалось только 64 КБ для кода и 64 КБ для данных, этого было достаточно.

Вместе с тем, по мере того, как прикладные программы становились более сложными, и следовательно, большими по объему, появлялась необходимость во множестве сегментов для кода и данных. Это заставило производителей компиляторов языка С определить близкие (near) указатели, имевшие величину 16 бит, и используемые для доступа к сегментам кода и данных по умолчанию, и дальние (far) указатели, которые имели ширину 32 бита, и состоявшие из смещения и сегмента. Однако, с помощью этого 32-разрядного адреса нельзя было адресовать память непосредственно. Кроме того, нельзя было увеличить адрес на 1 без учета логики, обрабатывающей переполнение смещения и установку сегментного адреса. Производители компиляторов языка С определили различные модели программирования: маленькая (small) (один сегмент кода, один сегмент данных), средняя (medium) (много сегментов кода), компактная (compact) (много сегментов данных), большая (large) (много сегментов кода и данных), огромная (huge) (аналогично большой, но со встроенной логикой обработки увеличения адреса).

Сама MS DOS не имела достаточно большой поддержки управления памятью. Многие программы ранних версий MS DOS просто определяли, какое количество памяти имеется в их распоряжении, и использовали ее всю целиком. В те годы программисты гордились собой тем больше, чем лучше они могли использовать персональный компьютер, обходя ограничения операционной системы.

Поскольку Windows 1.0 была многозадачной средой, появилась необходимость расширить возможности управления памятью по сравнению с возможностями MS DOS. Подумайте, пожалуйста, над утверждением: в то время, как множество программ загружаются в память, позднее освобождают ее, память становится фрагментированной. Операционная система должна перемещать блоки памяти, чтобы объединить свободное пространство. Другими словами, многозадачность без управления памятью существовать не может.

Как это может быть реализовано? Вы не можете просто перемещать блоки кода и данных в памяти безотносительно прикладной программы. В этом случае она будет содержать неправильный адрес. И вот здесь сегментированная память показывает свои возможности. Если программа использует только смещения, то сегменты могут быть изменены операционной системой. Именно это и осуществлялось в ранних версиях Windows.

Одним из следствий этого было то, что программы для Windows были ограничены использованием только маленькой и средней моделями памяти с одним 64-килобайтным сегментом для данных. Программы использовали близкие указатели для ссылок на свой сегмент данных; адрес сегмента данных для конкретного процесса устанавливался операционной системой при передаче управления программе. Это позволяло Windows перемещать сегмент данных программы и переустанавливать адрес сегмента. Все дальние вызовы функций, выполняемые программой (включая вызовы функций операционной системы), выполнялись тогда, когда сегменты кода, используемые программой, были перемещены в память.

Программа, выполнявшаяся в 16-разрядной версии Windows, могла либо выделять память из собственного сегмента данных (называемого локальной памятью, адресуемой с помощью 16-разрядного указателя смещения), либо за пределами своего сегмента данных (эта область памяти называлась глобальной, и адресовалась с помощью 32-разрядных адресов). В обоих случаях функции выделения памяти возвращали описатель блока памяти. Программы должны были фиксировать блок в памяти. Функции фиксации возвращали дальний указатель. После использования памяти следовало снять фиксацию с блока памяти. Это давало Windows возможность перемещать блоки памяти при необходимости. Процесс фиксации и снятия фиксации часто выполнялся в процессе обработки одного сообщения, и часто приводил к многочисленным ошибкам и головной боли для программистов.

Как и в других случаях, описатели памяти, в действительности, были просто указателями на таблицу в ядре Windows. Ранние версии Windows содержали сегмент памяти, названный BURGERMASTER, который содержал главную таблицу описателей памяти. Этот сегмент был так назван в честь излюбленного ресторанчика разработчиков ранних версий Windows, который располагался на противоположной стороне высокоскоростной магистрали от первых офисов фирмы Microsoft в Вашингтоне. (Если посетить этот ресторанчик, то можно заметить, что разработчики Windows должны были быть очень хорошими бегунами, чтобы пересекать столь оживленное шоссе без происшествий.)

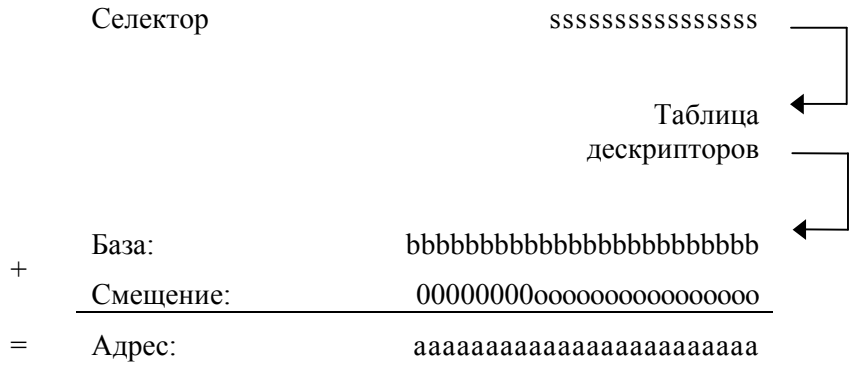
Существует все-таки одна причина, по которой важно обсуждение этой темы. Она состоит в том, что в Windows 95 содержатся некоторые части описанной выше схемы адресации Windows 1.0. Архитектура Windows 95 довольно сильно скрывает это, но иногда, это все-таки проявляется в структуре и синтаксисе вызовов функций.

Проблемы с ближними и дальними адресами в 16-разрядных версиях Windows переносились и на файловый ввод/вывод. Во многих случаях программа получала от диалогового окна имя файла, которое являлось дальним указателем. Но в маленькой и средней моделях памяти библиотечные функции файлового ввода/вывода С ожидали ближнего указателя. Аналогично, программа, которая хранила данные в блоках глобальной памяти, должна была адресовать их с помощью дальних указателей и, как следствие, файловый ввод/вывод в эти блоки был затруднен. Возникала необходимость, чтобы Windows дублировала вызовы всех функций файлового ввода/вывода, написанных для MS DOS.

Промежуточные решения

Очевидно, что обсуждение этого вопроса уже возбудило неприятные воспоминания у ветеранов программирования в MS DOS и Windows. Другим неприятным воспоминанием является, вероятно, спецификация отображаемой памяти (Expanded Memory Specification), разработанная фирмами Lotus, Intel и Microsoft (LIM EMS). Для реализации этой спецификации обычно использовалась специальная плата, содержащая память, которая могла быть адресована 16-килобайтными блоками через 64-килобайтное окно, располагавшееся в верхней зоне памяти, не занятой платой видеоадаптера и ROM BIOS. Различные блоки памяти размером 16 КБ могли быть включены/выключены в/из окно. Windows 2.0 поддерживала спецификацию LIM EMS, имеющую несколько функций.

К тому времени, как версия Windows 3.0 получила широкое распространение, у Microsoft появилась возможность поддерживать защищенный режим (protected mode) процессора Intel 286 (вместо реального режима процессоров 8086 и 8088, рассмотренного выше), причем без существенных проблем для уже существовавших программ. В защищенном режиме сегментный адрес называется селектором (selector). Он также имеет ширину 16 бит, но внутри 286 процессора он ссылается на 24-разрядный базовый адрес (base address), который складывается затем с 16-разрядным смещением, и таким образом, формируется 24-разрядный физический адрес, с помощью которого можно адресовать до 16 МБ памяти:



Чтобы обеспечить переход от более ранних версий Windows, описатель, возвращаемый функцией выделения глобальной памяти, был просто селектором. Такое использование защищенного режима отчасти облегчало управление памятью, но истинная цель еще не была достигнута.

И, наконец, 32 бита

Windows 95 требует наличия микропроцессоров Intel 386, 486 или Pentium. Эти микропроцессоры используют 32-разрядную адресацию памяти и, следовательно, могут реализовать доступ к 2^{32}, т. е. 4 294 967 296 байтам (или 4 ГБ) физической памяти. Конечно, большинству пользователей Windows еще очень далеко до этого предела. В соответствии с официальной информацией, операционная система Windows 95 требует всего 4 МБ памяти, но рекомендуется 8 МБ. На сегодняшний день 16 МБ памяти считается достаточным для обеспечения необходимого свободного пространства для большинства приложений.

Несмотря на то, что микропроцессоры Intel 386, 486 или Pentium могут использовать сегментную адресацию памяти, Windows 95 сохраняет неизменными сегментные регистры и использует 32-разрядную плоскую (flat) адресацию памяти. Это означает, что адреса в приложении Windows 95 хранятся как простые 32-разрядные величины, обеспечивая доступ к 4 ГБ памяти.

Используемые прикладными программами Windows 95 32-разрядные адреса для доступа к коду и данным, *не являются* 32-разрядными физическими адресами, которые микропроцессор использует для адресации физической памяти. Адрес, который используется приложением, называется виртуальным адресом (virtual address). Он преобразуется в физический адрес посредством таблицы страниц (page table). Этот процесс обычно прозрачен для прикладных программ. Программе кажется, что она расположена в 32-разрядном адресном пространстве, и для доступа к ней не требуется никаких особых усилий. Однако, в технической документации по Windows 95 существуют ссылки на виртуальные адреса и таблицы страниц. Поэтому, полезно будет рассмотреть механизм виртуальной памяти.

Физическая память делится на страницы (pages) размером 4096 байт (4 КБ). Следовательно, каждая страница начинается с адреса, в котором младшие 12 бит нулевые. Машина, оснащенная 8 МБ памяти, содержит 2048 страниц. Операционная система Windows 95 хранит набор таблиц страниц (каждая таблица сама представляет собой страницу) для преобразования виртуального адреса в физический.

Каждый процесс, выполняемый в Windows 95, имеет свою собственную страницу каталога (directory page) таблиц страниц, которая содержит до 1024 32-разрядных дескриптора таблиц страниц. Физический адрес страницы каталога таблиц страниц хранится в регистре CR3 микропроцессора. Содержимое этого регистра изменяется при переключении Windows 95 управления между процессами. Старшие 10 бит виртуального адреса определяют один из 1024 возможных дескрипторов в каталоге таблиц страниц. В свою очередь, старшие 20 бит дескриптора таблицы страниц определяют физический адрес таблицы страниц (младшие 12 бит физического адреса равны нулю). Каждая таблица страниц содержит, в свою очередь, до 1024 32-разрядных дескриптора страниц. Выбор одного из этих дескрипторов определяется содержимым средних 10 битов исходного виртуального адреса. Старшие 20 бит дескриптора страницы определяют физический адрес начала страницы, а младшие 12 бит виртуального адреса определяют физическое смещение в пределах этой страницы.

Очевидно, что это сложно понять с первого раза. Проиллюстрируем этот процесс еще раз в символьной форме. Вы можете представить 32-разрядный виртуальный адрес (с которым оперирует программа) в виде 10-разрядного индекса в таблице каталога таблиц страниц (d), 10-разрядного индекса в таблице страниц (p), 12-разрядного смещения (o):

dddd-dddd-ddpp-pppp-pppp-oooo-oooo-oooo

Для каждого процесса микропроцессор хранит в регистре CR3 (r) старшие 20 бит физического адреса таблицы каталога таблиц страниц:

rrrr-rrrr-rrrr-rrrr-rrrr

Начальный физический адрес каталога таблиц страниц определяется как:

rrrr-rrrr-rrrr-rrrr-rrrr-0000-0000-0000

Запомните, что каждая страница имеет размер 4 КБ и начинается с адреса, у которого 12 младших бит нулевые. Сначала микропроцессор получает физический адрес:

rrrr-rrrr-rrrr-rrrr-rrrr-dddd-dddd-dd00

По этому адресу содержится другое 20-разрядное значение (t-table):

tttt-tttt-tttt-tttt-tttt

соответствующее начальному физическому адресу таблицы страниц:

tttt-tttt-tttt-tttt-tttt-0000-0000-0000

Затем, микропроцессор осуществляет доступ по физическому адресу:

tttt-tttt-tttt-tttt-tttt-pppp-pppp-pp00

Здесь хранится 20-битная величина, являющаяся основой для физического адреса начала страницы памяти (f-page frame):

ffff-ffff-ffff-ffff-ffff

Результирующий 32-разрядный физический адрес получается в результате комбинирования основы физического адреса страницы и 12-разрядного смещения виртуального адреса:

ffff-ffff-ffff-ffff-ffff-oooo-oooo-oooo

Это и есть результирующий физический адрес.

Может показаться, что для преобразования виртуального адреса в физический требуется много времени, но на самом деле, это не так. Микропроцессоры Intel 386, 486 и Pentium имеют внутреннюю кэш-память, в которой могут храниться таблицы страниц. Фактически, преобразование адреса осуществляется очень быстро без каких-либо существенных потерь производительности. Такое двухступенчатое разделение памяти на страницы дает каждому приложению теоретическое ограничение по памяти в 1 миллион страниц по четыре килобайта каждая.

Преимущества разделения памяти на страницы огромны. Во-первых, приложения изолированы друг от друга. Никакой процесс не может случайно или преднамеренно использовать адресное пространство другого процесса, т. к. он не имеет возможности его адресовать без указания соответствующего значения регистра CR3 этого процесса, которое устанавливается только внутри ядра Windows 95.

Во-вторых, такой механизм разделения на страницы решает одну из основных проблем в многозадачной среде — объединение свободной памяти. При более простых схемах адресации в то время, как множество программ выполняются и завершаются, память может стать фрагментированной. В случае, если память сильно

фрагментирована, программы не могут выполняться из-за недостатка непрерывной памяти, даже если общего количества свободной памяти вполне достаточно. При использовании разделения на страницы нет необходимости объединять свободную физическую память, поскольку страницы необязательно должны быть расположены последовательно. Все управление памятью производится с помощью манипуляций с таблицами страниц. Потери связаны только собственно с самими таблицами страниц и с их 4 КБ размером.

В-третьих, в 32-битных дескрипторах страниц существует еще 12 бит, кроме тех, которые используются для адреса страницы. Один из этих битов показывает возможность доступа к конкретной странице (он называется битом доступа, "accessed bit"); другой показывает, была ли произведена запись в эту страницу (он называется битом мусора, "dirty bit"). Windows 95 может использовать эти биты для того чтобы определить, можно ли сохранить эту страницу в файле подкачки для освобождения памяти. Еще один бит — бит присутствия (present bit) показывает, была ли страница сброшена на диск и должна ли быть подкачена обратно в память.

Другой бит ("чтения/записи") показывает, разрешена ли запись в данную страницу памяти. Этот бит обеспечивает защиту кода от "блуждающих" указателей. Например, если включить следующий оператор в программу для Windows:

```
*(int*) WinMain = 0;
```

то на экран будет выведено следующее окно сообщения:

"This program has performed an illegal operation and will be shutdown."
("Эта программа выполнила недопустимую операцию и будет завершена").

Этот бит не препятствует компилированной и загруженной в память программе быть запущенной на выполнение.

Приведем несколько замечаний по поводу управления памятью в Windows 95:

Виртуальные адреса имеют разрядность 32 бита. Программа и данные имеют адреса в диапазоне от 0x00000000 до 0x7FFFFFFF. Сама Windows 95 использует адреса от 0x80000000 до 0xFFFFFFFF. В этой области располагаются точки входа в динамически подключаемые библиотеки Windows 95.

Общее количество свободной памяти, доступной программе, определяется как количество свободной физической памяти плюс количество свободного места на жестком диске, доступного для свопинга страниц. Как правило, при управлении виртуальной памятью Windows 95 использует алгоритм LRU (least recently used) для определения того, какие страницы будут сброшены на диск. Бит доступа и бит мусора помогают осуществить эту операцию. Страницы кода не должны сбрасываться на диск: поскольку запись в его страницы запрещена, они могут быть просто загружены из файла с расширением .EXE или из динамически подключаемой библиотеки.

Иногда вы можете заметить, что происходит обращение к диску во время перемещения мыши из рабочей области одной программы в рабочую область другой программы. Почему это происходит? Windows 95 должна посылать сообщения о передвижении мыши второму приложению. Если программа, выполняющая обработку этого сообщения, не находится в данный момент в памяти, то Windows загружает ее с диска. Если запущено несколько больших приложений одновременно, а в компьютере немного памяти, то вы можете стать свидетелем чрезмерного количества обращений к диску в моменты перехода от одной программы к другой, т. к. Windows вновь загружает с диска ранее удаленные страницы. Время от времени отдельные программы будут замедляться (или приостанавливаться на какое-то время), пока Windows осуществляет свопинг.

Страницы кода могут разделяться между приложениями. Это особенно полезно для динамически подключаемых библиотек. Несколько программ, выполняемых одновременно, могут использовать одни и те же функции Windows 95, не требуя, чтобы один и тот же код загружался в память несколько раз. Достаточно одной копии.

При динамическом выделении памяти каждый выделяемый блок необязательно получает свою собственную страницу. Последовательное выделение небольших объемов памяти производится в одной и той же физической странице с ближайшего 4-разрядного стартового адреса (т. е. для выделения одного байта используется 16 бит). Если какой-либо блок расширяется, то он может быть физически перемещен в памяти, если следующий за ним блок памяти занят.

Кроме разделения памяти на страницы по 4 КБ физическая память не может стать безнадежно фрагментированной, поскольку дефрагментация заключается только в манипуляциях с таблицами страниц. Однако виртуальная память конкретного приложения *может* стать фрагментированной, если приложение осуществляет выделение, освобождение, повторное выделение и освобождение слишком большого числа блоков памяти. Ограничения в 2 ГБ обычно достаточно для приложения и его данных. Но возможно, что программа столкнется с нехваткой физической памяти до того, как будет достигнут предел виртуальной памяти. Если вы считаете, что это может случиться с вашей программой, то вам стоит подумать об использовании перемещаемой (moveable) памяти. Об этом будет рассказано ниже в этой главе.

И в заключение, после всех предварительных замечаний, наш совет остается тем же: используйте библиотечные функции C везде, где это возможно. В вашем распоряжении имеется 32-разрядное адресное пространство.

Выделение памяти

Достаточно ли убедил вас совет использовать библиотечные функции C для выделения памяти? Для того чтобы подкрепить это утверждение, начнем с краткого обзора.

Библиотечные функции C

Вы можете определить в программе указатель (например, на массив целых чисел) следующим образом:

```
int *p;
```

Указатель *p* — 32-разрядное число, которое неинициализировано. Вы можете выделить блок памяти, на который будет указывать *p*, следующим образом:

```
p =(int *) malloc(1024);
```

При этом выделяется блок памяти размером 1024 байта, который может хранить 256 32-разрядных целых. Указатель, равный NULL, показывает, что выделение памяти не было успешным. Можно также выделить такой блок памяти, используя следующий вызов:

```
p =(int *) calloc(256, sizeof(int));
```

Два параметра функции *calloc* перемножаются и в результате получается 1024 байта. Кроме того, функция *calloc* производит обнуление блока памяти.

Если вам необходимо увеличить размер блока памяти (например, удвоить его), то вы можете вызвать функцию:

```
p =(int *) realloc(p, 2048);
```

Указатель является параметром функции, и указатель (возможно, отличающийся по значению от первого, особенно, если блок увеличивается) является возвращаемым значением функции. Этот пример показывает, что операционная система (в данном случае Windows 95) может перемещать блок в рамках виртуальной памяти. Например, если вы выделили блок размером 1024 байта, то его виртуальный адрес может быть равен 0x00750100. Вы можете выделить второй блок памяти и получить виртуальный адрес 0x00750504. Расширив первый блок памяти до 2048 байт, использовать тот же виртуальный адрес невозможно. В этом случае Windows 95 должна переместить блок в физической памяти на новую страницу.

При окончании работы с памятью, вызовите функцию:

```
free(p);
```

Только указанные четыре функции определены в стандарте ANSI языка C. Часто производители компиляторов реализуют несколько большее количество функций, наиболее распространенной из которых является функция *_msize*, возвращающая размер выделенного блока.

Фундаментальное выделение памяти в Windows 95

Как уже говорилось ранее, все, что вы можете делать с помощью библиотечных функций C, вы можете делать самостоятельно, или используя вызовы функций ядра Windows 95. Ниже приведена функция Windows 95 для выделения блока памяти для указателя на целые:

```
p =(int *) GlobalAlloc(uiFlags, dwSize);
```

Функция имеет два параметра: набор флагов и размер выделяемого блока в байтах. Она возвращает виртуальный адрес, который может использоваться в программе для доступа к выделенной памяти. Значение NULL говорит о том, что имеющейся в распоряжении памяти для выделения недостаточно.

За исключением одной, для каждой функции, начинающейся со слова *Global*, существует другая, начинающаяся со слова *Local*. Эти два набора функций в Windows 95 идентичны. Два различных слова сохранены для совместимости с предыдущими версиями Windows, где функции *Global* возвращали дальние указатели, а функции *Local* — ближние.

Хотя определения параметров немного отличаются, они оба являются 32-разрядными беззнаковыми целыми. Если первый параметр задать нулевым, то это эквивалентно использованию флага

```
GMEM_FIXED(равен нулю).
```

Такой вызов функции *GlobalAlloc* эквивалентен вызову функции *malloc*. В ранних версиях Windows присутствие флага GMEM_FIXED приводило к проблемам управления памятью, поскольку Windows не могла перемещать такие блоки в физической памяти. В Windows 95 флаг GMEM_FIXED вполне допустим, поскольку функция возвращает виртуальный адрес, и операционная система может перемещать блок в физической памяти, внося изменения в таблицу страниц.

Вы можете также использовать флаг:

```
GMEM_ZEROINIT
```

для обнуления всех байтов выделяемого блока памяти. Флаг GPTR включает в себя флаги GMEM_FIXED и GMEM_ZEROINIT, как определено в заголовочных файлах Windows:

```
#define GPTR(GMEM_FIXED | GMEM_ZEROINIT)
```

Имеется также функция изменения размера блока памяти:

```
p =(int *) GlobalReAlloc(p, dwSize, uiFlags);
```

Вы можете использовать флаг GMEM_ZEROINIT для обнуления добавляющихся в блок памяти байтов, если блок расширяется.

Существует функция, возвращающая размер блока памяти:

```
dwSize = GlobalSize(p);
```

и функция освобождения памяти:

```
GlobalFree(p);
```

Перемещаемая память

Как уже отмечалось ранее, в предыдущих версиях Windows наличие флага GMEM_FIXED приводило к проблемам при управлении памятью, поскольку Windows должна была сохранять блок памяти в фиксированном месте физической памяти. В Windows 95 блок памяти может быть перемещен в физической памяти при сохранении виртуального адреса неизменным. Функция *GlobalAlloc*, тем не менее, поддерживает флаг

```
GMEM_MOVEABLE
```

и комбинированный флаг для дополнительного обнуления блока памяти (как описано в заголовочных файлах Windows):

```
#define GHND.GMEM_MOVEABLE | GMEM_ZEROINIT)
```

Флаг GMEM_MOVEABLE позволяет перемещать блок памяти в *виртуальной* памяти. Это необязательно означает, что блок памяти будет перемещен в физической памяти, но адрес, которым пользуется программа для чтения и записи, может измениться. Это звучит странно? Возможно. Но вскоре мы увидим, как работает этот механизм.

Вы можете задать вопрос: почему я могу захотеть использовать перемещаемую память, если я не обязан это делать? (И вы можете поставить его более четко после того, как увидите, что для этого требуется.) Ответ состоит в том, чтобы сохранить совместимость с существующим исходным кодом программ для Windows. Более удачный ответ — вы можете беспокоиться о фрагментации виртуальной памяти. Если ваша программа выделяет, расширяет, освобождает память, как сумасшедшая, то виртуальная память может сделаться фрагментированной. Может ли ваша программа дойти до предела в 2 ГБ виртуальной памяти до того, как будет достигнут предел 4, 8 или 16 МБ физической памяти? Это *может* произойти. Проблема встает острее при непрерывном использовании машин. (Как известно, после окончания рабочего дня запускаются программы-хранители экрана.) Программы, выполнение которых происходит в течение нескольких дней, в конце концов могут столкнуться с большей фрагментацией памяти, чем программы, разработанные для выполнения в течение одного-двух часов.

Поскольку это потенциальная проблема, то вы можете захотеть использовать перемещаемую память. Теперь рассмотрим, как это делается. Первым делом определим указатель и переменную типа GLOBALHANDLE:

```
int *p;
GLOBALHANDLE hGlobal;
```

Затем, выделим память, например так:

```
hGlobal = GlobalAlloc(GHND, 1024);
```

Обратите внимание, что не требуется никакого преобразования типа для возвращаемого функцией *GlobalAlloc* значения. Функция определена, как возвращающая значение типа GLOBALHANDLE, поскольку именно так и было в предыдущих версиях Windows.

Как и при использовании любого другого описателя Windows вам не нужно беспокоиться о его численном значении. Когда вам необходимо обратиться к памяти, для фиксации блока используйте вызов:

```
p =(int *) GlobalLock(hGlobal);
```

Эта функция преобразует описатель памяти в указатель. Пока блок зафиксирован, Windows не изменяет его виртуальный адрес. Когда вы заканчиваете работу с блоком, для снятия фиксации вызовите функцию:

```
GlobalUnlock(hGlobal);
```

Этот вызов дает Windows свободу перемещать блок в виртуальной памяти. Для того чтобы правильно осуществлять этот процесс (и чтобы испытать муки программистов более ранних версий Windows), вы должны фиксировать и снимать фиксацию блока памяти в ходе обработки одного сообщения.

Когда вы хотите освободить память, вызовите функцию *GlobalFree* с параметром-описателем, а не указателем. Если в данный момент вы не имеете доступа к описателю, то вам необходимо использовать функцию:

```
hGlobal = GlobalHandle(p);
```

Вы можете фиксировать блок памяти несколько раз до того, как снять с него фиксацию. Windows запоминает количество фиксаций, и каждое фиксирование требует снятия для того чтобы дать возможность блоку перемещаться. Перемещение блока в виртуальной памяти не есть перемещение байтов с одного места в другое — производятся только манипуляции с таблицами страниц. Единственной причиной для выделения перемещаемой памяти служит предотвращение фрагментации виртуальной памяти.

Удаляемая память

Если вы уже набрались смелости, чтобы использовать опцию GMEM_MOVEABLE, то, может быть, у вас хватит смелости попробовать использовать опцию:

```
GMEM_DISCARDABLE
```

Эта опция может быть использована только совместно с GMEM_MOVEABLE. Блок памяти, выделенный с этим флагом, может быть удален из физической памяти ядром Windows, когда необходима свободная память.

Может быть, это звучит кощунственно, но немного подумайте об этом. Например, блоки памяти, содержащие код, являются удаляемыми. Они являются защищенными от записи. Следовательно, быстрее загрузить код из исходного файла .EXE, чем записывать его на диск, а затем вновь загружать с диска. Если вы выделяете память для неизменяемых данных, которые могут быть легко регенерированы (обычно загрузкой из файла), то можно сделать этот блок удаляемым. О том, что данные были сброшены, вы узнаете, когда вызовите функцию *GlobalLock* и получите в ответ NULL. Теперь, вы восстанавливаете данные.

Блок памяти не может быть удален до тех пор, пока счетчик фиксаций больше нуля. Для преднамеренного удаления блока памяти, вызовите:

```
GlobalDiscard(hGlobal);
```

Другие функции и флаги

Другим доступным для использования в функции *GlobalAlloc* является флаг GMEM_SHARE или GMEM_DDESHARE (они идентичны). Как следует из его имени, этот флаг предназначен для динамического обмена данными, который подробно рассматривается в главе 17.

Функции *GlobalAlloc* и *GlobalReAlloc* могут также включать флаги GMEM_NODISCARD и GMEM_NOCOMPACT. Эти флаги дают указание Windows не удалять и не перемещать блоки памяти для удовлетворения запросов памяти. Только излишне альтруистичные программисты используют эти флаги.

Функция *GlobalReAlloc* может также изменять флаги (например, преобразовывать фиксированный блок памяти в перемещаемый, и наоборот), если новый размер блока задан равным нулю, и указан флаг GMEM_MODIFY в параметре флагов.

Функция *GlobalFlags* возвращает комбинацию флагов GMEM_DISCARDABLE, GMEM_DISCARDED и GMEM_SHARE.

Наконец, вы можете вызвать функцию *GlobalMemoryStatus* (для этой функции нет функции-двойника со словом *Local*) с указателем на структуру типа MEMORYSTATUS для определения количества физической и виртуальной памяти, доступной приложению.

На этом заканчивается обзор функций, начинающихся со слова *Global*. Windows 95 также поддерживает некоторые функции, которые вы реализуете сами или дублируете библиотечными функциями C. Это функции *FreeMemory* (заполнение конкретным байтом), *ZeroMemory* (обнуление памяти), *CopyMemory* и *MoveMemory* — обе копируют данные из одной области памяти в другую. Если эти области перекрываются, то функция *CopyMemory* может работать некорректно. Вместо нее используйте функцию *MoveMemory*.

Хорошо ли это?

Перед тем как осуществить доступ к памяти, вам, может быть, захочется проверить, возможен доступ или нет. Если указатель является недействительным, исключается общая защита программы. Предварительная проверка указателя гарантирует, что этого не произойдет. Функции *IsBadCodePtr, IsBadReadPtr, IsBadWritePtr и IsBadStringPtr* выполняют эту проверку. Первая из этих функций просто принимает указатель в качестве параметра

и возвращает ненулевое значение (TRUE), если указатель действителен. Другие три функции получают указатель в качестве первого параметра и длину блока памяти в качестве второго параметра. Четвертая функция, кроме того, осуществляет проверку до тех пор, пока не встретит нулевой ограничитель строки.

Функции управления виртуальной памятью

Windows 95 поддерживает ряд функций, начинающихся со слова *Virtual*. Эти функции предоставляют значительно больше возможностей управления памятью. Однако, только очень необычные приложения требуют использования этих функций.

Например, вы разрабатываете интегрированную среду разработчика, которая включает в себя компилятор и исполнительную систему. Программа читает исходный код и компилирует его, а результат компиляции заносится в память. Затем, вы хотите пометить этот блок памяти как "только для выполнения", т. е. так, чтобы было невозможно случайно или преднамеренно прочитать (или того хуже, записать) в него что-либо из программы, которая была только что скомпилирована и готова к выполнению. Это одно из действий, которое должна осуществлять развитая среда разработчика для обработки ошибок при работе с указателями в пользовательской программе. Вы можете также пометить часть блоков памяти "только для чтения". Оба указанных действия выполняются только с помощью функций управления виртуальной памятью. Но, повторяем, эта операция достаточно экзотическая.

Может быть, более часто возникает необходимость зарезервировать большой блок виртуальной памяти для данных, который может быть сильно увеличен в процессе выполнения программы. Действуя обычным путем — т. е. часто используя функции *realloc* или функцию Windows *GlobalReAlloc* для динамического изменения размера выделенного блока памяти — можно резко снизить производительность программы. Функции управления виртуальной памятью могут помочь избежать этого. Рассмотрим теперь, как это делается.

В Windows 95 любой блок виртуальной памяти может находиться в одном из трех состояний: "committed" (т. е. блок спроецирован в физическую память), "free" (свободен, т. е. доступен для будущего выделения), "reserved" (зарезервирован, это нечто среднее между двумя предыдущими состояниями). Зарезервированный блок виртуальной памяти не отображается в физическую память. Адреса в пределах этого блока будут недоступны до тех пор, пока всему блоку или его какой-либо части не будет передан блок физической памяти. Таким образом, вы можете зарезервировать достаточно большой блок виртуальной памяти, не передавая ему физической памяти. Когда будет необходимо обратиться по какому-либо виртуальному адресу в пределах этого блока, вы передаете по этому адресу ровно столько физической памяти, сколько необходимо, т. е. в зарезервированном блоке виртуальной памяти могут быть участки, как связанные, так и несвязанные с блоками физической памяти. Спроецировав физическую память на нужный участок зарезервированной области виртуальной памяти, программа может обращаться к нему, не вызывая при этом исключения нарушения доступа.

Для того чтобы использовать функции работы с виртуальной памятью, вашей программе необходимо знать размер страницы памяти. В отличие от Windows NT, Windows 95 работает только на микропроцессорах фирмы Intel, и размер страницы всегда равен 4096 байт. Если ваша программа разрабатывается также для запуска под Windows NT, используйте функцию *GetSystemInfo* для получения размера страницы. Эта функция имеет один параметр, который является указателем на структуру типа SYSTEM_INFO. Поле *dwPageSize* этой структуры содержит размер страницы. Используются также поля *lpMinimumApplicationAddress* и *lpMaximumApplicationAddress*, содержащие минимальный и максимальный адреса, имеющиеся в распоряжении приложения. Для Windows 95 эти значения равны соответственно 0x00400000 и 0x7FFFFFFF.

Функция *VirtualAlloc* выглядит следующим образом:

```
p = VirtualAlloc(pAddr, dwSize, iAllocType, iProtect);
```

Первый параметр показывает желаемый стартовый базовый адрес виртуальной памяти, и вы можете установить его значение в NULL при первом вызове этой функции. Второй параметр задает размер. Третий параметр может быть равен MEM_RESERVE или MEM_COMMIT для резервирования блока виртуальной памяти или для резервирования и передачи ему физической памяти. Четвертый параметр может быть константой, начинающейся с префикса PAGE_ (например, PAGE_READONLY или PAGE_EXECUTE) для задания защиты блока памяти. Последовательные вызовы функции *VirtualAlloc* могут передавать или резервировать секции этого блока. Функция *VirtualFree* используется для освобождения виртуальной памяти.

Функции работы с "кучей"

Последняя группа функций работы с памятью — это функции, имена которых начинаются со слова *Heap* (куча). Эти функции создают и поддерживают непрерывный блок виртуальной памяти, из которого вы можете выделять память более мелкими блоками. Вы начинаете с вызова функции *HeapCreate*. Затем, используете функции *HeapAllocate*, *HeapReAllocate* и *HeapFree* для выделения и освобождения блоков памяти в рамках "кучи". "Куча" может быть уплотнена для объединения свободного пространства.

Существует немного причин для использования функций, работающих с "кучей".

Файловый ввод/вывод

Повторим еще раз: для файлового ввода/вывода используйте библиотечные функции языка С везде, где это возможно. Это уже было сделано в программе POPPAD3 в главе 11. Там использовались функции *fopen, fseek, fread, fwrite* и *fclose*.

Старый путь

Работа с файлами под Windows постепенно совершенствовалась год от года. В те времена, когда использовались Windows 1.0 и Windows 2.0, единственной функцией файлового ввода/вывода была функция *OpenFile*, и официально рекомендовавшимся подходом к чтению и записи файлов была запись маленьких файлов на ассемблере, которые непосредственно осуществляли доступ к функциям MS DOS. Хотя использование стандартных функций библиотеки времени выполнения языка С было возможно, в маленькой и средней моделях памяти эти функции работали только с ближними указателями. Это было неудобно в программах, хранивших файлы данных в блоках глобальной памяти. Даже имена файлов, которые часто получали из диалоговых окон, были доступны с помощью дальних указателей.

К счастью, многие программисты вскоре обнаружили несколько недокументированных функций для работы с файлами с использованием дальних указателей. Они имели имена *_lopen, _lread, _lwrite* и т. д., и содержали непосредственные вызовы функций MS DOS. Начиная с Windows 3.0, эти функции были документированы и приняты как стандартные функции работы с файлами при программировании под Windows. Но применять их при программировании для Windows 95 не рекомендуется.

Отличия Windows 95

Windows 95 реализует несколько усовершенствований файлового ввода/вывода по сравнению с более ранними версиями Windows.

Первое, Windows 95 так же как и Windows 3.1 поддерживает библиотеку диалоговых окон общего пользования (common dialog box library), которая содержит диалоговые окна FileOpen и FileSave. Использование этих диалоговых окон было показано в главе 11. Рекомендуется при программировании использовать именно эти диалоговые окна. При их использовании исчезает необходимость разбора имени файла, который может быть системно-зависимым.

Второе, Windows 95 является 32-разрядной системой. Это значит, что вы можете читать и записывать файл большими блоками информации за один прием, используя однократный вызов функций *fread* и *fwrite* (или их эквивалентами, поддерживаемыми Windows 95). Изменения по отношению к существующему коду состоит в том, что отпадает необходимость в использовании циклов при работе с файлами большого размера.

Третье, Windows 95 поддерживает длинные имена файлов. Самое лучшее, что могут делать ваши программы с длинными именами, это просто ничего с ними не делать. (Хорошо звучит, не правда ли?) В документации по Windows сказано, что вы можете использовать данные, возвращаемые функцией *GetVolumeInformation*, для динамического выделения буферов для хранения имен файлов. Но, обычно в этом нет необходимости. Вам рекомендуется использовать две константы, определенные в файле STDLIB.H: _MAX_PATH (равно 260) и _MAX_FNAME (256) для статического выделения памяти.

Функции файлового ввода/вывода, поддерживаемые Windows 95

Если вы не пользуетесь функциями файлового ввода/вывода стандартной библиотеки времени выполнения языка С, то вы можете использовать функции, поддерживаемые Windows 95. Функция *CreateFile* является достаточно мощной:

```
hFile = CreateFile(szName, dwAccess, dwShare, NULL, dwCreate, dwFlags, 0);
```

Несмотря на имя, эта функция используется также для открытия существующего файла. Кроме того, эта функция используется для открытия каналов, используемых при обмене между процессами, а также коммуникационных устройств.

Для файлов — первый параметр является именем файла. Второй имеет значение либо GENERIC_READ, либо GENERIC_WRITE, либо GENERIC_READ | GENERIC_WRITE. Использование нулевого значения позволяет получить информацию о файле без доступа к его содержимому. Параметр *dwShare* открывает файл с общими атрибутами, позволяя другим процессам читать из него (FILE_SHARE_READ), или записывать в него (FILE_SHARE_WRITE), или и то и другое вместе.

Флаг *dwCreate* — это одна из нескольких констант, показывающая, каким образом файл должен быть открыт. Их имена сжаты и прекрасно поясняют суть. Флаг CREATE_NEW вызывает ошибку, если файл уже существует, в то время как флаг CREATE_ALWAYS приводит к удалению содержимого существующего файла. Аналогичным образом, флаг OPEN_EXISTING вызывает ошибку, если файл не существует, а флаг OPEN_ALWAYS создает файл, если он не существует. Флаг TRUNCATE_EXISTING приводит к ошибке, если файл не существует, и удаляет все содержимое, если файл существует.

Параметр *dwFlags* может быть комбинацией констант, начинающихся со слов FILE_ATTRIBUTE и FILE_FLAG, для установки атрибутов файла и других особенностей.

Функция *CreateFile* возвращает переменную типа HANDLE. При завершении работы с файлом его необходимо закрыть, используя функцию *CloseHandle* с описателем файла в качестве параметра. Функции *ReadFile* и *WriteFile* похожи:

```
ReadFile(hFile, pBuffer, dwToRead, &dwHaveRead, NULL);
WriteFile(hFile, pBuffer, dwToWrite, &dwHaveWritten, NULL);
```

Второй параметр — это указатель на буфер, содержащий данные; третий параметр содержит количество байтов для чтения или записи; четвертый параметр — указатель на переменную, в которую при возврате из функции будет занесено количество байтов, которые были реально считаны или записаны. (Последний параметр используется только для файла, открываемого с флагом FILE_FLAG_OVERLAPPED, но этот случай не входит в предмет рассмотрения данной книги.)

Ввод/вывод с использованием файлов, проецируемых в память

При работе в Windows 95 (и это является одним из усовершенствований системы по сравнению с более ранними 16-разрядными версиями Windows) существует возможность читать и записывать данные в файл так, как будто это блок памяти. На первый взгляд это может показаться несколько странным, но со временем становится понятно, что это очень удобный механизм. Это прием рекомендуется использовать также при разделении памяти между двумя и более процессами. Пример такого использования файлов, проецируемых в память, приведен в главе 19 "Динамически подключаемые библиотеки".

Вот простейший подход к вводу/выводу с использованием файлов, проецируемых в память (memory mapped files): Сначала создается обычный файл с использованием функции *CreateFile*. Затем вызывается функция:

```
hMap = CreateFileMapping(hFile, NULL, dwProtect, 0, 0, szName);
```

Параметр *dwProtect* может принимать одно из следующих значений, и должен быть совместим с режимом разделения файла: PAGE_READONLY, PAGE_WRITECOPY, PAGE_READWRITE. Последний параметр функции — необязательное имя, обычно используемое для разделения данных между процессами. В этом случае, функция *OpenFileMapping* открывает тот же файл с указанным именем. Обе функции возвращают значение типа HANDLE.

Если вам необходимо осуществить доступ к части файла, вызовите функцию *MapViewOfFile*:

```
p = MapViewOfFile(hMap, dwAccess, dwHigh, dwLow, dwNumber);
```

Весь файл или его часть могут быть спроецированы в память, начиная с заданного 64-разрядного смещения, которое задается параметрами *dwHigh* и *dwLow*. (Очевидно, что *dwHigh* будет иметь нулевое значение, если файл имеет размер менее 4 ГБ.) Параметр *dwNumber* задает количество байтов, которое вы хотите спроецировать в память. Параметр *dwAccess* может быть равен FILE_MAP_WRITE (данные можно записывать и считывать) или FILE_MAP_READ (данные можно только считывать), и должен соответствовать параметру *dwProtect* функции *CreateFileMapping*.

После этого вы можете использовать указатель, возвращаемый функцией, для доступа или модификации данных в файле. Функция *FlushViewOfFile* записывает на диск все измененные страницы файла, спроецированного в память. Функция *UnmapViewOfFile* делает недействительным указатель, возвращаемый функцией *MapViewOfFile*. Затем необходимо закрыть файл, используя функцию *CloseHandle*.

Мы рассмотрим пример этого процесса в главе, посвященной динамически подключаемым библиотекам.

Глава 14 Многозадачность и многопоточность

14

Многозадачность (multitasking) — это способность операционной системы выполнять несколько программ одновременно. В основе этого принципа лежит использование операционной системой аппаратного таймера для выделения отрезков времени (time slices) для каждого из одновременно выполняемых процессов. Если эти отрезки времени достаточно малы, и машина не перегружена слишком большим числом программ, то пользователю кажется, что все эти программы выполняются параллельно.

Идея многозадачности не нова. Многозадачность реализуется на больших компьютерах типа мэйнфрэйм (mainframe), к которым подключены десятки, а иногда и сотни, терминалов. У каждого пользователя, сидящего за экраном такого терминала, создается впечатление, что он имеет эксклюзивный доступ ко всей машине. Кроме того, операционные системы мэйнфрэймов часто дают возможность пользователям перевести задачу в фоновый режим, где они выполняются в то время, как пользователь может работать с другой программой.

Для того, чтобы многозадачность стала реальностью на персональных компьютерах, потребовалось достаточно много времени. Но, кажется, сейчас мы приближаемся к эпохе использования многозадачности на ПК (PC). Как мы увидим вскоре, некоторые расширенные 16-разрядные версии Windows поддерживают многозадачность, а имеющиеся теперь в нашем распоряжении Windows NT и Windows 95 — 32-разрядные версии Windows, поддерживают кроме многозадачности еще и многопоточность (multithreading).

Многопоточность — это возможность программы самой быть многозадачной. Программа может быть разделена на отдельные потоки выполнения (threads), которые, как кажется, выполняются параллельно. На первый взгляд эта концепция может показаться едва ли полезной, но оказывается, что программы могут использовать многопоточность для выполнения протяженных во времени операций в фоновом режиме, не вынуждая пользователя надолго отрываться от машины.

Режимы многозадачности

На заре использования персональных компьютеров некоторые отстаивали идею многозадачности для будущего, но многие ломали головы над вопросом: какая польза от многозадачности на однопользовательской машине? В действительности оказалось, что многозадачность — это именно то, что необходимо пользователям, даже не подозревавшим об этом.

Многозадачность в DOS

Микропроцессор Intel 8088, использовавшийся в первых ПК, не был специально разработан для реализации многозадачности. Частично проблема (как было показано в предыдущей главе) заключалась в недостатках управления памятью. В то время, как множество программ начинает и заканчивает свое выполнение, многозадачная операционная система должна осуществлять перемещение блоков памяти для объединения свободного пространства. На процессоре 8088 это было невозможно реализовать в стиле, прозрачном для приложений.

Сама DOS не могла здесь чем-либо существенно помочь. Будучи разработанной таким образом, чтобы быть маленькой и не мешать приложениям, DOS поддерживала, кроме загрузки программ и обеспечения им доступа к файловой системе, еще не так много средств.

Тем не менее, творческие программисты, работавшие с DOS на заре ее появления, нашли путь преодоления этих препятствий, преимущественно при использовании резидентных (terminate-and-stay-resident, TSR) программ. Некоторые TSR-программы, такие как спулер печати, использовали прерывание аппаратного таймера для выполнения процесса в фоновом режиме. Другие, подобно всплывающим (popup) утилитам, таким как SideKick, могли выполнять одну из задач переключения — приостановку выполнения приложения на время работы утилиты. DOS также была усовершенствована для обеспечения поддержки резидентных программ.

Некоторые производители программного обеспечения пытались создать многозадачные оболочки или оболочки, использующие переключение между задачами, как надстройки над DOS (например, Quarterdeck's DeskView), но только одна из этих оболочек получила широкое распространение на рынке. Это, конечно, Windows.

Невытесняющая многозадачность

Когда Microsoft выпустила на рынок Windows 1.0 в 1985 году, это было еще в большой степени искусственным решением, придуманным для преодоления ограничений MS DOS. В то время Windows работала в реальном режиме (real mode), но даже тогда она была способна перемещать блоки физической памяти (одно из необходимых условий многозадачности) и делала это, хотя и не очень прозрачно для приложений, но все-таки вполне удовлетворительно.

В графической оконной среде многозадачность приобретает гораздо больший смысл, чем в однопользовательской операционной системе, использующей командную строку. Например, в классической операционной системе UNIX, работающей с командной строкой, существует возможность запускать программы из командной строки так, чтобы они выполнялись в фоновом режиме. Однако, любой вывод на экран из программы должен быть переадресован в файл, иначе этот вывод смешается с текущим содержимым экрана.

Оконная оболочка позволяет нескольким программам выполняться совместно, разделяя один экран. Переключение вперед и назад становится тривиальным, существует возможность быстро передавать данные из одной программы в другую, например, разместить картинку, созданную в программе рисования, в текстовом файле, образованном с помощью текстового процессора. Передача данных поддерживалась в различных версиях Windows: сначала с использованием папки обмена (clipboard), позднее — посредством механизма динамического обмена данными (Dynamic Data Exchange, DDE), сейчас — через внедрение и связывание объектов (Object Linking and Embedding, OLE).

И все же, реализованная в ранних версиях Windows многозадачность не была традиционной вытесняющей, основанной на выделении отрезков времени, как в многопользовательских операционных системах. Такие операционные системы используют системный таймер для периодического прерывания выполнения одной задачи и запуска другой. 16-разрядные версии Windows поддерживали так называемую невытесняющую многозадачность (non-preemptive multitasking). Такой тип многозадачности был возможен благодаря основанной на сообщениях архитектуре Windows. В общем случае, Windows-программа находилась в памяти и не выполнялась до тех пор, пока не получала сообщение. Эти сообщения часто являлись прямым или косвенным результатом ввода информации пользователем с клавиатуры или мыши. После обработки сообщения программа возвращала управление обратно Windows.

16-разрядные версии Windows не имели возможности произвольно переключать управление с одной Windows-программы на другую, основываясь на квантах времени таймера. Переключение между задачами происходило в момент, когда программа завершала обработку сообщения и возвращала управление Windows. Такую невытесняющую многозадачность называют также кооперативной многозадачностью (cooperative multitasking) потому, что она требует некоторого согласования между приложениями. Одна Windows-программа могла парализовать работу всей системы, если ей требовалось много времени для обработки сообщения.

Хотя невытесняющая многозадачность была основным типом многозадачности в 16-разрядных версиях Windows, некоторые элементы вытесняющей (примитивной, preemptive) многозадачности в них тоже присутствовали. Windows использовала вытесняющую многозадачность для выполнения DOS-программ, а также позволяла библиотекам динамической компоновки (DLL) получать прерывания аппаратного таймера для задач мультимедиа.

16-разрядные версии Windows имели некоторые особенности, которые помогали программистам если не разрешить, то, по крайней мере, справиться с ограничениями, связанными с невытесняющей многозадачностью. Наиболее известной является отображение курсора мыши в виде песочных часов. Конечно, это не решение проблемы, а только лишь возможность дать знать пользователю, что программа занята выполнением протяженной во времени работы, и что система какое-то время будет недоступна. Другим частичным решением является использование системного таймера Windows, что позволяет выполнять какие-либо действия периодически. Таймер часто используется в приложениях типа часов и приложениях, работающих с анимацией.

Другим решением по преодолению ограничений невытесняющей многозадачности является вызов функции *PeekMessage*, как мы видели в программе RANDRECT в главе 4. Обычно программа использует вызов функции *GetMessage* для извлечения сообщений из очереди. Однако, если в данный момент времени очередь сообщений пуста, то функция *GetMessage* будет ждать поступления сообщения в очередь, а затем возвратит его. Функция *PeekMessage* работает иначе — она возвращает управление программе даже в том случае, если нет сообщений в очереди. Таким образом, выполнение работы, требующей больших затрат времени, будет продолжаться до того момента, пока в очереди не появятся сообщения для данной или любой другой программы.

Presentation Manager и последовательная очередь сообщений

Первой попыткой фирмы Microsoft (в сотрудничестве с IBM) внедрить многозадачность в квази-DOS/Windows оболочку была система OS/2 и Presentation Manager (PM). Хотя OS/2, конечно, поддерживала вытесняющую многозадачность, часто казалось, что это вытеснение не было перенесено в PM. Дело в том, что PM выстраивал в очередь сообщения, формируемые в результате пользовательского ввода от клавиатуры или мыши. Это означает, что PM не предоставляет программе такое пользовательское сообщение до тех пор, пока предыдущее сообщение, введенное пользователем, не будет полностью обработано.

Хотя сообщения от клавиатуры или мыши — это только часть множества сообщений, которые может получить программа в PM или Windows, большинство других сообщений являются результатом событий, связанных с клавиатурой или мышью. Например, сообщение от меню команд является результатом выбора пункта меню с помощью клавиатуры или мыши. Сообщение от клавиатуры или мыши не будет обработано до тех пор, пока не будет полностью обработано сообщение от меню.

Основная причина организации последовательной очереди сообщений состоит в том, чтобы отследить все действия пользователя. Если какое-либо сообщение от клавиатуры или мыши вызывает переход фокуса ввода от одного окна к другому, то следующее сообщение клавиатуры должно быть направлено в окно, на которое установился фокус ввода. Таким образом, система не знает, в какое окно передавать сообщение на обработку до тех пор, пока не будет обработано предыдущее сообщение.

В настоящее время принято соглашение о том, что не должно быть возможности для какого-либо одного приложения парализовать работу всей системы, и что требуется использовать непоследовательную очередь сообщений, поддерживаемую системами Windows 95 и Windows NT. Если одна программа занята выполнением протяженной во времени операции, то существует возможность переключить фокус ввода на другое приложение.

Решения, использующие многопоточность

Выше был рассмотрен Presentation Manager операционной системы OS/2 только из-за того, что это была первая оболочка, которая подготовила сознание некоторых ветеранов программирования под Windows (в том числе и автора) к введению многопоточности. Интересно, что *ограниченная* поддержка многопоточности в PM дала программистам основную идею организации программ, использующих многопоточность. Хотя эти ограничения сейчас преимущественно преодолены в Windows 95, тем не менее уроки, полученные при работе с более ограниченными системами, остаются актуальными и по сей день.

В многопоточной среде программы могут быть разделены на части, называемые потоками выполнения (threads), которые выполняются одновременно. Поддержка многопоточности оказывается лучшим решением проблемы последовательной очереди сообщений в PM и приобретает полный смысл при ее реализации в Windows 95.

В терминах программы "поток" — это просто функция, которая может также вызывать другие функции программы. Программа начинает выполняться со своего главного (первичного) потока, который в традиционных программах на языке C является функцией *main*, а в Windows-программах — *WinMain*. Будучи выполняемой, функция может создавать новые потоки обработки, выполняя системный вызов с указанием функции инициализации потока (initial threading function). Операционная система в вытесняющем режиме переключает управление между потоками подобно тому, как она это делает с процессами.

В PM системы OS/2 любой поток может либо создавать очередь сообщений, либо не создавать. PM-поток должен создавать очередь сообщений, если он собирается создавать окно. С другой стороны, поток может не создавать очередь сообщений, если он осуществляет только обработку данных или графический вывод. Поскольку потоки, не создающие очереди сообщений, не обрабатывают сообщения, то они не могут привести к "зависанию" системы. На поток, не имеющий очереди сообщений, накладывается только одно ограничение — он не может посылать асинхронное сообщение в окно потока, имеющего очередь сообщений, или вызывать какую-либо функцию, если это приведет к посылке сообщения. (Однако эти потоки могут *посылать* синхронные сообщения потокам, имеющим очередь сообщений.)

Таким образом, программисты, работавшие с PM, научились разбивать свои программы на один поток с очередью сообщений, создающий все окна и обрабатывающий сообщения для них, и один или несколько потоков, не имеющих очередей сообщений, и выполняющих продолжительные действия в фоновом режиме. Кроме того, программисты, работавшие с PM, узнали о "правиле 1/10 секунды". Оно состоит в том, что поток с очередью сообщений тратит не более 1/10 секунды на обработку любого сообщения. Все, что требует большего времени, следовало выделять в отдельный поток. Если все программисты придерживались этого правила, то никакая PM-программа не могла вызвать зависание системы более чем на 1/10 секунды.

Многопоточная архитектура

Как уже отмечалось выше, ограничения PM дали программистам основные идеи для понимания того, как использовать множество потоков в программе, выполняемой в графической среде. Ниже приведены наши

рекомендации по архитектуре многопоточных программ: первичный или главный (primary) поток вашей программы создает все окна и соответствующие им оконные процедуры, необходимые в программе и обрабатывает все сообщения для этих окон. Все остальные потоки — это просто фоновые задачи. Они не имеют интерактивной связи с пользователем, кроме как через первичный поток.

Один из способов добиться этого состоит в том, чтобы первичный поток обрабатывал пользовательский ввод и другие сообщения, возможно создавая при этом вторичные (secondary) потоки в процессе. Эти вторичные потоки выполняют не связанные с пользователем задачи.

Другими словами, первичный поток вашей программы является губернатором, а вторичные потоки — свитой губернатора. Губернатор поручает всю большую работу своим помощникам на то время, пока он осуществляет контакты с внешним миром. Поскольку вторичные потоки являются членами свиты, они не могут проводить свои пресс-конференции. Они скромно выполняют каждый свое задание, делают отчет губернатору и ждут новых указаний.

Потоки внутри отдельной программы являются частями одного процесса, поэтому они разделяют все ресурсы процесса, такие как память и открытые файлы. Поскольку потоки разделяют память, отведенную программе, то они разделяют и статические переменные. Однако, у каждого потока есть свой собственный стек, и значит, автоматические переменные являются уникальными для каждого потока. Каждый поток, также, имеет свое состояние процессора, которое сохраняется и восстанавливается при переключении между потоками.

Коллизии, возникающие при использовании потоков

Собственно разработка, программирование и отладка сложного многопоточного приложения являются, естественно, самыми сложными задачами, с которыми приходится сталкиваться программисту для Windows. Поскольку в системе с вытесняющей многозадачностью поток может быть прерван в любой момент для переключения на другой поток, то может случайно произойти любое нежелательное взаимодействие между двумя потоками.

Одной из основных ошибок в многопоточных программах является так называемое состояние гонки (race condition). Это случается, если программист считает, что один поток закончит выполнение своих действий, например, подготовку каких-либо данных, до того, как эти данные потребуются другому потоку. Для координации действий потоков операционным системам необходимы различные формы синхронизации. Одной из таких форм является семафор (semaphore), который позволяет программисту приостановить выполнение потока в конкретной точке программы до тех пор, пока он не получит от другого потока сигнал о том, что он может возобновить работу. Похожи на семафоры критические разделы (critical sections), которые представляют собой разделы кода, во время выполнения которого, поток не может быть прерван.

Но использование семафоров может привести к другой распространенной ошибке, связанной с потоками, которая называется тупиком (deadlock). Это случается, когда два потока блокируют выполнение друг друга, а для того, чтобы их разблокировать необходимо продолжить работу.

К счастью, 32-разрядные программы более устойчивы к определенным проблемам, включая проблемы с потоками, чем 16-разрядные программы. Например, предположим, что один поток выполняет простое действие:

```
lCount++;
```

где *lCount* — 32-разрядная глобальная переменная типа длинное целое, используемая другими потоками. В 16-разрядной программе, в которой такой оператор языка C транслируется в две инструкции машинного кода (сначала инкрементируется младшие 16 разрядов, а затем добавляется перенос в старшие 16 разрядов). Допустим, что операционная система прервала поток между этими двумя инструкциями машинного кода. Если переменная *lCount* имела значение 0x0000FFFF, то после выполнения первой инструкции машинного кода *lCount* будет иметь нулевое значение. Если в этот момент произойдет прерывание потока, то другой поток получит нулевое значение переменной *lCount*. Только после окончания этого потока значение *lCount* будет увеличено на единицу до своего истинного значения 0x00010000.

Такого рода ошибка может быть никогда не выявлена, поскольку довольно редко приводит к проблемам во время выполнения. Для 16-разрядных программ наилучший путь предотвратить такую ошибку — это поместить данное выражение в критический раздел, в рамках которого поток не может быть прерван. В 32-разрядной программе, однако, приведенное выражение является абсолютно корректным, поскольку оно компилируется в одну инструкцию машинного кода.

Преимущества Windows

Операционные системы Windows 95 и Windows NT не имеют последовательной очереди сообщений. Такое решение кажется очень хорошим: если программа выполняет длительную обработку сообщения, то курсор мыши принимает форму песочных часов при расположении над окном этой программы, и изменяется на обычную

стрелку, если он располагается над окном другой программы. Простым щелчком кнопкой мыши можно перевести другое окно на передний план.

Однако, пользователь по-прежнему не может работать с приложением, выполняющим длительную операцию, поскольку выполнение длительной операции предотвращает получение сообщений программой. А это нежелательно. Программа должна быть всегда открыта для сообщений, а это требует использования вторичных потоков.

В Windows 95 и Windows NT не существует различия между потоками, имеющими очередь сообщений, и потоками без очереди сообщений. При создании каждый поток получает свою собственную очередь сообщений. Это снижает число ограничений, существующих для потоков в PM-программе. (Однако, в большинстве случаев все еще обработка ввода и сообщений осуществляется в одном потоке, а протяженные во времени задачи передаются другим потокам, которые не создают окон.) Такая схема организации приложения, как мы увидим, почти всегда является наиболее разумной.

Еще хорошая новость: в Windows 95 и Windows NT есть функция, которая позволяет одному потоку уничтожить другой поток, принадлежащий тому же процессу. Как вы обнаружите, когда начнете писать многопоточные приложения под Windows, иногда это очень удобно. Ранние версии операционной системы OS/2 не содержали функции для уничтожения потоков.

И последняя хорошая новость (по крайней мере, по этой тематике): Windows 95 и Windows NT поддерживают так называемую локальную память потока (thread local storage, TLS). Для того чтобы понять, что это такое, вспомним о том, что статические переменные, как глобальные так и локальные по отношению к функциям, разделяются между потоками, поскольку они расположены в зоне памяти данных процесса. Автоматические переменные (которые являются всегда локальными по отношению к функции) — уникальны для каждого потока, т. к. они располагаются в стеке, а каждый поток имеет свой стек.

Иногда бывает удобно использовать для двух и более потоков одну и ту же функцию, а статические данные использовать уникальные для каждого потока. Это и есть пример использования локальной памяти потока. Существует несколько вызовов функций Windows для работы с локальной памятью потока. Фирма Microsoft ввела расширение в компилятор C, которое позволяет использовать локальную память потока более прозрачным для программиста образом.

Новая программа! Усовершенствованная программа! Многопоточная!

Теперь, когда потокам уделено немного внимания, давайте переведем тему обсуждения в правильное русло. Иногда имеет место тенденция использовать в программе каждую возможность, предлагаемую операционной системой. Хуже всего бывает, когда к вам подходит ваш руководитель и говорит: "Я слышал, что новая возможность очень хороша. Давай включим ее в нашу программу." А затем вы тратите неделю на то, чтобы понять, какую пользу может принести вашему приложению эта возможность.

Мораль такая — нет смысла использовать множество потоков в программе, которая в этом не нуждается. Если ваша программа выводит на экран курсор в виде песочных часов на достаточно долгий период времени, или, если она использует функцию *PeekMessage* для того, чтобы избежать появления курсора в виде песочных часов, то тогда идея реструктуризации программы в многопоточную, вероятно, может оказаться хорошей. В противном случае, вы только усложните себе работу и, возможно, внесете в программу новые ошибки.

Есть даже некоторые ситуации, когда появление курсора мыши в виде песочных часов, может быть совершенно подходящим. Выше уже упоминалось "правило 1/10 секунды". Так вот, загрузка большого файла в память может потребовать больше времени, чем 1/10 секунды. Значит ли это, что функции загрузки файла должны были быть реализованы с использованием разделения на потоки? Совсем необязательно. Когда пользователь дает программе команду открыть файл, то он или она обычно хочет, чтобы операционная система выполнила ее немедленно. Выделение процесса загрузки файла в отдельный поток просто приведет к усложнению программы. Не стоит это делать даже ради того, чтобы похвастаться перед друзьями, что вы пишите многопоточные приложения.

Многопоточность в Windows 95

Давайте рассмотрим несколько программ, использующих многопоточность.

И снова случайные прямоугольники

Программа RNDRCTMT (рис. 14.1) является многопоточной версией программы RANDRECT, приведенной в главе 4. Если вы помните, программа RANDRECT для вывода последовательности случайных прямоугольников использовала цикл обработки сообщений, содержащий вызов функции *PeekMessage*.

RNDRCTMT.MAK

```
#------------------------
# RNDRCTMT.MAK make file
```

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

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

rndrctmt.obj : rndrctmt.c
     $(CC) $(CFLAGSMT) rndrctmt.c
```

RNDRCTMT.C

```
/*----------------------------------------
   RNDRCTMT.C -- Displays Random Rectangles
                 (c) Charles Petzold, 1996
  ----------------------------------------*/

#include <windows.h>
#include <process.h>
#include <stdlib.h>

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

HWND hwnd;
int  cxClient, cyClient;

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

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

     hwnd = CreateWindow(szAppName, "Random Rectangles",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

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

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

     return msg.wParam;
     }

VOID Thread(PVOID pvoid)
     {
     HBRUSH hBrush;
```

```
     HDC    hdc;
     int    xLeft, xRight, yTop, yBottom, iRed, iGreen, iBlue;

     while(TRUE)
          {
          if(cxClient != 0 || cyClient != 0)
               {
               xLeft   = rand() % cxClient;
               xRight  = rand() % cxClient;
               yTop    = rand() % cyClient;
               yBottom = rand() % cyClient;
               iRed    = rand() & 255;
               iGreen  = rand() & 255;
               iBlue   = rand() & 255;

               hdc = GetDC(hwnd);
               hBrush = CreateSolidBrush(RGB(iRed, iGreen, iBlue));
               SelectObject(hdc, hBrush);

               Rectangle(hdc, min(xLeft, xRight), min(yTop, yBottom),
                              max(xLeft, xRight), max(yTop, yBottom));

               ReleaseDC(hwnd, hdc);
               DeleteObject(hBrush);
               }
          }
     }
LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     switch(iMsg)
          {
          case WM_CREATE :
               _beginthread(Thread, 0, NULL);
               return 0;

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

          case WM_DESTROY :
               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

Рис. 14.1 Программа RNDRCTMT

Обратите внимание, во-первых, на небольшое изменение в make-файле. На этапе компиляции переменная CFLAGS была заменена переменной CFLAGSMT. Это та же переменная CFLAGS, но с добавлением флага _MT, который необходимо задать компилятору для компиляции многопоточного приложения. В частности, компилятор вставляет имя файла LIBCMT.LIB вместо LIBC.LIB в файл с расширением .OBJ. Эта информация используется компоновщиком.

Файлы LIBCMT.LIB и LIBC.LIB содержат библиотечные функции языка C. Некоторые библиотечные функции C используют статические переменные. Например, функция *strtok* разработана таким образом, чтобы ее можно было вызывать более чем один раз последовательно, и хранит указатель на строку в статической памяти. В многопоточной программе каждый поток должен иметь свой статический указатель в функции *strtok*. Следовательно, многопоточная версия этой функции несколько отличается от однопоточной.

Кроме того, обратите внимание, что в программу RNDRCTMT.C включен заголовочный файл PROCESS.H. В этом файле описывается функция C с именем *_beginthread*, которая запускает новый поток (это будет показано ниже). Файл не определен, если не определен флаг _MT, и это еще один результат включения его при компиляции.

В функции *WinMain* значение *hwnd*, возвращаемое функцией *CreateWindow*, сохраняется в глобальной переменной. Аналогичное происходит со значениями переменных *cxClient* и *cyClient*, полученными из сообщения WM_SIZE в оконной процедуре.

Оконная процедура вызывает функцию *_beginthread* самым простым способом, используя только адрес функции потока, имеющей имя *Thread*, в качестве первого параметра, и остальные нулевые параметры. Функция потока возвращает значение типа VOID и имеет один параметр типа указатель на VOID. Функция *Thread* в программе RNDRCTMT не использует этот параметр.

После вызова функции *_beginthread* код функции потока, также как и код любой другой функции, которая может быть вызвана из функции потока, выполняется одновременно с оставшимся кодом программы. Два и более потока могут использовать одну и ту же функцию процесса. В этом случае автоматические локальные переменные (хранящиеся в стеке) уникальны для каждого потока; все статические переменные являются общими для всех потоков процесса. Поэтому, оконная процедура устанавливает значения переменных *cxClient* и *cyClient*, а функция *Thread* может их использовать.

Это были случаи, когда вам были необходимы данные уникальные более чем для одного потока. Обычно такие данные хранятся в статических переменных. В Windows 95 существует также так называемая локальная память потока, о которой речь пойдет ниже.

Задание на конкурсе программистов

3 октября 1986 года фирма Microsoft провела однодневный брифинг для технических редакторов и авторов компьютерных журналов, чтобы обсудить имевшиеся на тот момент языковые продукты, включая свою первую интерактивную среду разработки QUICKBASIC 2.0. Тогда Windows 1.0 было меньше года, и никто не знал, когда мы могли бы получить что-либо похожее на эту среду (на это потребовалось всего несколько лет). Что сделало это событие уникальным, так это подготовленный департаментом по связям с общественностью фирмы Microsoft конкурс программистов под девизом "Storm the Gates" (штурмует ворота). Билл Гейтс использовал QUICKBASIC 2.0, а люди, представлявшие компьютерную прессу, могли использовать любой языковой продукт, какой им больше понравится.

Конкретная задача, использовавшаяся в конкурсе, была выбрана среди нескольких других (разработанных в соответствии с принципом — около получаса на написание программы), предложенных представителями прессы. Она выглядела примерно так:

> *Создать псевдомногозадачную среду, состоящую из четырех окон. В первом окне должна выводиться последовательность возрастающих на единицу чисел, во втором — возрастающая последовательность простых чисел, в третьем — последовательность чисел Фибоначчи. (Последовательность чисел Фибоначчи начинается с 0 и 1, а каждое следующее число является суммой двух предыдущих, т. е. 0, 1, 1, 2, 3, 5, 8 и т. д.) Эти три окна либо прокручиваются, либо очищаются при полном заполнении окна числами. Четвертое окно должно отображать круги случайного радиуса. Программа завершается при нажатии клавиши <Escape>.*

Конечно, в октябре 1986 года такая программа, выполнявшаяся в MS DOS, не могла быть более чем имитацией многозадачности, и никто из участвовавших в конкурсе не осмелился — большинство из них не имело еще достаточных знаний для этого — написать программу для Windows. Кроме того, для написания такой программы с нуля под Windows потребовалось бы значительно больше времени чем полчаса.

Большинство участников конкурса написали программу, которая делила экран на четыре области. Программа содержала цикл, в котором последовательно обновлялись данные в каждом окне, а затем проверялось, не нажата ли клавиша <Escape>. Как это обычно и происходит в DOS, программа загружала процессор на все 100%.

Если бы программа была написана для Windows 1.0, то результат мог бы быть похожим на программу MULTI1, приведенную на рис. 14.2. Мы говорим "похожим", поскольку приведенная ниже программа преобразована для работы в 32-разрядной среде. Но структура и большая часть кода, кроме определения переменных и параметров функций, могли бы быть такими же.

MULTI1.MAK

```
#----------------------
# MULTI1.MAK make file
#----------------------

multi1.exe : multi1.obj
     $(LINKER) $(GUIFLAGS) -OUT:multi1.exe multi1.obj $(GUILIBS)

multi1.obj : multi1.c
     $(CC) $(CFLAGS) multi1.c
```

MULTI1.C

```
/*--------------------------------------
   MULTI1.C -- Multitasking Demo
               (c) Charles Petzold, 1996
  --------------------------------------*/

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

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

int cyChar;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Multi1";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

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

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Multitasking Demo",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

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

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

int CheckBottom(HWND hwnd, int cyClient, int iLine)
     {
     if(iLine * cyChar + cyChar > cyClient)
          {
          InvalidateRect(hwnd, NULL, TRUE);
          UpdateWindow(hwnd);
          iLine = 0;
          }
     return iLine;
```

```
     }

// Window 1: Display increasing sequence of numbers
// ------------------------------------------------

LRESULT APIENTRY WndProc1(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int   iNum, iLine;
     static short cyClient;
     char         szBuffer[16];
     HDC          hdc;

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

          case WM_TIMER :
               if(iNum < 0)
                    iNum = 0;

               iLine = CheckBottom(hwnd, cyClient, iLine);

               wsprintf(szBuffer, "%d", iNum++);

               hdc = GetDC(hwnd);
               TextOut(hdc, 0, iLine * cyChar, szBuffer, strlen(szBuffer));
               ReleaseDC(hwnd, hdc);

               iLine++;

               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }

// Window 2: Display increasing sequence of prime numbers
// ------------------------------------------------------

LRESULT APIENTRY WndProc2(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int   iNum = 1, iLine;
     static short cyClient;
     char         szBuffer[16];
     int          i, iSqrt;
     HDC          hdc;

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

          case WM_TIMER :
               do   {
                    if(++iNum < 0)
                         iNum = 0;
                    iSqrt =(int) sqrt(iNum);

                    for(i = 2; i <= iSqrt; i++)
                         if(iNum % i == 0)
                              break;
                    }
               while(i <= iSqrt);
```

```
               iLine = CheckBottom(hwnd, cyClient, iLine);

               wsprintf(szBuffer, "%d", iNum);

               hdc = GetDC(hwnd);
               TextOut(hdc, 0, iLine * cyChar, szBuffer, strlen(szBuffer));
               ReleaseDC(hwnd, hdc);

               iLine++;

               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }

// Window 3: Display increasing sequence of Fibonacci numbers
// ----------------------------------------------------------

LRESULT APIENTRY WndProc3(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int   iNum = 0, iNext = 1, iLine;
     static short cyClient;
     char         szBuffer[16];
     int          iTemp;
     HDC          hdc;

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

          case WM_TIMER :
               if(iNum < 0)
                    {
                    iNum  = 0;
                    iNext = 1;
                    }

               iLine = CheckBottom(hwnd, cyClient, iLine);

               wsprintf(szBuffer, "%d", iNum);
               hdc = GetDC(hwnd);
               TextOut(hdc, 0, iLine * cyChar, szBuffer, strlen(szBuffer));
               ReleaseDC(hwnd, hdc);

               iTemp  = iNum;
               iNum   = iNext;
               iNext += iTemp;

               iLine++;

               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }

// Window 4: Display circles of random radii
// -----------------------------------------

LRESULT APIENTRY WndProc4(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static short cxClient, cyClient;
     HDC          hdc;
```

```
     int          iDiameter;

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

          case WM_TIMER :
               InvalidateRect(hwnd, NULL, TRUE);
               UpdateWindow(hwnd);

               iDiameter = rand() %(max(1, min(cxClient, cyClient)));

               hdc = GetDC(hwnd);

               Ellipse(hdc,(cxClient - iDiameter) / 2,
                           (cyClient - iDiameter) / 2,
                           (cxClient + iDiameter) / 2,
                           (cyClient + iDiameter) / 2);

               ReleaseDC(hwnd, hdc);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }

// Main window to create child windows
// -----------------------------------
LRESULT APIENTRY WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char   *szChildClass[] = { "Child1", "Child2",
                                       "Child3", "Child4" };
     static HWND    hwndChild[4];
     static WNDPROC ChildProc[] = { WndProc1, WndProc2,
                                    WndProc3, WndProc4 };
     HINSTANCE      hInstance;
     int            i, cxClient, cyClient;
     WNDCLASSEX     wndclass;

     switch(iMsg)
          {
          case WM_CREATE :
               hInstance =(HINSTANCE) GetWindowLong(hwnd, GWL_HINSTANCE);

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

               for(i = 0; i < 4; i++)
                    {
                    wndclass.lpfnWndProc   = ChildProc[i];
                    wndclass.lpszClassName = szChildClass[i];

                    RegisterClassEx(&wndclass);

                    hwndChild[i] = CreateWindow(szChildClass[i], NULL,
```

```
                    WS_CHILDWINDOW | WS_BORDER | WS_VISIBLE,
                    0, 0, 0, 0, hwnd,(HMENU) i, hInstance, NULL);
               }

          cyChar = HIWORD(GetDialogBaseUnits());
          SetTimer(hwnd, 1, 10, NULL);
          return 0;

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

          for(i = 0; i < 4; i++)
               MoveWindow(hwndChild[i],(i % 2) * cxClient / 2,
                                       (i > 1) * cyClient / 2,
                          cxClient / 2, cyClient / 2, TRUE);
          return 0;

     case WM_TIMER :
          for(i = 0; i < 4; i++)
               SendMessage(hwndChild[i], WM_TIMER, wParam, lParam);

          return 0;

     case WM_CHAR :
          if(wParam == '\x1B')
               DestroyWindow(hwnd);

          return 0;

     case WM_DESTROY :
          KillTimer(hwnd, 1);
          PostQuitMessage(0);
          return 0;
     }
return DefWindowProc(hwnd, iMsg, wParam, lParam);
}
```

Рис. 14.2 Программа MULTI1

Данная программа не использует ничего такого, чего мы не видели раньше. Главное окно создает четыре дочерних окна, каждое из которых занимает четверть рабочей области родительского окна. Главное окно также устанавливает таймер Windows и посылает сообщения WM_TIMER каждому из четырех дочерних окон.

Обычно, Windows-программа могла бы сохранять достаточно информации для того, чтобы обновлять содержимое окон в процессе обработки сообщения WM_PAINT. Программа MULTI1 не делает этого, а окна рисуются и обновляются настолько быстро, что, видимо, в этом и нет необходимости.

Генератор простых чисел, созданный в функции *WndProc2*, не очень эффективен, но он работает. Число является простым, если у него нет других делителей кроме 1 и самого себя. Чтобы проверить, является ли конкретное число простым, не требуется делить его на все числа и проверять остатки от деления. Достаточно извлечь квадратный корень из этого числа. Вычисление квадратного корня является причиной странного, казалось бы, введения арифметики с плавающей точкой в программу, основанную на операциях с целыми.

В программе MULTI1 нет ничего неправильного. Использование таймера Windows — это отличный путь имитации многозадачности в ранних версиях Windows и Windows 95. Однако, использование таймера иногда замедляет программу. Если программа может обновить все свои окна во время обработки одного сообщения WM_TIMER, имея запас времени, то при этом не используются все возможности компьютера.

Одно из возможных решений этой проблемы — выполнение двух и более обновлений во время обработки одного сообщения WM_TIMER. Но скольких? Это зависит в основном от скорости машины, которая может меняться в широких пределах. Единственное, что не хотелось бы делать, так это писать программу, настроенную только на конкретную машину — 25МГц 386 или 50МГц 486 или какую-нибудь 700МГц 786.

Решение с использованием многопоточности

Давайте рассмотрим решение указанной задачи с применением многопоточности. Программа MULTI2 приведена на рис. 14.3.

MULTI2.MAK

```
#---------------------
# MULTI2.MAK make file
#---------------------

multi2.exe : multi2.obj
     $(LINKER) $(GUIFLAGS) -OUT:multi2.exe multi2.obj $(GUILIBS)

multi2.obj : multi2.c
     $(CC) $(CFLAGSMT) multi2.c
```

MULTI2.C

```
/*--------------------------------------
   MULTI2.C -- Multitasking Demo
               (c) Charles Petzold, 1996
  --------------------------------------*/

#include <windows.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <process.h>

typedef struct
     {
     HWND hwnd;
     int  cxClient;
     int  cyClient;
     int  cyChar;
     BOOL bKill;
     }
     PARAMS, *PPARAMS;

LRESULT APIENTRY WndProc(HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Multi2";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

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

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Multitasking Demo",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
```

```
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

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

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

int CheckBottom(HWND hwnd, int cyClient, int cyChar, int iLine)
     {
     if(iLine * cyChar + cyChar > cyClient)
          {
          InvalidateRect(hwnd, NULL, TRUE);
          UpdateWindow(hwnd);
          iLine = 0;
          }
     return iLine;
     }
// Window 1: Display increasing sequence of numbers
// ------------------------------------------------

void Thread1(PVOID pvoid)
     {
     int     iNum = 0, iLine = 0;
     char    szBuffer[16];
     HDC     hdc;
     PPARAMS pparams;

     pparams =(PPARAMS) pvoid;

     while(!pparams->bKill)
          {
          if(iNum < 0)
               iNum = 0;

          iLine = CheckBottom(pparams->hwnd,   pparams->cyClient,
                              pparams->cyChar, iLine);

          wsprintf(szBuffer, "%d", iNum++);

          hdc = GetDC(pparams->hwnd);

          TextOut(hdc, 0, iLine * pparams->cyChar,
                   szBuffer, strlen(szBuffer));

          ReleaseDC(pparams->hwnd, hdc);

          iLine++;
          }
     _endthread();
     }

LRESULT APIENTRY WndProc1(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static PARAMS params;

     switch(iMsg)
          {
          case WM_CREATE :
```

```
               params.hwnd = hwnd;
               params.cyChar = HIWORD(GetDialogBaseUnits());
               _beginthread(Thread1, 0, &params);
               return 0;

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

          case WM_DESTROY :
               params.bKill = TRUE;
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }

// Window 2: Display increasing sequence of prime numbers
// ------------------------------------------------------

void Thread2(PVOID pvoid)
     {
     char    szBuffer[16];
     int     iNum = 1, iLine = 0, i, iSqrt;
     HDC     hdc;
     PPARAMS pparams;

     pparams =(PPARAMS) pvoid;

     while(!pparams->bKill)
          {
          do
               {
               if(++iNum < 0)
                    iNum = 0;

               iSqrt =(int) sqrt(iNum);

               for(i = 2; i <= iSqrt; i++)
                    if(iNum % i == 0)
                         break;
               }
          while(i <= iSqrt);

          iLine = CheckBottom(pparams->hwnd,   pparams->cyClient,
                              pparams->cyChar, iLine);

          wsprintf(szBuffer, "%d", iNum);

          hdc = GetDC(pparams->hwnd);

          TextOut(hdc, 0, iLine * pparams->cyChar,
                  szBuffer, strlen(szBuffer));

          ReleaseDC(pparams->hwnd, hdc);

          iLine++;
          }
     _endthread();
     }

LRESULT APIENTRY WndProc2(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static PARAMS params;

     switch(iMsg)
```

```
          {
          case WM_CREATE :
               params.hwnd = hwnd;
               params.cyChar = HIWORD(GetDialogBaseUnits());
               _beginthread(Thread2, 0, &params);
               return 0;

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

          case WM_DESTROY :
               params.bKill = TRUE;
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }

// Window 3: Display increasing sequence of Fibonacci numbers
// ----------------------------------------------------------

void Thread3(PVOID pvoid)
     {
     char    szBuffer[16];
     int     iNum = 0, iNext = 1, iLine = 0, iTemp;
     HDC     hdc;
     PPARAMS pparams;

     pparams =(PPARAMS) pvoid;

     while(!pparams->bKill)
          {
          if(iNum < 0)
               {
               iNum  = 0;
               iNext = 1;
               }

          iLine = CheckBottom(pparams->hwnd,   pparams->cyClient,
                              pparams->cyChar, iLine);

          wsprintf(szBuffer, "%d", iNum);

          hdc = GetDC(pparams->hwnd);

          TextOut(hdc, 0, iLine * pparams->cyChar,
                   szBuffer, strlen(szBuffer));

          ReleaseDC(pparams->hwnd, hdc);

          iTemp  = iNum;
          iNum   = iNext;
          iNext += iTemp;

          iLine++;
          }
     _endthread();
     }

LRESULT APIENTRY WndProc3(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static PARAMS params;

     switch(iMsg)
          {
```

```
          case WM_CREATE :
               params.hwnd = hwnd;
               params.cyChar = HIWORD(GetDialogBaseUnits());
               _beginthread(Thread3, 0, &params);
               return 0;

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

          case WM_DESTROY :
               params.bKill = TRUE;
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }

// Window 4: Display circles of random radii
// -----------------------------------------

void Thread4(PVOID pvoid)
     {
     HDC     hdc;
     int     iDiameter;
     PPARAMS pparams;

     pparams =(PPARAMS) pvoid;

     while(!pparams->bKill)
          {
          InvalidateRect(pparams->hwnd, NULL, TRUE);
          UpdateWindow(pparams->hwnd);

          iDiameter = rand() %(max(1,
                               min(pparams->cxClient, pparams->cyClient)));

          hdc = GetDC(pparams->hwnd);

          Ellipse(hdc,(pparams->cxClient - iDiameter) / 2,
                      (pparams->cyClient - iDiameter) / 2,
                      (pparams->cxClient + iDiameter) / 2,
                      (pparams->cyClient + iDiameter) / 2);

          ReleaseDC(pparams->hwnd, hdc);
          }
     _endthread();
     }

LRESULT APIENTRY WndProc4(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static PARAMS params;

     switch(iMsg)
          {
          case WM_CREATE :
               params.hwnd = hwnd;
               params.cyChar = HIWORD(GetDialogBaseUnits());
               _beginthread(Thread4, 0, &params);
               return 0;

          case WM_SIZE :
               params.cxClient = LOWORD(lParam);
               params.cyClient = HIWORD(lParam);
               return 0;
```

```
          case WM_DESTROY :
               params.bKill = TRUE;
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }

// Main window to create child windows
// -----------------------------------

LRESULT APIENTRY WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char   *szChildClass[] = { "Child1", "Child2",
                                       "Child3", "Child4" };
     static HWND    hwndChild[4];
     static WNDPROC ChildProc[] = { WndProc1, WndProc2,
                                    WndProc3, WndProc4 };
     HINSTANCE      hInstance;
     int            i, cxClient, cyClient;
     WNDCLASSEX     wndclass;

     switch(iMsg)
          {
          case WM_CREATE :
               hInstance =(HINSTANCE) GetWindowLong(hwnd, GWL_HINSTANCE);

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

               for(i = 0; i < 4; i++)
                    {
                    wndclass.lpfnWndProc   = ChildProc[i];
                    wndclass.lpszClassName = szChildClass[i];

                    RegisterClassEx(&wndclass);

                    hwndChild[i] = CreateWindow(szChildClass[i], NULL,
                          WS_CHILDWINDOW | WS_BORDER | WS_VISIBLE,
                          0, 0, 0, 0, hwnd,(HMENU) i, hInstance, NULL);
                    }

               return 0;

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

               for(i = 0; i < 4; i++)
                    MoveWindow(hwndChild[i],(i % 2) * cxClient / 2,
                                            (i > 1) * cyClient / 2,
                               cxClient / 2, cyClient / 2, TRUE);
               return 0;

          case WM_CHAR :
               if(wParam == '\x1B')
                    DestroyWindow(hwnd);
```

```
            return 0;

        case WM_DESTROY :
            PostQuitMessage(0);
            return 0;
        }
    return DefWindowProc(hwnd, iMsg, wParam, lParam);
    }
```

Рис. 14.3 Программа MULTI2

Функции *WinMain* и *WndProc* в программе MULTI2.C очень похожи на соответствующие функции в программе MULTI1.C. Функция *WndProc* регистрирует четыре класса окна для четырех окон, создает эти окна и устанавливает их размер в процессе обработки сообщения WM_SIZE. Единственное отличие в данной функции *WndProc* состоит в том, что она не устанавливает таймер и не посылает сообщений WM_TIMER.

Основное отличие программы MULTI2 состоит в том, что каждая дочерняя оконная процедура создает свой поток обработки, вызывая функцию *_beginthread* во время обработки сообщения WM_CREATE. Всего в программе MULTI2 имеется пять потоков обработки, которые выполняются совместно. Первичный поток содержит оконную процедуру главного окна и четыре оконные процедуры дочерних окон. Остальные четыре потока используют функции с именами *Thread1*, *Thread2* и т. д. Эти четыре потока отвечают за вывод в четырех окнах на экране.

В коде многопоточной программы RNDRCTMT не использовался третий параметр функции *_beginthread*. Этот параметр позволяет потоку, который создает другой поток, передавать информацию этому потоку в виде 32-разрядной переменной. Обычно такая переменная является указателем на структуру данных. Это дает возможность создающему и создаваемому потокам совместно владеть информацией без использования глобальных переменных. Как вы можете видеть, в программе MULTI2 нет глобальных переменных.

В программе MULTI2 определена структура с именем PARAMS (в начале программы) и указатель на эту структуру — PPARAMS. В этой структуре пять полей: описатель окна, ширина и высота окна, высота символа и булева переменная с именем *bKill*. Последнее поле позволяет создающему потоку передавать создаваемому потоку информацию о том, в какой момент времени он должен закончить работу.

Рассмотрим функцию *WndProc1* — оконную процедуру дочернего окна, которое выводит последовательность увеличивающихся на единицу чисел. Оконная процедура стала очень простой. Единственная локальная переменная — это структура PARAMS. Во время обработки сообщения WM_CREATE устанавливаются значения полей *hwnd* и *cyChar* этой структуры, и вызывается функция *_beginthread* для создания нового потока, использующего функцию *Thread1*, и передавая в качестве параметра указатель на эту структуру. Во время обработки сообщения WM_SIZE функция *WndProc1* устанавливает значение поля *cyClient* этой структуры, а во время обработки сообщения WM_DESTROY устанавливает значение поля *bKill* в TRUE. Выполнение функции *Thread1* завершается при вызове функции *_endthread*. Это не является строго необходимым, поскольку поток уничтожается после выхода из функции потока. Однако, функция *_endthread* является полезной при выходе из потока, сидящего глубоко в иерархии потоков обработки.

Функция *Thread1* осуществляет рисование в окне, и она выполняется совместно с другими четырьмя потоками в процессе. Функция получает указатель на структуру PARAMS и выполняется циклически, используя цикл *while*, проверяя каждый раз значение поля *bKill*. Если в поле *bKill* содержится значение FALSE, то функция выполняет те же действия, что и во время обработки сообщения WM_TIMER в программе MULTI1 — формирование числа, получение описателя контекста устройства и отображение числа на экране с использованием функции *TextOut*.

Как вы увидите, когда запустите программу MULTI2 на выполнение под Windows 95, обновление окон происходит значительно быстрее, чем в программе MULTI1, иллюстрируя тем самым, что программа использует мощность процессора более эффективно. Существует еще одно отличие между программами MULTI1 и MULTI2: обычно при перемещении окна или изменении его размеров оконная процедура по умолчанию входит в модальный цикл, и весь вывод в окно приостанавливается. В программе MULTI2 вывод в этом случае продолжается.

Еще есть проблемы?

Может показаться, что программа MULTI2 не настолько убедительна, как могла бы быть. Чтобы увидеть, чего нам действительно удалось достичь, давайте рассмотрим некоторые "недостатки" многопоточности в программе MULTI2.C, взяв для примера функции *WndProc1* и *Thread1*.

Функция *WndProc1* выполняется в главном потоке программы MULTI2, а функция *Thread1* выполняется параллельно с ней. Моменты времени, когда Windows 95 переключается между этими двумя потоками, являются переменными и заранее непредсказуемыми. Предположим, что функция *Thread1* активна и только что выполнила код, проверяющий, имеет ли поле *bKill* структуры PARAMS значение TRUE. Оно не равно TRUE, а Windows 95 переключает управление на главный поток, в котором пользователь завершает приложение. Функция *WndProc1*

получает сообщение WM_DESTROY и устанавливает значение поля *bKill* равным TRUE. Но это уже слишком поздно. Внезапно операционная система переключает управление на функцию *Thread1*, и эта функция пытается получить описатель контекста устройства уже несуществующего окна.

Оказывается, что здесь нет никакой проблемы. Windows 95 достаточно устойчива, и графические функции просто не выполняются, не вызывая при этом никаких проблем.

Правильная техника программирования многопоточных приложений включает использование синхронизации потоков (thread synchronization) (и в частности, критических разделов, critical sections). Синхронизацию потоков мы рассмотрим более детально позже. В нескольких словах, критические разделы ограничиваются вызовами функций *EnterCriticalSection* и *LeaveCriticalSection*. Если один поток входит в критический раздел, то другой поток уже не может войти в него. Для второго потока вызов функции *EnterCriticalSection* приводит к приостановке выполнения внутри этой функции до тех пор, пока первый поток не вызовет функцию *LeaveCriticalSection*.

Другой возможной проблемой в программе MULTI2 является то, что главный поток может получить сообщение WM_ERASEBKGND или WM_PAINT в то время, как вторичный поток осуществляет рисование в окне. И снова использование критических разделов могло бы помочь предотвратить какие-либо проблемы, которые могли бы возникнуть при попытке двух потоков осуществить одновременное рисование в окне. Однако, эксперименты показывают, что Windows 95 правильно организует последовательность доступа к графическим функциям рисования. Таким образом, один поток не может рисовать в окне, пока другой поток выполняет такое же действие.

Документация по Windows 95 предупреждает об одной области, в которой графические функции не упорядочиваются. Это использование объектов GDI, таких как перья, кисти, шрифты, битовые образы и палитры. Существует возможность для одного потока разрушить объект, который в это время используется другим потоком. Решение этой проблемы состоит в использовании критических разделов. Однако, еще лучше делать так, чтобы объекты GDI не разделялись между различными потоками.

О пользе использования функции *Sleep*

Выше было показано, как лучше организовать архитектуру программы, использующей многопоточность, а именно, чтобы первичный поток создавал все окна в программе, содержал все оконные процедуры этих окон и обрабатывал все сообщения. Вторичные потоки выполняют фоновые задачи или задачи, протяженные во времени.

Однако, предположим, что вы хотите реализовать анимацию во вторичном потоке. Обычно анимация в Windows осуществляется с использованием сообщения WM_TIMER. Но если вторичный поток не создает окно, то он не может получить это сообщение. А без задания определенного темпа анимация могла бы осуществляться слишком быстро.

Решение состоит в использовании функции *Sleep*. Поток вызывает функцию *Sleep* для того, чтобы добровольно отложить свое выполнение. Единственный параметр этой функции — время, задаваемое в миллисекундах. Функция *Sleep* не осуществляет возврата до тех пор, пока не истечет указанное время. В течение него выполнение потока приостанавливается и выделения для него процессорного времени не происходит (хотя очевидно, что для потока все-таки требуется какое-то незначительное время, за которое система должна определить, пора возобновлять выполнение потока или нет). Если параметр функции *Sleep* задан равным нулю, то поток будет лишен остатка выделенного ему кванта процессорного времени.

Когда поток вызывает функцию *Sleep*, задержка на заданное время относится только к этому потоку. Система продолжает выполнять другие потоки этого и других процессов. Функция *Sleep* была использована в программе SCRAMBLE в главе 4 для замедления взаимного обмена содержимым прямоугольников на экране.

Обычно вам не требуется использовать функцию *Sleep* в первичном потоке, т. к. она замедляет обработку сообщений, но поскольку программа SCRAMBLE не создает никаких окон, проблем при использовании этой функции не возникает.

Синхронизация потоков

Примерно раз в году светофор на каком-нибудь оживленном перекрестке перестает работать. В результате возникает хаос, и хотя машины обычно избегают аварий, довольно часто они сближаются слишком опасно.

Мы могли бы описать перекресток двух дорог в терминах программирования как критический раздел (critical section). Машина, двигающаяся на юг, и машина, двигающаяся на запад, не могут проехать через перекресток одновременно, избежав при этом столкновения. В зависимости от интенсивности движения возможны разные подходы к решению этой проблемы. При очень маленьком движении через перекресток, когда хорошая видимость, водители могут рассчитывать, что они правильно уступят дорогу друг другу. Более оживленное движение требует установки знака "Стоп", а для еще более напряженного движения необходим светофор. Светофор помогает координировать движение через перекресток (если он работает, конечно).

Критический раздел

В однозадачной операционной системе обычные программы не нуждаются в "светофорах" для координации их действий. Они выполняются так, как будто они являются хозяевами дороги, по которой они следуют. Не существует ничего, что могло бы вмешаться в то, что они делают.

Даже в многозадачной операционной системе большинство программ выполняются независимо друг от друга. Но некоторые проблемы все же могут возникнуть. Например, двум программам может понадобиться читать и писать в один файл в одно и то же время. Для таких случаев операционная система поддерживает механизм разделения файлов (shared files) и блокирования отдельных фрагментов файла (record locking).

Однако, в операционной системе, поддерживающей многопоточность, такое решение может внести путаницу и создать потенциальную опасность. Разделение данных между двумя и более потоками является общим случаем. Например, один поток может обновлять одну или более переменных, а другой может использовать эти переменные. Иногда в этой ситуации может возникнуть проблема, а иногда — нет. (Помните, что операционная система может переключать управление потоками только между инструкциями машинного кода. Если простое целое число разделяется между двумя потоками, то изменение этой переменной обычно осуществляется одной инструкцией машинного кода, и потенциальные проблемы сводятся к минимуму.)

Однако, предположим, что потоки разделяют несколько переменных или структуру данных. Часто эти сложные переменные или поля структур данных должны быть согласованными между собой. Операционная система может прерывать поток в середине процесса обновления этих переменных. В этом случае поток, который затем использует эти переменные, будет иметь дело с несогласованными данными.

В результате бы возникла коллизия, и нетрудно представить себе, как такого рода ошибка может привести к краху программы. В этой ситуации нам необходимо нечто похожее на светофор, который мог бы синхронизировать и координировать работу потоков. Таким средством и является критический раздел. Критический раздел — это блок кода, при выполнении которого поток не может быть прерван.

Имеется четыре функции для работы с критическими разделами. Чтобы их использовать, вам необходимо определить объект типа критический раздел, который является глобальной переменной типа CRITICAL_SECTION. Например,

```
CRITICAL_SECTION cs;
```

Тип данных CRITICAL_SECTION является структурой, но ее поля используются только внутри Windows. Объект типа критический раздел сначала должен быть инициализирован одним из потоков программы с помощью функции:

```
InitializeCriticalSection(&cs);
```

Эта функция создает объект критический раздел с именем *cs*. В документации содержится следующее предупреждение: "Объект критический раздел не может быть перемещен или скопирован. Процесс также не должен модифицировать объект, а должен обращаться с ним, как с "черным ящиком". "

После инициализации объекта критический раздел поток входит в критический раздел, вызывая функцию:

```
EnterCriticalSection(&cs);
```

В этот момент поток становится владельцем объекта. Два различных потока не могут быть владельцами одного объекта критический раздел одновременно. Следовательно, если один поток вошел в критический раздел, то следующий поток, вызывая функцию *EnterCriticalSection* с тем же самым объектом критический раздел, будет задержан внутри функции. Возврат из функции произойдет только тогда, когда первый поток покинет критический раздел, вызвав функцию:

```
LeaveCriticalSection(&cs);
```

В этот момент второй поток, задержанный в функции *EnterCriticalSection*, станет владельцем критического раздела, и его выполнение будет возобновлено.

Когда объект критический раздел больше не нужен вашей программе, его можно удалить с помощью функции:

```
DeleteCriticalSection(&cs);
```

Это приведет к освобождению всех ресурсов системы, задействованных для поддержки объекта критический раздел.

Механизм критических разделов основан на принципе взаимного исключения (mutual exclusion). Этот термин нам еще встретится при дальнейшем рассмотрении синхронизации потоков. Только один поток может быть владельцем критического раздела в каждый конкретный момент времени. Следовательно, один поток может войти в критический раздел, установить значения полей структуры и выйти из критического раздела. Другой поток, использующий эту структуру, также мог бы войти в критический раздел перед осуществлением доступа к полям структуры, а затем выйти из критического раздела.

Обратите внимание, что возможно определение нескольких объектов типа критический раздел, например, *cs1* и *cs2*. Если в программе имеется четыре потока, и два первых из них разделяют некоторые данные, то они могут использовать первый объект критический раздел, а два других потока, также разделяющих другие данные, могут использовать второй объект критический раздел.

Обратите внимание, что надо быть весьма осторожным при использовании критического раздела в главном потоке. Если вторичный поток проводит слишком много времени в его собственном критическом разделе, то это может привести к "зависанию" главного потока на слишком большой период времени.

Объект Mutex

Существует одно ограничение в использовании критических разделов. Оно заключается в том, что их можно применять для синхронизации потоков только в рамках одного процесса. Но бывают случаи, когда необходимо синхронизировать действия потоков различных процессов, которые разделяют какие-либо ресурсы (например, память). Использовать критические разделы в такой ситуации нельзя. Вместо них подключаются объекты типа mutex (mutex object).

Составное слово "mutex" происходит из словосочетания "mutual exclusion", что означает взаимное исключение, и очень точно отражает назначение объектов. Мы хотим предотвратить возможность прерывания потока в программе до тех пор, пока не будет выполнено обновление или использование разделяемых данных. Или, используя употребленную выше аналогию, мы хотим, чтобы поток транспорта, движущийся на юг, и поток транспорта, движущийся на запад, не пересекались на перекрестке.

Уведомления о событиях

Мы можем определить понятие большой работы как действия, выполняя которые, программа нарушит "правило 1/10 секунды". Примерами большой работы могут служить: проверка орфографии в текстовых процессорах, сортировка и индексирование файлов баз данных, пересчет электронной таблицы, печать и даже сложное рисование. Конечно, как мы уже знаем, лучшее решение состоит в следовании "правилу 1/10 секунды", т. е. в передаче большой работы вторичным потокам обработки. Эти вторичные потоки не создают окон и, значит, не ограничены "правилом 1/10 секунды".

Часто бывает, что вторичному потоку надо проинформировать первичный поток о том, что он завершился, или первичному потоку надо прервать работу, выполняемую вторичным потоком. Этот случай мы и рассмотрим.

Программа BIGJOB1

В качестве гипотетической большой работы рассмотрим последовательность вычислений с плавающей точкой, известную иногда как "дикий" тест производительности. В этих вычислениях значение целого числа увеличивается на единицу в так называемой карусельной манере: сначала число возводится в квадрат, затем из результата извлекается квадратный корень, выполняются функции *log* и *exp*, отменяющие действия друг друга, затем выполняются функции *atan* и *tan*, и наконец, просто прибавляется 1 для получения результата.

Программа BIGJOB1 приведена на рис. 14.4.

BIGJOB1.MAK

```
#-----------------------
# BIGJOB1.MAK make file
#-----------------------

bigjob1.exe : bigjob1.obj
     $(LINKER) $(GUIFLAGS) -OUT:bigjob1.exe bigjob1.obj $(GUILIBS)

bigjob1.obj : bigjob1.c
     $(CC) $(CFLAGSMT) bigjob1.c
```

BIGJOB1.C

```
/*--------------------------------------
   BIGJOB1.C -- Multithreading Demo
                (c) Charles Petzold, 1996
  --------------------------------------*/

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

```
#define REP             100000

#define STATUS_READY     0
#define STATUS_WORKING   1
#define STATUS_DONE      2

#define WM_CALC_DONE    (WM_USER + 0)
#define WM_CALC_ABORTED (WM_USER + 1)

typedef struct
     {
     HWND hwnd;
     BOOL bContinue;
     }
     PARAMS, *PPARAMS;

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "BigJob1";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

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

     RegisterClassEx(&wndclass);
     hwnd = CreateWindow(szAppName, "Multithreading Demo",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

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

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

void Thread(PVOID pvoid)
     {
     double  A = 1.0;
     INT     i;
     LONG    lTime;
     PPARAMS pparams;

     pparams =(PPARAMS) pvoid;
```

```
     lTime = GetCurrentTime();

     for(i = 0; i < REP && pparams->bContinue; i++)
          A = tan(atan(exp(log(sqrt(A * A))))) + 1.0;

     if(i == REP)
          {
          lTime = GetCurrentTime() - lTime;
          SendMessage(pparams->hwnd, WM_CALC_DONE, 0, lTime);
          }
     else
          SendMessage(pparams->hwnd, WM_CALC_ABORTED, 0, 0);

     _endthread();
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char  *szMessage[] = { "Ready(left mouse button begins)",
                                   "Working(right mouse button aborts)",
                                   "%d repetitions in %ld msec" };
     static INT    iStatus;
     static LONG   lTime;
     static PARAMS params;
     char          szBuffer[64];
     HDC           hdc;
     PAINTSTRUCT   ps;
     RECT          rect;

     switch(iMsg)
          {
          case WM_LBUTTONDOWN :
               if(iStatus == STATUS_WORKING)
                    {
                    MessageBeep(0);
                    return 0;
                    }

               iStatus = STATUS_WORKING;

               params.hwnd = hwnd;
               params.bContinue = TRUE;

               _beginthread(Thread, 0, &params);

               InvalidateRect(hwnd, NULL, TRUE);
               return 0;

          case WM_RBUTTONDOWN :
               params.bContinue = FALSE;
               return 0;

          case WM_CALC_DONE :
               lTime = lParam;
               iStatus = STATUS_DONE;
               InvalidateRect(hwnd, NULL, TRUE);
               return 0;

          case WM_CALC_ABORTED :
               iStatus = STATUS_READY;
               InvalidateRect(hwnd, NULL, TRUE);
               return 0;

          case WM_PAINT :
```

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

            GetClientRect(hwnd, &rect);

            wsprintf(szBuffer, szMessage[iStatus], REP, lTime);

            DrawText(hdc, szBuffer, -1, &rect,
                     DT_SINGLELINE | DT_CENTER | DT_VCENTER);

            EndPaint(hwnd, &ps);
            return 0;
        case WM_DESTROY :
            PostQuitMessage(0);
            return 0;
        }
    return DefWindowProc(hwnd, iMsg, wParam, lParam);
    }
```

Рис. 14.4 Программа BIGJOB1

Это достаточно простая программа, но на ее примере можно увидеть, как она реализует подход к выполнению больших работ в многопоточной программе. Чтобы использовать программу BIGJOB1, щелкните левой кнопкой мыши в рабочей области окна. Это приведет к запуску 100 000 повторов "диких" вычислений, что на 33МГц процессоре 386 потребует около 8 секунд. Когда вычисления будут закончены, затраченное время будет отображено в окне. Во время расчетов вы можете щелкнуть правой кнопкой мыши в рабочей области окна для прерывания вычислений.

Теперь давайте рассмотрим, как это работает:

Оконная процедура содержит статическую переменную с именем *iStatus* (которая может быть установлена в одно из трех значений, определенных в начале программы, начинающихся со слова STATUS). Эта переменная показывает, находится ли программа в состоянии готовности к вычислениям, в процессе вычислений или вычисления завершены. Программа использует переменную *iStatus* в процессе обработки сообщения WM_PAINT для вывода на экран соответствующей строки символов в центре рабочей области.

Кроме того, оконная процедура содержит статическую структуру типа PARAMS, определенную также в начале программы, для разделения данных между оконной процедурой и вторичным потоком. В этой структуре только два поля — поле *hwnd* (описатель окна программы) и поле *bContinue*, которое является булевой переменной для указания вторичному потоку — продолжать ему вычисления или нет.

Когда вы щелкаете левой кнопкой мыши в рабочей области, оконная процедура присваивает переменной *iStatus* значение STATUS_WORKING, а также заполняет оба поля структуры PARAMS. В поле *hwnd* заносится, очевидно, значение описателя окна, а в поле *bContinue* заносится значение TRUE.

Затем оконная процедура создает вторичный поток, вызывая функцию *_beginthread*. Функция вторичного потока с именем *Thread* начинается с вызова функции *GetCurrentTime* для получения значения времени в миллисекундах, прошедшего с момента запуска Windows. Затем она входит в цикл *for* для выполнения "диких" вычислений. Обратите внимание, что поток может выйти из цикла, если в какой-либо момент в поле *bContinue* окажется значение FALSE.

После выполнения цикла *for* функция потока проверяет, действительно ли выполнено 100 000 повторов вычислений. Если да, то она снова вызывает функцию *GetCurrentTime*, чтобы определить затраченное на вычисления время, а затем использует функцию *SendMessage* для отправки оконной процедуре определенного в программе сообщения WM_CALC_DONE с указанием затраченного времени в параметре *lParam*. Если вычисления были прерваны (в том случае, когда поле *bContinue* структуры PARAMS было установлено в FALSE во время выполнения цикла), то поток посылает оконной процедуре сообщение WM_CALC_ABORTED. Затем поток красиво завершается вызовом функции *_endthread*.

Внутри оконной процедуры поле *bContinue* структуры PARAMS принимает значение FALSE, когда вы щелкаете правой кнопкой мыши в рабочей области окна. Таким образом происходит прерывание вычислений до их завершения.

Оконная процедура обрабатывает сообщение WM_CALC_DONE, сначала сохраняя затраченное на вычисления время. Обработка сообщений WM_CALC_DONE или WM_CALC_ABORTED продолжается вызовом функции *InvalidateRect* для того, чтобы сгенерировать сообщение WM_PAINT и отобразить на экране новую строку текста в рабочей области.

Неплохо включить в программу заготовку (такую как поле *bContinue* структуры PARAMS), позволяющую потоку завершиться "чисто". Функция *KillThread* могла бы быть использована в том случае, когда "чистое" завершение

невозможно. Причина может состоять в том, что потоки имеют возможность захватывать ресурсы, например, память. Если эта память не освобождается при завершении потока, то она продолжает оставаться захваченной. Потоки — не процессы: захваченные ресурсы разделяются между всеми потоками процесса, и поэтому они не освобождаются автоматически при завершении потока. Хороший стиль программирования предписывает, чтобы поток освобождал все захваченные им ресурсы.

Обратите внимание также на то, что третий поток может быть создан, пока второй еще выполняется. Это может произойти, если Windows 95 переключит управление со второго потока на первый между вызовами функций *SendMessage* и *_endthread*. В этом случае оконная процедура создаст новый поток в ответ на щелчок мыши. Здесь это не является проблемой, но она может возникнуть в одном из ваших собственных приложений, и тогда вам следует использовать критические разделы для того, чтобы избежать коллизий между потоками.

Объект Event

Программа BIGJOB1 создает поток каждый раз, когда необходимо произвести "дикие" вычисления; поток завершается после выполнения вычислений.

Альтернативным вариантом является сохранение потока готовым к выполнению на протяжении всего времени работы программы, и приведение его в действие только при необходимости. Это идеальный случай для применения объекта Событие (event object).

Объект Событие может быть либо свободным (signaled) или установленным (set), либо занятым (non-signaled) или сброшенным (reset). Вы можете создать объект Событие с помощью функции:

```
hEvent = CreateEvent(&sa, fManual, fInitial, pszName);
```

Первый параметр (указатель на структуру типа SECURITY_ATTRIBUTES) и последний параметр (имя объекта событие) имеют смысл только в том случае, когда объект Событие разделяется между процессами. В случае с одним процессом эти параметры обычно имеют значение NULL. Установите значение параметра *fInitial* равным TRUE, если вы хотите, чтобы объект Событие был изначально свободным, или равным FALSE, чтобы он был занятым. Параметр *fManual* будет описан несколько позже.

Для того, чтобы сделать свободным существующий объект Событие, вызовите функцию:

```
SetEvent(hEvent);
```

Чтобы сделать объект Событие занятым, вызовите функцию:

```
ResetEvent(hEvent);
```

Обычно программа вызывает функцию:

```
WaitForSingleObject(hEvent, dwTimeOut);
```

где второй параметр имеет значение INFINITE. Возврат из функции происходит немедленно, если объект Событие в настоящее время свободен. В противном случае поток будет приостановлен в функции до тех пор, пока событие не станет свободным. Вы можете установить значение тайм-аута во втором параметре, задав его величину в миллисекундах. Тогда возврат из функции произойдет, когда объект Событие станет свободным или истечет тайм-аут.

Если параметр *fManual* при вызове функции *CreateEvent* имеет значение FALSE, то объект Событие автоматически становится занятым, когда осуществляется возврат из функции *WaitForSingleObject*. Эта особенность обычно позволяет избежать использования функции *ResetEvent*.

Рассмотрим теперь программу BIGJOB2, представленную на рис. 14.5.

BIGJOB2.MAK

```
#-----------------------
# BIGJOB2.MAK make file
#------------ ---------

bigjob2.exe : bigjob2.obj
     $(LINKER) $(GUIFLAGS) -OUT:bigjob2.exe bigjob2.obj $(GUILIBS)

bigjob2.obj : bigjob2.c
     $(CC) $(CFLAGSMT) bigjob2.c
```

BIGJOB2.C

```
/*---------------------------------------
   BIGJOB2.C -- Multithreading Demo
              (c) Charles Petzold, 1996
```

```
  ---------------------------------------*/

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

#define REP             100000

#define STATUS_READY     0
#define STATUS_WORKING   1
#define STATUS_DONE      2

#define WM_CALC_DONE    (WM_USER + 0)
#define WM_CALC_ABORTED (WM_USER + 1)

typedef struct
     {
     HWND   hwnd;
     HANDLE hEvent;
     BOOL   bContinue;
     }
     PARAMS, *PPARAMS;

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "BigJob2";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);
     RegisterClassEx(&wndclass);
     hwnd = CreateWindow(szAppName, "Multithreading Demo",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

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

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

void Thread(PVOID pvoid)
     {
```

```
     double  A = 1.0;
     INT     i;
     LONG    lTime;
     PPARAMS pparams;

     pparams =(PPARAMS) pvoid;

     while(TRUE)
          {
          WaitForSingleObject(pparams->hEvent, INFINITE);

          lTime = GetCurrentTime();

          for(i = 0; i < REP && pparams->bContinue; i++)
               A = tan(atan(exp(log(sqrt(A * A))))) + 1.0;

          if(i == REP)
               {
               lTime = GetCurrentTime() - lTime;
               SendMessage(pparams->hwnd, WM_CALC_DONE, 0, lTime);
               }
          else
               SendMessage(pparams->hwnd, WM_CALC_ABORTED, 0, 0);
          }
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char  *szMessage[] = { "Ready(left mouse button begins)",
                                   "Working(right mouse button aborts)",
                                   "%d repetitions in %ld msec" };
     static HANDLE hEvent;
     static INT    iStatus;
     static LONG   lTime;
     static PARAMS params;
     char          szBuffer[64];
     HDC           hdc;
     PAINTSTRUCT   ps;
     RECT          rect;

     switch(iMsg)
          {
          case WM_CREATE :
               hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

               params.hwnd = hwnd;
               params.hEvent = hEvent;
               params.bContinue = FALSE;

               _beginthread(Thread, 0, &params);

               return 0;

          case WM_LBUTTONDOWN :
               if(iStatus == STATUS_WORKING)
                    {
                    MessageBeep(0);
                    return 0;
                    }

               iStatus = STATUS_WORKING;

               params.bContinue = TRUE;

               SetEvent(hEvent);
```

```
            InvalidateRect(hwnd, NULL, TRUE);
            return 0;

        case WM_RBUTTONDOWN :
            params.bContinue = FALSE;
            return 0;

        case WM_CALC_DONE :
            lTime = lParam;
            iStatus = STATUS_DONE;
            InvalidateRect(hwnd, NULL, TRUE);
            return 0;

        case WM_CALC_ABORTED :
            iStatus = STATUS_READY;
            InvalidateRect(hwnd, NULL, TRUE);
            return 0;
        case WM_PAINT :
            hdc = BeginPaint(hwnd, &ps);

            GetClientRect(hwnd, &rect);

            wsprintf(szBuffer, szMessage[iStatus], REP, lTime);

            DrawText(hdc, szBuffer, -1, &rect,
                     DT_SINGLELINE | DT_CENTER | DT_VCENTER);

            EndPaint(hwnd, &ps);
            return 0;

        case WM_DESTROY :
            _endthread();
            PostQuitMessage(0);
            return 0;
        }
    return DefWindowProc(hwnd, iMsg, wParam, lParam);
    }
```

Рис. 14.5 Программа BIGJOB2

Оконная процедура обрабатывает сообщение WM_CREATE, создавая событие с автоматическим сбросом и инициализирует его как занятое, а затем создает поток.

Функция *Thread* входит в бесконечный цикл *while*, но при этом вызывает функцию *WaitForSingleObject* в начале цикла (обратите внимание, что структура PARAMS имеет третье поле, содержащее описатель объекта событие). Поскольку событие изначально занято, поток задерживается в функции. Щелчок левой клавишей мыши заставляет оконную процедуру вызвать функцию *SetEvent*. Это освобождает второй поток, находящийся в ожидании внутри функции *WaitForSingleObject*, и он начинает выполнять "дикие" вычисления. После окончания очередного цикла поток снова вызывает функцию *WaitForSingleObject*, но при этом объект Событие уже оказывается в занятом состоянии с момента предыдущего вызова этой функции. Следовательно, поток задерживается до тех пор, пока не будет произведен следующий щелчок левой клавишей мыши.

В остальном программа BIGJOB2 является такой же, как и программа BIGJOB1.

Локальная память потока

Глобальные переменные в многопоточных программах (так же как и любая выделенная память) разделяются между всеми потоками в программе. Локальные статические переменные функций также разделяются между всеми потоками, использующими эту функцию. Локальные автоматические переменные в функции являются уникальными для каждого потока, потому что они хранятся в стеке, а каждый поток имеет свой собственный стек.

Может возникнуть необходимость иметь постоянную область памяти, уникальную для каждого потока. Например, функция *strtok* языка C, которая уже упоминалась в этой главе, требует такого типа память. Нет сомнений, что C его не поддерживает. В Windows 95 имеется четыре функции, поддерживающие эту память, которая называется локальной памятью потока (thread local storage, TLS).

Теперь рассмотрим, как она работает. Во-первых, определим структуру, содержащую все данные, которые должны быть уникальны для потоков, например:

```
typedef struct
{
       int a;
       int b;
} DATA, *PDATA;
```

Первичный поток вызывает функцию *TlsAlloc* для получения значения индекса:

```
dwTlsIndex = TlsAlloc();
```

Он может храниться в глобальной переменной или может быть передан функции потока в параметре-структуре.

Функция потока начинается с выделения памяти для структуры данных и с вызова функции *TlsSetValue*, используя индекс, полученный ранее:

```
TlsSetValue(dwTlsIndex, GlobalAlloc(GPTR, sizeof(DATA)));
```

Это действие устанавливает соответствие указателя с конкретным потоком и конкретным индексом в потоке. Теперь, любая функция, которой нужно использовать этот указатель (включая саму базовую функцию потока), может использовать код, подобный такому:

```
PDATA pdata;
[другие строки программы]
pdata =(PDATA) TlsGetValue(dwTlsIndex);
```

Теперь она может изменять значения *pdata->a* и *pdata->b*. Перед завершением функции потока необходимо освободить захваченную память:

```
GlobalFree(TlsGetValue(dwTlsIndex));
```

Когда все потоки, использующие эти данные будут завершены, первичный поток освобождает индекс:

```
TlsFree(dwTlsIndex);
```

Этот процесс может поначалу вас смутить, поэтому, может быть, было бы полезно посмотреть как организована локальная память потока. (Мне неизвестно, как в действительности Windows 95 это делает, но описываемая нами схема вполне правдоподобна.) Во-первых, функция *TlsAlloc* могла бы просто выделить блок памяти (длиной 0 байт) и вернуть значение индекса, который является указателем на этот блок. Каждый раз при вызове функции *TlsSetValue* с этим индексом блок памяти увеличивается на 8 байт с помощью функции *GlobalReAlloc*. В этих 8 байтах хранятся идентификатор потока, вызывающего функцию, полученный с помощью функции *GetCurrentThreadID*, и указатель, переданный функции *TlsSetValue*. Функция *TlsGetValue* просто использует идентификатор потока для поиска в таблице, а затем возвращает указатель. Функция *TlsFree* освобождает блок памяти.

Вероятно, это вы могли бы реализовать и сами. Но все-таки приятно иметь возможность использовать готовое.

Глава 15 Использование принтера

15

Когда в главах 3 и 4 для вывода текста и графики использовался экран дисплея, концепция независимости от устройства могла показаться совершенной и очень удобной, но насколько полно эта концепция поддерживается для принтеров и плоттеров? В Microsoft Windows 95 принтеры и плоттеры обеспечены независимым от устройства графическим интерфейсом. При программировании для принтера можно не думать о последовательности управляющих сигналов и протоколах связи с принтером. В продаваемых программах для Windows бросается в глаза отсутствие дисков со специализированными драйверами принтеров, что характерно для программ текстовых редакторов и графических программ в MS-DOS. Если в продаваемую программу для Windows включаются драйверы принтера, то это, как правило, касается модернизированных версий существующих драйверов.

Из программ для Windows можно печатать текст и графику с использованием тех же функций GDI, какие использовались для вывода на экран монитора. Многое из изученного в главах 3 и 4, что связано с концепцией независимости от устройства — в большей степени это касается размеров и разрешающей способности дисплея, а также его возможностей по воспроизведению цветов — вполне применимо и допустимо здесь. Тем не менее принтер или плоттер это не просто монитор, в котором вместо кинескопа используется бумага. Имеется несколько очень важных отличий. Например, мы никогда не думали о том, что монитор может быть не подключен к видеоадаптеру, но для принтеров аналогичная проблема вполне реальна: они могут быть либо неподключенными, либо у них может кончиться бумага.

Не нужно было беспокоиться о том, что видеоадаптер не сможет реализовать определенные графические операции. Видеоадаптер либо может управлять выводом графики, либо нет. Последнее означает, что он вообще не может использоваться под Windows. В то же время на некоторых принтерах нельзя печатать графику (тем не менее они все еще используются при работе с Windows), а на плоттеры можно выводить векторную графику, но с пересылкой битовых блоков возникают проблемы.

Имеются и другие особенности, требующие внимания:

- Принтеры медленнее мониторов. Хотя мы пытались добиться наилучшей производительности, мы не беспокоились о времени, необходимом для отображения информации на экране. Но никто не захочет ждать, пока принтер закончит печатать, чтобы возобновить работу.
- В программах, когда одни данные сменяются другими, поверхность экрана используется многократно. На принтере это невозможно. Вместо этого, заполненную страницу сменяет следующая.
- На экране монитора одновременно имеются окна, куда выводят данные различные приложения. Для печати из различных приложений на принтере необходимо, чтобы их вывод был разделен на отдельные документы или задания.

Для поддержки принтера в GDI в Windows 95 имеется несколько специальных функций, которые называют функциями печати. Функции для вывода на печать обычного текста и графики используются в программе точно также, как они используются для вывода информации на экран. Функции печати, такие как *StartDoc*, *EndDoc*, *StartPage* и *EndPage*, используются для упорядочивания выводимой информации и передачи ее на принтер.

Печать, буферизация и функции печати

Применение принтера в Windows 95 фактически включает в себя комплекс различных действий, связанных с использованием модуля библиотеки GDI32, модуля библиотеки драйвера принтера (файлы с расширением .DRV) и спулера печати Windows, а также некоторых других модулей. Перед тем, как начать программировать для принтера, рассмотрим, как все это работает.

Когда приложение собирается работать с принтером, оно в первую очередь, используя функции *CreateDC* и *PrintDLG*, получает описатель контекста принтера (printer device context). Это приводит к тому, что модуль библиотеки драйвера принтера загружается в оперативную память (если он к этому времени еще не загружен) и

инициализируется. Затем программа вызывает функцию *StartDoc*, которая сигнализирует о запуске в работу нового документа. Функция *StartDoc* обрабатывается в модуле GDI. Модуль GDI вызывает в драйвере принтера функцию *Control*, сообщая драйверу принтера о необходимости инициализировать данные и подготовиться к печати.

Вызов функции *StartDoc* начинает процесс печати документа; заканчивается этот процесс, когда программа вызывает функцию *EndDoc*. Эти две функции действуют как ограничители для обычных команд GDI, с помощью которых на страницах документа выводится текст и графика. Сама по себе каждая страница — это своего рода поддокумент. Для начала печати страницы вызывается функция *StartPage*, а для ее окончания функция *EndPage*.

Например, при необходимости изобразить на странице эллипс, первой для начала печати документа вызывается функция *StartDoc*, затем функция *StartPage*, которая сигнализирует о начале новой страницы. После этого программа вызывает функцию *Ellipse*, точно так же как при выводе эллипса на экран. Как правило модуль GDI сохраняет вызванные функции GDI в файле на диске, обычно — в метафайле расширенного формата, который располагается в подкаталоге, задаваемом в переменной окружения TEMP. (При отсутствии переменной TEMP, Windows использует корневой каталог первого жесткого диска системы.) Этот файл начинается с символов ~EMF (расширенный метафайл, enhanced metafile) и имеет расширение .TMP. Однако, как будет далее коротко рассказано, для драйвера принтера возможна ситуация, когда этот шаг можно опустить.

Когда приложения заканчивает формирование первой страницы (используя вызовы функций GDI), программа вызывает функцию *EndPage*. Теперь начинается реальная работа. Драйвер принтера должен преобразовать различные команды рисования, хранящиеся в метафайле, в выходные данные для принтера. Выходных данных для принтера, необходимых для изображения страницы графики, может быть очень много, особенно если в принтере для организации страниц не применяется язык высокого уровня. Например, для лазерного принтера с разрешающей способностью 600 точек на дюйм и страницы размером 8,5 на 11 дюймов для описания страницы графики необходимо более четырех мегабайт данных.

По этой причине драйверы принтера часто используют прием, называемый разбиением на полосы (banding), который заключается в том, что страница делится на прямоугольники, называемые полосами. (О разбиении на полосы будет рассказано далее в этой главе.) Модуль GDI получает размеры каждой из этих полос из драйвера принтера. Затем он устанавливает регион отсечения, равный этой полосе, и из драйвера принтера для каждой из функций рисования, хранящихся в расширенном метафайле, вызывает функцию *Output*. Этот процесс называется проигрыванием метафайла в драйвере устройства (playing the metafile into the device driver). Модуль GDI должен целиком проиграть расширенный метафайл в драйвере устройства для каждой полосы, которую драйвер устройства задает на странице. После завершения этого процесса, Windows удаляет расширенный метафайл.

Для каждой полосы драйвер устройства преобразует полученные из метафайла функции рисования в выходные данные, необходимые для реализации их на принтере. Формат этих данных зависит от принтера. Для матричных принтеров они представляют из себя набор управляющих сигналов, включая последовательность сигналов для воспроизведения графики. (Для определенного содействия в такой организации выходных данных в драйвере принтера могут использоваться различные вспомогательные функции, которые также находятся в модуле GDI.) Для лазерного принтера, в котором для организации страниц используется язык высокого уровня (например, PostScript), выходные данные для принтера будут представлять собой программу на этом языке.

Драйвер принтера для каждой полосы передает выходные данные для принтера в модуль GDI, в котором они сохраняются во временном файле, также расположенном в подкаталоге TEMP. Этот файл начинается с символов ~SPL и имеет расширение .TMP. Когда вся страница сформирована, модуль GDI вызывает спулер печати, указывая, что готова новая страница. Затем приложение выполняет тот же процесс для следующей страницы. После того, как это сделано для всех страниц, которые должны быть напечатаны, вызывается функция *EndDoc*, сигнализирующая о том, что задание на печать выполнено. На рис. 15.1 показано взаимодействие приложения, модуля GDI, драйвера принтера и спулера печати.

Спулер печати в Windows 95 фактически представляет собой набор следующих компонентов:

Компоненты спулера	**Описание**
Print Request Router (маршрутизатор запроса на печать)	Направляет поток данных поставщику печати
Local print provider (локальный поставщик печати)	Создает буферизуемые файлы, предназначенные для локального принтера
Network print provider (сетевой поставщик печати)	Создает буферизуемые файлы, предназначенные для сетевого принтера
Print processor (процессор печати)	Выполняет дебуферизацию, которая заключается в преобразовании буферизованных, независящих от устройства данных в форму, предназначенную для конкретного принтера
Port monitor (монитор порта)	Управляет портом, с которым соединен принтер
Language monitor (языковой монитор)	Управляет двусторонней связью с принтером для задания конфигурации устройства и контроля состояния принтера

Спулер освобождает приложения от множества действий, связанных с печатью. Windows загружает спулер печати при старте, поэтому, когда программа начинает процесс печати, спулер уже работает. Когда программа печатает документ, модуль GDI создает файлы, содержащие выходные данные для принтера. Задание на печать для спулера заключается в том, чтобы отправить на принтер эти файлы. Спулер уведомляется о новом задании на печать модулем GDI. Затем он начинает считывать файлы и переправлять их непосредственно на принтер. Для передачи файлов спулер использует различные функции, которые обеспечивают связь с параллельным или последовательным портом, к которому подключен принтер. Когда спулер отправляет файл на принтер, он удаляет временный файл, в котором хранятся выходные данные. Этот процесс показан на рис. 15.2.

Большая часть этого процесса прозрачна для приложения. С точки зрения программы, печать имеет место только на время, необходимое для того, чтобы модуль GDI сохранил все выходные данные для печати в файлах на диске. После этого, или даже раньше, если печать осуществляется в отдельном потоке, программа может заниматься другими делами.

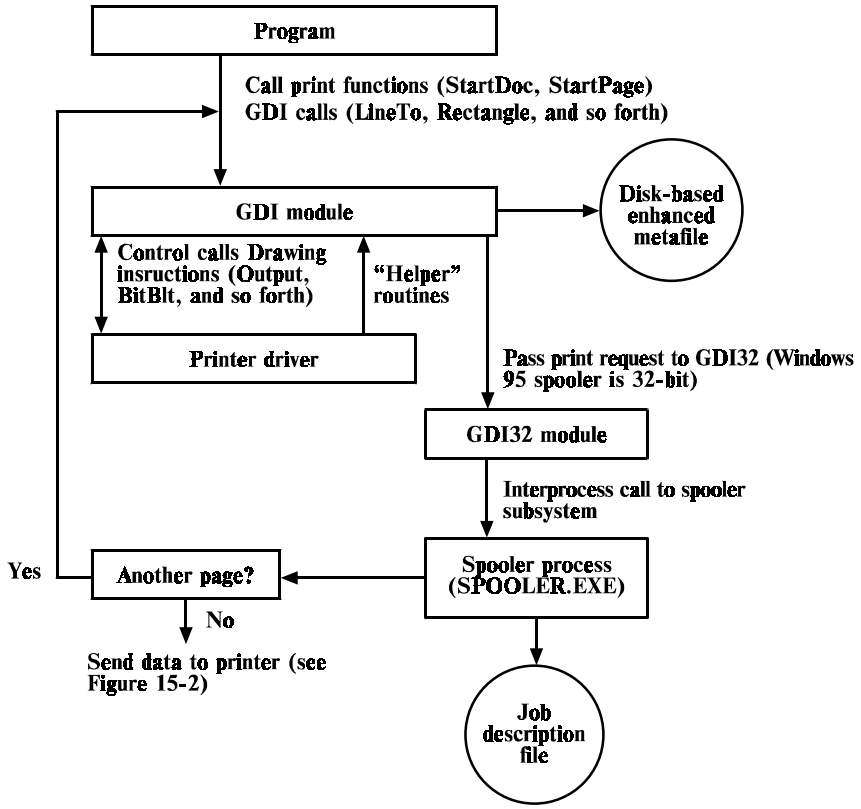


Рис. 15.1 Взаимодействие приложения, модуля GDI, драйвера принтера и спулера печати

Фактически, процесс печати вместо программы берет на себя спулер. Из папки Printers пользователь может приостанавливать выполнение заданий на печать, изменять их приоритеты, или отменять их. Эта схема дает программе возможность "печатать" быстрее, чем это было бы возможно, если бы задания на печать выводились в реальном времени, и надо было бы ждать, пока принтер, напечатав одну страницу, начнет процесс создания следующей.

Здесь были показаны только основы процесса печати, но у этой схемы имеются различные варианты. Один из них состоит в том, что спулер печати не обязательно должен быть запущен, чтобы программы для Windows могли использовать принтер. Пользователь может отключить буферизацию на странице Details набора страниц свойств принтера.

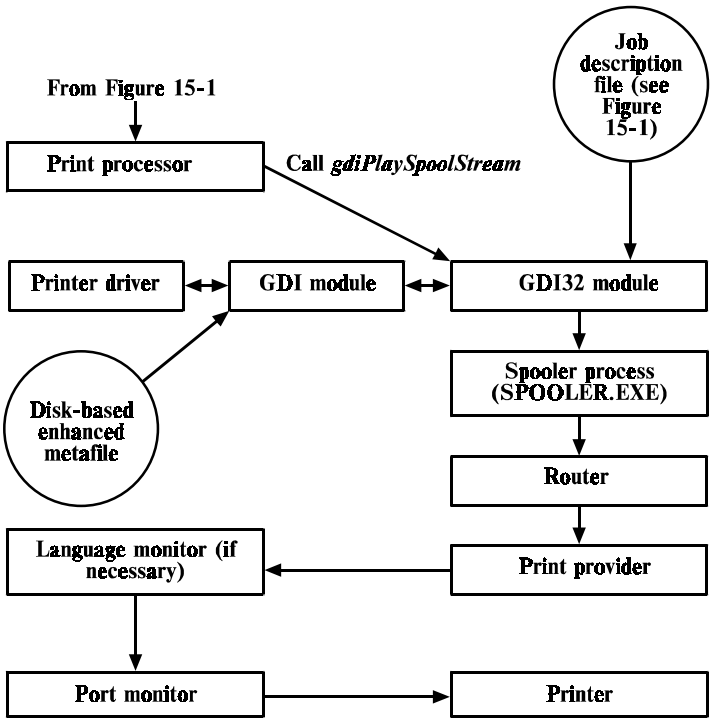


Рис. 15.2. Работа спулера печати

Зачем пользователю может понадобиться отключать спулер? Это может случиться, если у пользователя есть аппаратный или программный спулер печати, работающий быстрее, чем работает спулер Windows. Или если принтер подключен к сети, в которой имеется собственный спулер. Основное правило гласит, что один спулер работает быстрее двух. Поэтому удаление спулера Windows ускоряет печать, поскольку тогда нет нужды хранить на диске выходные данные для печати. Они направляются прямо на принтер и обрабатываются внешним аппаратным или программным спулером печати.

Если спулер Windows выключен, то модуль GDI не сохраняет в файле выходные данные, полученные из драйвера устройства. Вместо этого сам модуль GDI отправляет выходные данные прямо в параллельный или последовательный порт, к которому подключен принтер.

А вот другой вариант основной схемы печати. При нормальной работе модуль GDI хранит все функции, необходимые для определения страницы, в расширенном метафайле, а затем каждый раз, когда драйвер принтера задает очередную полосу, проигрывает этот метафайл в драйвере принтера. Однако, если драйвер принтера не требует разбиения страницы на полосы, расширенный метафайл не создается; модуль GDI просто передает функции рисования непосредственно в драйвер. Или еще один вариант, в нем предполагается, что выходные данные для принтера делятся на полосы в самом приложении. Это делает программу печати в приложении более сложной, но освобождает модуль GDI от создания метафайла. И снова модуль GDI просто передает в драйвер принтера функции рисования для каждой полосы.

Теперь, возможно, вы начинаете понимать, что Windows-программе для печати требуется несколько больше ресурсов, чем ей потребовалось бы при использовании монитора. При этом могут появиться проблемы, особенно, если модуль GDI при создании метафайла или файлов выходных данных принтера сталкивается с нехваткой дискового пространства. Пользователь может быть уведомлен о возникшей проблеме, и либо попытаться ее решить, либо остаться в стороне.

Далее рассмотрим несколько различных подходов к печати страниц. Но первое, с чего мы начнем, это получение контекста устройства.

Контекст принтера

Точно также, как перед выводом изображения на экран монитора необходимо получить описатель контекста устройства, перед началом печати необходимо получить описатель контекста принтера (printer device context). После получения этого описателя (и вызова функции *StartDoc*, чтобы объявить о намерении создать новый документ), его можно использовать точно таким же образом, каким использовался описатель контекста устройства монитора, а именно в качестве первого параметра различных функций GDI, как было показано в главах 3 и 4.

Получение описателя контекста принтера — дело совершенно обычное. В большинстве программ для этого просто вызывается функция *PrintDlg*. Помимо получения контекста устройства выбранного принтера, эта функция также дает возможность пользователю перед началом печати выбрать другой принтер или задать другие характеристики

печати. Функция *PrintDlg*, как будет показано в дальнейшем, может сэкономить массу затрат. Простейший путь начать печать на заданном по умолчанию принтере без всякого диалога реализован в программе NOTEPAD. Для реализации этого пути вам необходимо создать описатель контекста принтера с помощью функции *CreateDC*.

В главе 4 было показано, как используя эту функцию можно получить описатель контекста монитора:

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

Описатель контекста принтера можно получить с помощью этой же функции. Однако, для контекста принтера в Win32 игнорируются первый и третий параметры. Таким образом, синтаксис функции *CreateDC* для принтера следующий:

```
hdc = CreateDC(NULL, lpszDeviceName, NULL, lpInitializationData);
```

Параметр *lpInitializationData* обычно также устанавливается в NULL. Параметр *lpszDeviceName* — это указатель на символьную строку, в которой операционной системе Windows передается название принтера. Перед тем как указать название принтера, необходимо определить, какие принтеры имеются в системе.

Формирование параметров для функции *CreateDC*

В системе может быть более одного присоединенного принтера. В ней также могут иметься другие программы, такие как программы факсимильной связи, имитирующие принтер. Независимо от числа присоединенных принтеров, только один из них может рассматриваться как текущий принтер или как заданный по умолчанию принтер. Это последний принтер, который пользователь выбрал в окне диалога, появляющемся при выборе в меню опции Print, или при его назначении таковым в папке Printers. Многие небольшие программы для Windows ограничиваются использованием только этого принтера.

Windows 95 предлагает несколько методов для определения того, какие принтеры (или псевдопринтеры) присоединены к системе. Из них мы рассмотрим два метода: вызов функции *EnumPrinters* и поиск в файле WIN.INI.

Функция *EnumPrinters*

Вызов функции *EnumPrinters* — это простейший способ идентификации принтеров. Функция заполняет массив структур, в которых содержится информация о каждом подключенном принтере. Можно даже, в зависимости от желаемого уровня детализации, выбирать из трех типов структур. В структуре PRINTER_INFO_1 имеется информация не только о названии принтера. В структуре PRINTER_INFO_2 информации еще больше, например имя драйвера принтера, имя устройства печати и выходного порта. Структура PRINTER_INFO_5 предоставляет имя устройства печати, имя порта и значения тайм-аутов. Использование этой структуры дает дополнительное преимущество, поскольку заставляет функцию *EnumPrinters* не запрашивать открытия какого-либо устройства, а запрашивать реестр (registry), что приводит к ускорению работы. Для создания контекста принтера необходимо только его название, поэтому вполне достаточно структуры PRINTER_INFO_5.

Рассмотрим пример. Следующая функция вызывает функцию *EnumPrinters* и возвращает описатель контекста устройства для текущего принтера:

```
HDC GetPrinterDC(void)
{
       PRINTER_INFO_5 pinfo5[3];
       DWORD dwNeeded, dwReturned;

       if(EnumPrinters(PRINTER_ENUM_DEFAULT, NULL, 5,

              (LPBYTE) pinfo5, sizeof(pinfo5), &dwNeeded, &dwReturned))

              return CreateDC(NULL, pinfo5[0].pPrinterName, NULL, NULL);

       return 0;     //возврат нуля, если принтер не найден
}
```

Первый параметр функции *EnumPrinters* задает тип объектов принтера, которые необходимо получить. В данном случае используется флаг PRINTER_ENUM_DEFAULT для запроса информации только о текущем принтере. Третий параметр показывает тип заполняемой структуры: 1 для PRINTER_INFO_1, 2 для PRINTER_INFO_2 и 5 для PRINTER_INFO_5. Параметр *pinfo5* — это указатель на массив структур, первую из которых функция *EnumPrinters* заполняет названием текущего принтера.

Поскольку только один принтер может быть текущим, то может показаться странным, что задается массив из трех структур типа PRINTER_INFO_5. К сожалению функция *EnumPrinters* содержит ошибку. И хотя она правильно заполняет только первую структуру PRINTER_INFO_5 массива, для функции нужен буфер, содержащий более чем одну структуру. (Не верьте документации.) Увеличение размера буфера втрое, кажется, заставляет функцию *EnumPrinters* работать нормально.

Возвращаемым значением функции *GetPrinterDC* является 0, если контекст принтера создать не удалось. Это может произойти, если имя принтера определено неправильно, если Windows не может найти драйвер принтера, или если в качестве имени выходного порта получено "none" (означающее, что принтер не связан с каким-либо портом). Тем не менее, можно получить информацию и для несвязанного с портом принтера, если использовать не функцию *CreateDC*, а функцию *CreateIC*.

Пусть необходим список всех подключенных принтеров, а не только имя текущего. Для этого тоже годится функция *EnumPrinters*, но требуется чуть больше усилий. Задайте первый параметр равным PRINTER_ENUM_LOCAL. В Windows 95 этот флаг приводит к тому, что функция перечисляет все подключенные к системе принтеры, как непосредственно, так и через сеть.

Для перечисления неизвестного количества принтеров, функцию *EnumPrinters* сначала нужно использовать для определения требуемого размера буфера. Необходимый размер она возвращает в параметре *dwNeeded*, который затем и используется для выделения необходимого буфера:

```
LPPRINTER_INFO_5 ppinfo5;
DWORD dwNeeded, dwReturned;
BOOL bErr;

// во-первых, определите требуемую размерность массива
EnumPrinters(PRINTER_ENUM_LOCAL, NULL, 5, "", 0, &dwNeeded, &dwReturned);

// далее, для массива PRINTER_INFO_5 выделите необходимую память
ppinfo5 =(LPPRINTER_INFO_5) HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, dwNeeded);

// и наконец, для заполнения выделенного массива PRINTER_INFO_5
// снова вызовите функцию EnumPrinters
if(ppinfo5) bErr = EnumPrinters(
PRINTER_ENUM_LOCAL, NULL, 5,(LPBYTE) ppinfo5,
                                  dwNeeded, &dwNeeded, &dwReturned
                                );
```

В этом примере выделенный массив структур PRINTER_INFO_5 заполняется данными о всех принтерах, подключенных к системе, как локальных, так и удаленных. Первый вызов функции *EnumPrinters* делается чтобы получить в *dwNeeded* размер требуемого буфера. В следующих инструкциях для полученного размера буфера выделяется память, и снова вызывается функция *EnumPrinters*. На этот раз памяти достаточно. Фактическое число структур PRINTER_INFO_5 , которые заполняет функция *EnumPrinters*, находится в возвращаемом параметре *dwNeeded*. Если возвращаемым значением функции *EnumPrinters* является TRUE, то пользователю можно предложить список названий принтеров. (Далее это будет сделано в программе DEVCAPS2.)

Поиск в файле WIN.INI

Другой способ обнаружения подключенных к системе принтеров состоит в небольшом поиске в файле WIN.INI. Текущий принтер указан в секции [*windows*] файла WIN.INI с ключевым словом device. В приведенной далее строке содержатся имя устройства (необходимое для вызова функции *CreateDC*), имя драйвера и выходной порт:

```
[windows]
[другие строки]
device = IBM Graphics, IBMGRX, LPT1:
```

В данном случае именем устройства является IBM Graphics, именем драйвера — IBMGRX и именем выходного порта — LPT1.

Далее предлагается другой вариант написания функции *GetPrinterDC*, возвращающей описатель контекста принтера. В этом примере функция *GetProfileString* используется для получения из файла WIN.INI строки device, далее она разбивает строку на три части и для получения контекста текущего принтера вызывает функцию *CreateDC*. Хотя в Win32 функция *CreateDC* игнорирует имена драйвера и порта, в представленном ниже примере показано, как получить из возвращаемой функцией *GetProfileString* строки все необходимые имена:

```
HDC GetPrinterDC()
{
       char szPrinter[80];
       char *szDevice, *szDriver, *szOutput;

       GetProfileString("windows", "device", ",,,", szPrinter, 80);

       if( NULL !=(szDevice = strtok(szPrinter,  ",")) &&
     NULL !=(szDriver = strtok(NULL,     ", ")) &&
```

```
            NULL !=(szOutput = strtok(NULL,     ", ")) )
              return CreateDC(szDriver, szDevice, szOutput, NULL);

       return 0;
}
```

Функция *GetProfileString* ищет в секции [*windows*] файла WIN.INI ключевое слово device и копирует до 80 символов, следующих за знаком равенства, в *szPrinter*. (Если Windows не может найти в файле WIN.INI секцию [*windows*] или ключевое слово device, то получаемой строкой функции *GetProfileString* является ее третий параметр.) Строка разбивается с помощью обычной функции *strtok* языка C, которая разбивает строку на подстроки. Обратите внимание, что, поскольку внутри имени устройства могут иметь место пробелы, для обнаружения конца строки *szDevice* использовалась только запятая. Как запятая, так и пробел являются разделителями между именем драйвера и выходным портом, поэтому первые и последние пробелы из этих строк удаляются. (Использование функции *strtok* — это не слишком хороший метод разбиения строки, поскольку эта функция не распознает коды многобайтных символов, которые могут применяться в версиях Windows для дальневосточных стран. В таком случае можно использовать функцию *EnumPrinters* указанным выше способом или написать собственную версию функции *strtok*, в которой для перемещения по строке вызывалась бы функция *CharNext*.)

В секции [*windows*] файла WIN.INI указан только текущий принтер, а несколько принтеров могут быть перечислены в секции [*devices*] файла WIN.INI. Эта секция выглядит примерно следующим образом:

```
[devices]
IBM Graphics = IBMGRX, LPT1:
Generic / Text Only = TTY, output.prn
HP Plotter = HPPLOT, COM1:
Postscript Printer = PSCRIPT, COM2:
```

Слева от каждого из знаков равенства записывается имя устройства; справа первым идет имя драйвера, за которым следует имя выходного порта. Получение описателя контекста устройства с помощью одного из имен принтеров, заданных в секции [*devices*] файла WIN.INI, совершенно аналогично получению описателя контекста устройства с указанием имени принтера, заданного в секции [*windows*] файла WIN.INI, за исключением того, что последнее труднее, поскольку здесь может быть указано более одного устройства.

Для получения списка всех подключенных принтеров, заданных в секции [*devices*] файла WIN.INI, включая текущий принтер, заданный в секции [*windows*], используйте функцию *GetProfileString*, в которой второй параметр установлен в NULL:

```
static char szAllDevices[4096];
[другие строки программы]
GetProfileString("devices", NULL, "", szAllDevices, sizeof(szAllDevices));
```

После возврата из функции *GetProfileString* строка *szAllDevices* содержит список ключевых слов (имен устройств) из секции [*devices*]. Каждое ключевое слово оканчивается нулем, за исключением последнего, которое оканчивается двумя нулями. Например, для показанного выше списка *szAllDevices* содержала бы следующее (используя систему обозначений языка C):

```
IBM Graphics\0Generic / Text Only\0HP Plotter\0Postscript Printers\0\0
```

Предположим, что пользователь выбрал одно из этих устройств и что вы установили указатель *szDevice* на начало имени этого устройства в *szAllDevices*. Теперь, используя указатель *szDevice*, можно вызвать функцию *CreateDC* или *CreateIC*. Если кроме имени принтера необходимы имена драйвера и выходного порта, то с помощью повторного вызова функции *GetProfileString* нужно получить оставшуюся часть строки:

```
GetProfileString("devices", szDevice, "", szPrinter, 64);
```

Затем, для получения имен драйвера и выходного порта, используйте функцию *strtok* для разбиения строки *szPrinter*:

```
szDriver = strtok(szPrinter, ", ");
szOutput= strtok(NULL,     ", ");
```

Действительные выходные порты конкретной системы перечислены в секции [*ports*] файла WIN.INI. Для использования принтера нет необходимости доступа к этой секции. Можно предположить, что пользователь уже задал для принтера определенный порт, и тогда можно предположить, что пользователь должным образом определил и параметры связи для последовательных (COM) портов.

Секция [*ports*] файла WIN.INI часто выглядит примерно так:

```
[ports]
LPT1:=
LPT2:=
LPT3:=
```

```
COM1:=9600, n, 8, 1
COM2:=1200, n, 8, 1
output.prn=
```

Файл OUTPUT.PRN (или любой файл с расширением .PRN) может быть здесь указан для задания непосредственной печати в файл. Имя файла может использоваться в качестве выходного порта для принтера в секции [*windows*] и в секции [*devices*] файла WIN.INI.

Измененная программа DEVCAPS

В главе 4 в первой версии программы DEVCAPS1 на экран выводится только некоторая часть информации, полученная для монитора функцией *GetDeviceCaps*. В новой версии, представленной на рис. 15.3, на экран выводится больше информации как для монитора, так и для подключенных к системе принтеров.

DEVCAPS2.MAK

```
#-----------------------
# DEVCAPS2.MAK make file
#-----------------------

devcaps2.exe : devcaps2.obj devcaps2.res
     $(LINKER) $(GUIFLAGS) -OUT:devcaps2.exe devcaps2.obj \
          devcaps2.res $(GUILIBS) winspool.lib

devcaps2.obj : devcaps2.c devcaps2.h
     $(CC) $(CFLAGS) devcaps2.c

devcaps2.res : devcaps2.rc devcaps2.h
     $(RC) $(RCVARS) devcaps2.rc
```

DEVCAPS2.C

```
/*--------------------------------------------------------------------
   DEVCAPS2.C -- Displays Device Capability Information(Version 2)
                 (c) Charles Petzold, 1996
  --------------------------------------------------------------------*/

#include <windows.h>
#include <string.h>
#include <stdio.h>
#include "devcaps2.h"

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
void DoBasicInfo   (HDC, HDC, int, int);
void DoOtherInfo   (HDC, HDC, int, int);
void DoBitCodedCaps(HDC, HDC, int, int, int);
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "DevCaps2";
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

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

```
     RegisterClassEx(&wndclass);

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

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

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
     {
     static char   szDevice[32], szWindowText[64];
     static int    n, cxChar, cyChar,
                   nCurrentDevice = IDM_SCREEN,
                   nCurrentInfo   = IDM_BASIC;
     static DWORD  dwNeeded, dwReturned;
     static LPPRINTER_INFO_5 pinfo5;

     DWORD         i;
     HDC           hdc, hdcInfo;
     HMENU         hMenu;
     PAINTSTRUCT   ps;
     TEXTMETRIC    tm;
     HANDLE        hPrint;

     switch(msg)
          {
          case WM_CREATE :
               hdc = GetDC(hwnd);
               SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));
               GetTextMetrics(hdc, &tm);
               cxChar = tm.tmAveCharWidth;
               cyChar = tm.tmHeight + tm.tmExternalLeading;
               ReleaseDC(hwnd, hdc);

               lParam = 0;
                                                  // fall through
          case WM_WININICHANGE :
               if(lParam != 0 && lstrcmp((PSTR) lParam, "devices") != 0)
                    return 0;

               hMenu = GetSubMenu(GetMenu(hwnd), 0);

               while(GetMenuItemCount(hMenu) > 1)
                    DeleteMenu(hMenu, 1, MF_BYPOSITION);

               // Get a list of all local and remote printers
               //
               // First, find out how large an array we need; this
               //   call will fail, leaving the required size in dwNeeded
               EnumPrinters(PRINTER_ENUM_LOCAL,
                            NULL, 5,(LPBYTE) "", 0, &dwNeeded, &dwReturned);

               // Next, allocate space for PRINTER_INFO_5 array
               if(pinfo5)
```

```
               HeapFree(GetProcessHeap(), 0, pinfo5);
          pinfo5 =(LPPRINTER_INFO_5) HeapAlloc(GetProcessHeap(),
                                        HEAP_NO_SERIALIZE, dwNeeded);

          // Last, fill allocated PRINTER_INFO_5 array
          if(!pinfo5 || !EnumPrinters(PRINTER_ENUM_LOCAL,
                              NULL, 5,(LPBYTE) pinfo5, dwNeeded,
                              &dwNeeded, &dwReturned))
               {
               MessageBox(hwnd, "Could not enumerate printers!",
                                        NULL, MB_ICONSTOP);
               DestroyWindow(hwnd);
               return 0;
               }
          for(i = 0, n = IDM_SCREEN + 1; i < dwReturned; i++, n++)
               {
               AppendMenu(hMenu, n % 16 ? 0 : MF_MENUBARBREAK, n,
                                        pinfo5->pPrinterName);
               pinfo5++;
               }

          AppendMenu(hMenu, MF_SEPARATOR, 0, NULL);
          AppendMenu(hMenu, 0, IDM_DEVMODE, "Properties");

          wParam = IDM_SCREEN;
                                        // fall through
     case WM_COMMAND :
          hMenu = GetMenu(hwnd);

          if(wParam < IDM_DEVMODE)          // IDM_SCREEN & Printers
               {
               CheckMenuItem(hMenu, nCurrentDevice, MF_UNCHECKED);
               nCurrentDevice = wParam;
               CheckMenuItem(hMenu, nCurrentDevice, MF_CHECKED);
               }
          else if(wParam == IDM_DEVMODE)    // "Properties" selection
               {
               GetMenuString(hMenu, nCurrentDevice, szDevice,
                              sizeof(szDevice), MF_BYCOMMAND);

               if(OpenPrinter(szDevice, &hPrint, NULL))
                    {
                    PrinterProperties(hwnd, hPrint);
                    ClosePrinter(hPrint);
                    }
               }
          else                               // info menu items
               {
               CheckMenuItem(hMenu, nCurrentInfo, MF_UNCHECKED);
               nCurrentInfo = wParam;
               CheckMenuItem(hMenu, nCurrentInfo, MF_CHECKED);
               }
          InvalidateRect(hwnd, NULL, TRUE);
          return 0;

     case WM_INITMENUPOPUP :
          if(lParam == 0)
               EnableMenuItem(GetMenu(hwnd), IDM_DEVMODE,
                    nCurrentDevice == IDM_SCREEN ?
                         MF_GRAYED : MF_ENABLED);
          return 0;

     case WM_PAINT :
          strcpy(szWindowText, "Device Capabilities: ");
                    if(nCurrentDevice == IDM_SCREEN)
```

```
                    {
                    strcpy(szDevice, "DISPLAY");
                    hdcInfo = CreateIC(szDevice, NULL, NULL, NULL);
                    }
               else
                    {
                    hMenu = GetMenu(hwnd);
                    GetMenuString(hMenu, nCurrentDevice, szDevice,
                                  sizeof(szDevice), MF_BYCOMMAND);
                    hdcInfo = CreateIC(NULL, szDevice, NULL, NULL);
                    }

               strcat(szWindowText, szDevice);
               SetWindowText(hwnd, szWindowText);

               hdc = BeginPaint(hwnd, &ps);
               SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));

               if(hdcInfo)
                    {
                    switch(nCurrentInfo)
                         {
                         case IDM_BASIC :
                              DoBasicInfo(hdc, hdcInfo, cxChar, cyChar);
                              break;

                         case IDM_OTHER :
                              DoOtherInfo(hdc, hdcInfo, cxChar, cyChar);
                              break;

                         case IDM_CURVE :
                         case IDM_LINE :
                         case IDM_POLY :
                         case IDM_TEXT :
                              DoBitCodedCaps(hdc, hdcInfo, cxChar, cyChar,
                                   nCurrentInfo - IDM_CURVE);
                              break;
                         }
                    DeleteDC(hdcInfo);
                    }

               EndPaint(hwnd, &ps);
               return 0;

          case WM_DESTROY :
               if(pinfo5)
                    HeapFree(GetProcessHeap(), 0, pinfo5);

               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, msg, wParam, lParam);
     }

void DoBasicInfo(HDC hdc, HDC hdcInfo, int cxChar, int cyChar)
     {
     static struct
          {
          int   nIndex;
          char *szDesc;
          }
          info[] =
          {
          HORZSIZE,      "HORZSIZE     Width in millimeters:",
          VERTSIZE,      "VERTSIZE     Height in millimeters:",
```

```
          HORZRES,       "HORZRES      Width in pixels:",
          VERTRES,       "VERTRES      Height in raster lines:",
          BITSPIXEL,     "BITSPIXEL    Color bits per pixel:",
          PLANES,        "PLANES       Number of color planes:",
          NUMBRUSHES,    "NUMBRUSHES   Number of device brushes:",
          NUMPENS,       "NUMPENS      Number of device pens:",
          NUMMARKERS,    "NUMMARKERS   Number of device markers:",
          NUMFONTS,      "NUMFONTS     Number of device fonts:",
          NUMCOLORS,     "NUMCOLORS    Number of device colors:",
          PDEVICESIZE,   "PDEVICESIZE  Size of device structure:",
          ASPECTX,       "ASPECTX      Relative width of pixel:",
          ASPECTY,       "ASPECTY      Relative height of pixel:",
          ASPECTXY,      "ASPECTXY     Relative diagonal of pixel:",
          LOGPIXELSX,    "LOGPIXELSX   Horizontal dots per inch:",
          LOGPIXELSY,    "LOGPIXELSY   Vertical dots per inch:",
          SIZEPALETTE,   "SIZEPALETTE  Number of palette entries:",
          NUMRESERVED,   "NUMRESERVED  Reserved palette entries:",
          COLORRES,      "COLORRES     Actual color resolution:"
          };
     char   szBuffer[80];
     int    i;

     for(i = 0; i < sizeof(info) / sizeof(info[0]); i++)
          TextOut(hdc, cxChar,(i + 1) * cyChar, szBuffer,
               sprintf(szBuffer, "%-40s%8d", info[i].szDesc,
                    GetDeviceCaps(hdcInfo, info[i].nIndex)));
     }

void DoOtherInfo(HDC hdc, HDC hdcInfo, int cxChar, int cyChar)
     {
     static BITS clip[] =
          {
          CP_RECTANGLE,  "CP_RECTANGLE",     "Can Clip To Rectangle:"
          };
     static BITS raster[] =
          {
          RC_BITBLT,       "RC_BITBLT",       "Capable of simple BitBlt:",
          RC_BANDING,      "RC_BANDING",      "Requires banding support:",
          RC_SCALING,      "RC_SCALING",      "Requires scaling support:",
          RC_BITMAP64,     "RC_BITMAP64",     "Supports bitmaps >64K:",
          RC_GDI20_OUTPUT, "RC_GDI20_OUTPUT", "Has 2.0 output calls:",
          RC_DI_BITMAP,    "RC_DI_BITMAP",    "Supports DIB to memory:",
          RC_PALETTE,      "RC_PALETTE",      "Supports a palette:",
          RC_DIBTODEV,     "RC_DIBTODEV",     "Supports bitmap conversion:",
          RC_BIGFONT,      "RC_BIGFONT",      "Supports fonts >64K:",
          RC_STRETCHBLT,   "RC_STRETCHBLT",   "Supports StretchBlt:",
          RC_FLOODFILL,    "RC_FLOODFILL",    "Supports FloodFill:",
          RC_STRETCHDIB,   "RC_STRETCHDIB",   "Supports StretchDIBits:"
          };

     static char *szTech[] = { "DT_PLOTTER(Vector plotter)",
                               "DT_RASDISPLAY(Raster display)",
                               "DT_RASPRINTER(Raster printer)",
                               "DT_RASCAMERA(Raster camera)",
                               "DT_CHARSTREAM(Character-stream, PLP)",
                               "DT_METAFILE(Metafile, VDM)",
                               "DT_DISPFILE(Display-file)" };
     char        szBuffer[80];
     int         i;

     TextOut(hdc, cxChar, cyChar, szBuffer,
          sprintf(szBuffer, "%-24s%04XH",
               "DRIVERVERSION:", GetDeviceCaps(hdcInfo, DRIVERVERSION)));

     TextOut(hdc, cxChar, 2 * cyChar, szBuffer,
```

```
          sprintf(szBuffer, "%-24s%-40s", "TECHNOLOGY:",
                          szTech[GetDeviceCaps(hdcInfo, TECHNOLOGY)]));

     TextOut(hdc, cxChar, 4 * cyChar, szBuffer,
          sprintf(szBuffer, "CLIPCAPS(Clipping capabilities)"));

     for(i = 0; i < sizeof(clip) / sizeof(clip[0]); i++)
          TextOut(hdc, 9 * cxChar,(i + 6) * cyChar, szBuffer,
               sprintf(szBuffer, "%-16s%-28s %3s",
                    clip[i].szMask, clip[i].szDesc,
                    GetDeviceCaps(hdcInfo, CLIPCAPS) & clip[i].nMask ?
                         "Yes" : "No"));

     TextOut(hdc, cxChar, 8 * cyChar, szBuffer,
          sprintf(szBuffer, "RASTERCAPS(Raster capabilities)"));

     for(i = 0; i < sizeof(raster) / sizeof(raster[0]); i++)
          TextOut(hdc, 9 * cxChar,(i + 10) * cyChar, szBuffer,
               sprintf(szBuffer, "%-16s%-28s %3s",
                    raster[i].szMask, raster[i].szDesc,
                    GetDeviceCaps(hdcInfo, RASTERCAPS) & raster[i].nMask ?
                         "Yes" : "No"));
     }

void DoBitCodedCaps(HDC hdc, HDC hdcInfo, int cxChar, int cyChar, int nType)
     {
     static BITS curves[] =
          {
          CC_CIRCLES,    "CC_CIRCLES",    "circles:",
          CC_PIE,        "CC_PIE",        "pie wedges:",
          CC_CHORD,      "CC_CHORD",      "chord arcs:",
          CC_ELLIPSES,   "CC_ELLIPSES",   "ellipses:",
          CC_WIDE,       "CC_WIDE",       "wide borders:",
          CC_STYLED,     "CC_STYLED",     "styled borders:",
          CC_WIDESTYLED, "CC_WIDESTYLED", "wide and styled borders:",
          CC_INTERIORS,  "CC_INTERIORS",  "interiors:"
          };

     static BITS lines[] =
          {
          LC_POLYLINE,   "LC_POLYLINE",   "polyline:",
          LC_MARKER,     "LC_MARKER",     "markers:",
          LC_POLYMARKER, "LC_POLYMARKER", "polymarkers",
          LC_WIDE,       "LC_WIDE",       "wide lines:",
          LC_STYLED,     "LC_STYLED",     "styled lines:",
          LC_WIDESTYLED, "LC_WIDESTYLED", "wide and styled lines:",
          LC_INTERIORS,  "LC_INTERIORS",  "interiors:"
          };

     static BITS poly[] =
          {
          PC_POLYGON,    "PC_POLYGON",    "alternate fill polygon:",
          PC_RECTANGLE,  "PC_RECTANGLE",  "rectangle:",
          PC_WINDPOLYGON,"PC_WINDPOLYGON","winding number fill polygon:",
          PC_SCANLINE,   "PC_SCANLINE",   "scanlines:",
          PC_WIDE,       "PC_WIDE",       "wide borders:",
          PC_STYLED,     "PC_STYLED",     "styled borders:",
          PC_WIDESTYLED, "PC_WIDESTYLED", "wide and styled borders:",
          PC_INTERIORS,  "PC_INTERIORS",  "interiors:"
          };

     static BITS text[] =
          {
          TC_OP_CHARACTER, "TC_OP_CHARACTER", "character output precision:",
          TC_OP_STROKE,    "TC_OP_STROKE",    "stroke output precision:",
```

```
          TC_CP_STROKE,    "TC_CP_STROKE",    "stroke clip precision:",
          TC_CR_90,        "TC_CP_90",        "90 degree character rotation:",
          TC_CR_ANY,       "TC_CR_ANY",       "any character rotation:",
          TC_SF_X_YINDEP,  "TC_SF_X_YINDEP",  "scaling independent of X and Y:",
          TC_SA_DOUBLE,    "TC_SA_DOUBLE",    "doubled character for scaling:",
          TC_SA_INTEGER,   "TC_SA_INTEGER",   "integer multiples for scaling:",
          TC_SA_CONTIN,    "TC_SA_CONTIN",    "any multiples for exact scaling:",
          TC_EA_DOUBLE,    "TC_EA_DOUBLE",    "double weight characters:",
          TC_IA_ABLE,      "TC_IA_ABLE",      "italicizing:",
          TC_UA_ABLE,      "TC_UA_ABLE",      "underlining:",
          TC_SO_ABLE,      "TC_SO_ABLE",      "strikeouts:",
          TC_RA_ABLE,      "TC_RA_ABLE",      "raster fonts:",
          TC_VA_ABLE,      "TC_VA_ABLE",      "vector fonts:"
          };

     static struct
          {
          int   nIndex;
          char  *szTitle;
          BITS (*pbits)[];
          short nSize;
          }
          bitinfo[] =
          {
          CURVECAPS,  "CURVCAPS(Curve Capabilities)",
                      (BITS(*)[]) curves, sizeof(curves) / sizeof(curves[0]),
          LINECAPS,   "LINECAPS(Line Capabilities)",
                      (BITS(*)[]) lines, sizeof(lines) / sizeof(lines[0]),
          POLYGONALCAPS, "POLYGONALCAPS(Polygonal Capabilities)",
                      (BITS(*)[]) poly, sizeof(poly) / sizeof(poly[0]),
          TEXTCAPS,   "TEXTCAPS(Text Capabilities)",
                      (BITS(*)[]) text, sizeof(text) / sizeof(text[0])
          };

     static char szBuffer[80];
     BITS       (*pbits)[] = bitinfo[nType].pbits;
     int         nDevCaps = GetDeviceCaps(hdcInfo, bitinfo[nType].nIndex);
     int         i;

     TextOut(hdc, cxChar, cyChar, bitinfo[nType].szTitle,
                    strlen(bitinfo[nType].szTitle));

     for(i = 0; i < bitinfo[nType].nSize; i++)
          TextOut(hdc, cxChar,(i + 3) * cyChar, szBuffer,
               sprintf(szBuffer, "%-16s %s %-32s %3s",
                   (*pbits)[i].szMask, "Can do",(*pbits)[i].szDesc,
                    nDevCaps &(*pbits)[i].nMask ? "Yes" : "No"));
     }
```

DEVCAPS2.H

```
/*------------------------
   DEVCAPS2.H header file
  ------------------------*/

#define IDM_SCREEN  1
#define IDM_DEVMODE 0x100

#define IDM_BASIC   0x101
#define IDM_OTHER   0x102
#define IDM_CURVE   0x103
#define IDM_LINE    0x104
#define IDM_POLY    0x105
#define IDM_TEXT    0x106

typedef struct
```

```
     {
     short  nMask;
     char  *szMask;
     char  *szDesc;
     }
     BITS;
```

DEVCAPS2.RC

```
/*----------------------------
   DEVCAPS2.RC resource script
  ----------------------------*/

#include "devcaps2.h"

DevCaps2 MENU
     {
     POPUP "&Device"
          {
          MENUITEM "&Screen",                    IDM_SCREEN, CHECKED
          }
     POPUP "&Capabilities"
          {
          MENUITEM "&Basic Information",         IDM_BASIC, CHECKED
          MENUITEM "&Other Information",         IDM_OTHER
          MENUITEM "&Curve Capabilities",        IDM_CURVE
          MENUITEM "&Line Capabilities",         IDM_LINE
          MENUITEM "&Polygonal Capabilities",    IDM_POLY
          MENUITEM "&Text Capabilities",         IDM_TEXT
          }
     }
```

Рис. 15.3 Программа DEVCAPS2

Поскольку программа DEVCAPS2 получает информационный контекст принтера, вы можете выбирать принтеры из меню программы DEVCAPS2, хотя каждый из принтеров в качестве выходного порта может иметь "none". При необходимости сравнить возможности разных принтеров вы можете использовать папку Printers для добавления различных драйверов принтеров.

Вызов функции *PrinterProperties*

В меню Device программы DEVCAPS2 включена опция Properties (свойства). Для ее использования нужно сначала выбрать принтер из меню Device. При выборе опции Properties появляется всплывающее окно диалога. Откуда берется это окно диалога? Оно вызывается драйвером принтера, который требует, как минимум, задать размер листа. В большинстве драйверов принтера также предлагается выбор режима печати: portrait или landscape. Режим portrait (часто задаваемый по умолчанию) подразумевает, что узкая сторона листа находится сверху; а в режиме landscape вверху находится широкая сторона листа. При изменении режима, оно отражается в информации, которую программа DEVCAPS2 получает от функции *GetDeviceCaps*: горизонтальные размер и разрешающая способность меняются на вертикальные размер и разрешающую способность. Диалоговые окна свойств цветных плоттеров могут быть достаточно громоздкими, в которых задаются цвета установленных в плоттере перьев и тип используемой бумаги.

В Windows 95 во всех драйверах принтера имеется экспортируемая функция *ExtDeviceMode*, которая вызывает окно диалога и сохраняет заданную пользователем информацию. Некоторые драйверы принтера сохранят полученную информацию в реестре или в их собственной секции файла WIN.INI. Те, которые сохраняют эту информацию, получают к ней доступ и в следующем сеансе работы Windows.

В программах для Windows, в которых у пользователя есть возможность выбора принтера, для этого обычно применяется вызов функции *PrintDlg*. Эта полезная функция берет на себя всю заботу о взаимодействии с пользователем и обрабатывает все внесенные пользователем при подготовке к печати изменения. Кроме этого, функция *PrintDlg* вызывает диалоговое окно свойств, когда пользователь нажимает кнопку *Properties*. Попытайтесь проделать это в программе WORDPAD. Вы увидите то же самое окно диалога, которое появляется в программе DEVCAPS2.

Программа может также вывести на экран окно диалога свойств принтера напрямую, вызывая из драйвера принтера функции *ExtDeviceMode* или *ExtDeviceModePropSheet*. Однако, это не рекомендуется. Гораздо лучше вызывать окна диалога косвенно, с помощью функции *PrinterProperties*. Эта функция GDI может обеспечить

доступ к 16-разрядным драйверам принтера, в то время как ваша 32-разрядная программа не имеет к ним доступа (до тех пор, пока вы не напишете свой код, позволяющий это сделать, thunk).

Для функции *PrinterProperties* необходим описатель принтера, который получают с помощью вызова функции *OpenPrinter*. Когда пользователь закрывает окно свойств, функция *PrinterProperties* возвращает управление. Затем, полученный описатель принтера можно закрыть с помощью функции *ClosePrinter*. Так делается в программе DEVCAPS2.

Сначала программа получает имя выбранного в данный момент в меню Device принтера и сохраняет его в символьном массиве *szDevice*:

```
GetMenuString(hMenu, nCurrentDevice, szDevice, sizeof(szDevice), MF_BYCOMMAND);
```

Затем, с помощью функции *OpenPrinter*, она получает описатель этого устройства. Если вызов функции прошел удачно, то для вывода на экран окна диалога программа вызывает функцию *PrinterProperties*, а затем для закрытия описателя устройства функцию *ClosePrinter*:

```
if( OpenPrinter(szDevice, &hPrint, NULL) )
{
       PrinterProperties(hwnd, hPrint);
       ClosePrinter(hPrint);
}
```

Проверка возможности работы с битовыми блоками (*BitBlt*)

Для получения размера и разрешающей способности области печати на странице можно использовать функции *GetDeviceCaps* или *DeviceCapabilities*. (В большинстве случаев область печати не соответствует размеру полного листа.) При желании самостоятельно масштабировать выводимую информацию, эти функции можно также использовать для получения относительной ширины и высоты области печати в пикселях.

Другую полезную характеристику принтера можно извлечь из бита RC_BITBLT возвращаемого значения функции *GetDeviceCaps* с параметром RASTERCAPS (растровые возможности, raster capabilities). Этот бит показывает способность устройства к передаче битовых блоков (bit block transfer). Большинство матричных и лазерных принтеров обладают таким свойством, а большинство плоттеров — нет. Устройства, которые не могут управлять передачей битовых блоков не поддерживают следующие функции GDI: *CreateCompatibleDC, CreateCompatibleBitmap, PatBlt, BitBlt, StretchBlt, GrayString, DrawIcon, SetPixel, GetPixel, FloodFill, ExtFloodFill, FillRgn, FrameRgn, InvertRgn, PaintRgn, FillRect, FrameRect* и *InvertRect*. Это единственное наиболее серьезное отличие между использованием функций GDI для дисплеев и их использованием для принтеров.

Программа FORMFEED

Теперь мы готовы к печати и начнем с наиболее простого. Фактически наша первая программа для печати ничего не делает, она только приводит к тому, что принтер прогоняет страницу. В программе FORMFEED, представленной на рис. 15.4, продемонстрирован абсолютный минимум требований к печати.

FORMFEED.MAK

```
#------------------------
# FORMFEED.MAK make file
#------------------------

formfeed.exe : formfeed.obj
     $(LINKER) $(GUIFLAGS) -OUT:formfeed.exe formfeed.obj \
     $(GUILIBS) winspool.lib

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

FORMFEED.C

```
/*--------------------------------------------
   FORMFEED.C -- Advances printer to next page
                 (c) Charles Petzold, 1996
  --------------------------------------------*/

#include <windows.h>

HDC  GetPrinterDC(void);
```

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpszCmdLine, int iCmdShow)
     {
     static DOCINFO di       = { sizeof(DOCINFO), "FormFeed", NULL };
     HDC            hdcPrint = GetPrinterDC();

     if(hdcPrint != NULL)
          {
          if(StartDoc(hdcPrint, &di) > 0)
               if(StartPage(hdcPrint) > 0 && EndPage(hdcPrint) > 0)
                    EndDoc(hdcPrint);

          DeleteDC(hdcPrint);
          }
     return FALSE;
     }

HDC GetPrinterDC(void)
     {
     PRINTER_INFO_5 pinfo5[3];
     DWORD          dwNeeded, dwReturned;

     if(EnumPrinters(PRINTER_ENUM_DEFAULT, NULL, 5,(LPBYTE) pinfo5,
                     sizeof(pinfo5), &dwNeeded, &dwReturned))
          return CreateDC(NULL, pinfo5[0].pPrinterName, NULL, NULL);

     return 0;            // EnumPrinters failed, so return null hdc
     }
```

Рис. 15.4 Программа FORMFEED

Программа FORMFEED включает приведенную ранее функцию *GetPrinterDC*. Кроме получения контекста принтера (и его дальнейшего удаления) программа вызывает только четыре функции печати, о которых было рассказано ранее в этой главе. Первой в программе FORMFEED вызывается функция *StartDoc* для запуска в работу нового документа. В программе проверяется возвращаемое значение этой функции, и работа продолжается только в том случае, если это значение положительно:

```
if(StartDoc(hdcPrint, &di) > 0)
```

Второе поле структуры *di* типа DOCINFO указывает на строку, идентифицирующую печатаемый документ. Пока документ печатается, или готовится к печати, эта строка отображается в столбце Document Name очереди на печать принтера. (Чтобы увидеть очередь документов, подготовленных к печати, щелкните на кнопке Start, выберите Settings, потом Printers и дважды щелкните на значке принтера.) Как правило строка, идентифицирующая документ, включает в себя имя приложения, выполняющего печать, и имя файла, который будет напечатан. В нашем случае это просто имя "FormFeed".

Если вызов функции *StartDoc* прошел удачно (о чем говорит ее положительное возвращаемое значение), то программа FORMFEED вызывает функцию *StartPage*, за которой сразу следует вызов функции *EndPage*. Эта последовательность заставляет принтер перейти на новую страницу. И вновь возвращаемые значения функций проверяются:

```
if(StartPage(hdcPrint) > 0 && EndPage(hdcPrint) > 0)
```

И наконец, если до сих пор ошибок не произошло, печать документа заканчивается:

```
EndDoc(hdcPrint);
```

Обратите внимание, что функция *EndDoc* вызывается только в том случае, если не было обнаружено ни одной ошибки. Если одна из других функций печати возвратит ошибку, то GDI сразу же прекращает вывод документа. Если принтер в этот момент не печатает, то такая ошибка часто приводит к сбросу принтера в исходное состояние.

Элементарная проверка возвращаемых значений функций печати — это простейший способ контроля ошибок. При необходимости сообщить пользователю о конкретной ошибке, нужно вызвать функцию *GetLastError*, которая определяет случившуюся ошибку. (Далее в этой главе мы рассмотрим сообщения об ошибках печати, возвращаемые этой функцией.) Например, ошибка имеет место, если GDI не в состоянии найти дисковое пространство, достаточное для хранения выходных данных принтера, необходимых для прогона одной страницы. Для большинства принтеров это случается чрезвычайно редко. Ради развлечения, однако, можно попытаться задать в качестве текущего принтера драйвер PostScript с выходным портом OUTPUT.PRN. Запустите теперь программу FORMFEED и проверьте размер файла. (Он будет более 8 килобайт!)

Если вы уже писали простые программы для прогона бумаги в MS-DOS, то знаете, что для большинства принтеров ASCII-кодом для прогона бумаги является число 12. Почему бы просто не открыть порт принтера с помощью функции GDI *OpenPrinter* и затем вывести в качестве выходных данных ASCII-код 12 с помощью функции *write*? Конечно, ничто не мешает нам сделать это. Если необходимо определить параллельный или последовательный порт, к которому подключен принтер, то следует использовать функцию *GetPrinter* с идентификатором принтера, полученным в качестве возвращаемого значения функции *OpenPrinter*, и считать идентификатор порта из структуры PRINTER_INFO_5. Затем необходимо определить, не используется ли принтер в данный момент другой программой. (Чтобы прогон бумаги не случился при печати какого-нибудь документа.) И наконец, необходимо определить, действительно ли ASCII-код 12 является управляющим символом прогона бумаги в подключенном принтере. Он является таковым, но не для всех принтеров. Например, команда прогона бумаги в PostScript равна не 12, а ключевому слову *showpage*.

Короче говоря, не пытайтесь даже думать о том, чтобы пойти в обход Windows; лучше разберитесь с функциями Windows для печати.

Печать графики и текста

Печать из программы для Windows обычно представляет собой большее число действий, чем это показано в программе FORMFEED. Давайте напишем программу, которая печатает одну страницу текста и графики. Начнем также, как в программе FORMFEED, а затем кое-что добавим. Мы рассмотрим четыре версии этой программы PRINT1, PRINT2, PRINT3 и PRINT4. Чтобы избежать бесполезного дублирования, в каждой из этих программ будут использоваться функции из файла PRINT.C, который представлен на рис. 15.5.

PRINT.C

```
/*--------------------------------------------------------------------
   PRINT.C -- Common routines for Print1, Print2, Print3, and Print4
  --------------------------------------------------------------------*/

#include <windows.h>

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

extern HINSTANCE hInst;
extern char      szAppName[];
extern char      szCaption[];

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
     {
     HWND       hwnd;
     MSG        msg;
     WNDCLASSEX wndclass;

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

     RegisterClassEx(&wndclass);

     hInst = hInstance;

     hwnd = CreateWindow(szAppName, szCaption,
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
```

```
                         NULL, NULL, hInstance, NULL);

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

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

HDC GetPrinterDC(void)
     {
     PRINTER_INFO_5 pinfo5[3];
     DWORD          dwNeeded, dwReturned;

     if(EnumPrinters(PRINTER_ENUM_DEFAULT, NULL, 5,(LPBYTE) pinfo5,
                      sizeof(pinfo5), &dwNeeded, &dwReturned))
          return CreateDC(NULL, pinfo5[0].pPrinterName, NULL, NULL);

     return 0;            // EnumPrinters failed, so return null hdc
     }
void PageGDICalls(HDC hdcPrn, int cxPage, int cyPage)
     {
     static char szTextStr[] = "Hello, Printer!";

     Rectangle(hdcPrn, 0, 0, cxPage, cyPage);

     MoveToEx(hdcPrn, 0, 0, NULL);
     LineTo  (hdcPrn, cxPage, cyPage);
     MoveToEx(hdcPrn, cxPage, 0, NULL);
     LineTo  (hdcPrn, 0, cyPage);

     SaveDC(hdcPrn);

     SetMapMode      (hdcPrn, MM_ISOTROPIC);
     SetWindowExtEx  (hdcPrn, 1000, 1000, NULL);
     SetViewportExtEx(hdcPrn, cxPage / 2, -cyPage / 2, NULL);
     SetViewportOrgEx(hdcPrn, cxPage / 2,  cyPage / 2, NULL);

     Ellipse(hdcPrn, -500, 500, 500, -500);

     SetTextAlign(hdcPrn, TA_BASELINE | TA_CENTER);

     TextOut(hdcPrn, 0, 0, szTextStr, sizeof(szTextStr) - 1);

     RestoreDC(hdcPrn, -1);
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
     {
     static int   cxClient, cyClient;
     HDC          hdc;
     HMENU        hMenu;
     PAINTSTRUCT  ps;

     switch(msg)
          {
          case WM_CREATE :
               hMenu = GetSystemMenu(hwnd, FALSE);
               AppendMenu(hMenu, MF_SEPARATOR, 0, NULL);
               AppendMenu(hMenu, 0, 1, "&Print");
               return 0;
```

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

        case WM_SYSCOMMAND :
             if(wParam == 1)
                  {
                  if(PrintMyPage(hwnd))
                       MessageBox(hwnd, "Could not print page!",
                            szAppName, MB_OK | MB_ICONEXCLAMATION);
                  return 0;
                  }
             break;

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

             PageGDICalls(hdc, cxClient, cyClient);

             EndPaint(hwnd, &ps);
             return 0;

        case WM_DESTROY :
             PostQuitMessage(0);
             return 0;
        }
   return DefWindowProc(hwnd, msg, wParam, lParam);
   }
```

Рис. 15.5 Файл PRINT.C программ PRINT1, PRINT2, PRINT3 и PRINT4

В файле PRINT.C имеются функции *WinMain, WndProc, GetPrinterDC* и *PageGDICalls*. Для последней функции необходим описатель контекста принтера и две переменных, содержащих ширину и высоту страницы принтера. Функция *PageGDICalls* рисует прямоугольник, который ограничивает всю страницу, две линии между противоположными углами страницы, эллипс в середине страницы (его диаметр вдвое меньше ширины и высоты страницы) и текст "Hello, Printers!" в центре эллипса.

При обработке сообщения WM_CREATE функция *WndProc* добавляет к системному меню опцию Print. Выбор этой опции приводит к вызову функции *PrintMyPage*, которую мы будем совершенствовать по мере перехода к очередной версии программы. Возвращаемым значением функции *PrintMyPage* является TRUE (ненулевое значение), если в процессе печати имела место ошибка, в противном случае ее возвращаемым значением будет FALSE. Если функция *PrintMyPage* возвращает TRUE, то *WndProc* выводит на экран окно сообщения об ошибке.

Каркас программы печати

Программа PRINT1, представленная на рис. 15.6, является первой версией программы печати. После компиляции программы PRINT1, ее можно запустить и затем выбрать из системного меню опцию Print. Если в переменной TEMP окружения MS-DOS указан жесткий диск (или если переменная TEMP не задана), то вы заметите некоторую дисковую активность, когда модуль GDI сохраняет выходные данные для печати во временном файле. После окончания работы программы PRINT1, спулер должен начать передачу файла с диска на принтер.

Рассмотрим программу в файле PRINT1.C. Если функция *PrintMyPage* не может получить для принтера описатель контекста, ее возвращаемым значением будет TRUE, и *WndProc* выведет на экран окно сообщения об ошибке. Если функция благополучно получает описатель контекста принтера, то она, с помощью вызова функции *GetDiviceCaps*, определяет горизонтальный и вертикальный размер страницы в пикселях:

```
xPage = GetDeviceCaps(hdcPrn, HORZRES);
yPage = GetDeviceCaps(hdcPrn, VERTRES);
```

Эти размеры представляют собой не размеры страницы, а скорее, размеры области печати на ней. Не считая этого вызова, структура кодов функции *PrintMyPage* программы PRINT1 такая же, как в программе FORMFEED, за исключением того, что в программе PRINT1 вызов функции *PageGDICalls* происходит между вызовами функций *StartPage* и *EndPage*. Только в том случае, если вызовы функций *StartDoc*, *StartPage* и *EndPage* прошли удачно, программа PRINT1 вызывает функцию печати *EndDoc*.

PRINT1.MAK

```
#---------------------
# PRINT1.MAK make file
#---------------------

print1.exe : print.obj print1.obj
     $(LINKER) $(GUIFLAGS) -OUT:print1.exe print.obj print1.obj \
     $(GUILIBS) winspool.lib

print.obj : print.c
     $(CC) $(CFLAGS) print.c

print1.obj : print1.c
     $(CC) $(CFLAGS) print1.c
```

PRINT1.C

```
/*--------------------------------------
   PRINT1.C -- Bare Bones Printing
               (c) Charles Petzold, 1996
  --------------------------------------*/

#include <windows.h>

HDC  GetPrinterDC(void);              // in PRINT.C
void PageGDICalls(HDC, int, int);

HINSTANCE hInst;
char      szAppName[] = "Print1";
char      szCaption[] = "Print Program 1";

BOOL PrintMyPage(HWND hwnd)
     {
     static DOCINFO di     = { sizeof(DOCINFO), "Print1: Printing", NULL };
     BOOL           bError = FALSE;
     HDC            hdcPrn;
     int            xPage, yPage;

     if(NULL ==(hdcPrn = GetPrinterDC()))
          return TRUE;

     xPage = GetDeviceCaps(hdcPrn, HORZRES);
     yPage = GetDeviceCaps(hdcPrn, VERTRES);

     if(StartDoc(hdcPrn, &di) > 0)
          {
          if(StartPage(hdcPrn) > 0)
               {
               PageGDICalls(hdcPrn, xPage, yPage);

               if(EndPage(hdcPrn) > 0)
                    EndDoc(hdcPrn);
               else
                    bError = TRUE;
               }
          }
     else
          bError = TRUE;

     DeleteDC(hdcPrn);
     return bError;
     }
```

Рис. 15.6 Программа PRINT1

Если попытаться запустить программу PRINT1, когда на диске, в котором находится подкаталог TEMP, недостаточно дискового пространства для хранения целой страницы с выходными данными для вывода графики, Windows выведет на экран сообщение об ошибке: *Windows could not write to the printer spool file. Make sure your disk has enough free space and try again* (Windows не может создать файл печати для спулера. Убедитесь, что на диске достаточно свободного пространства и повторите попытку).

Это не очень серьезная проблема — нужно просто щелкнуть на кнопке ОК для удаления окна сообщения. Затем вы можете удалить некоторые ненужные файлы для того, чтобы освободить дисковое пространство, как это и предлагается в сообщении об ошибке. Однако, если ошибка случается, когда последовательно, одно за другим на печать отправлено сразу несколько заданий, простое ожидание перед повтором попытки печати может решить проблему, которая возникает, когда в подкаталоге TEMP находится сразу несколько временных файлов, созданных для печати модулем GDI. Предоставьте спулеру достаточно времени для отправки этих файлов на принтер и дальнейшего освобождения дискового пространства, что даст возможность текущей программе продолжить печать.

Прерывание печати с помощью процедуры Abort

Что может случиться при непреднамеренном вводе в программу PRINT1 ошибки, ведущей к тому, что вместо одной страницы печатается целая кипа? Что делать, если принтер начал непрерывно прогонять бумагу? Конечно, всегда можно отсоединить разъем. Можно также прервать печать из папки Printers. Для этого надо выбрать опцию Settings из меню Start, выбрать Printers, дважды щелкнуть на значке своего принтера и для прерывания печати выбрать из меню Printers опцию Purge Print Jobs (удалить задания на печать). (К этому времени уже будет испорчено много страниц.)

В процессе печати в программе необходимо предусмотреть возможность прерывания. Когда буферизация разрешена, времени для прерывания печати не так много — только до тех пор, пока спулер не получит выходные данные и не создаст файл печати. После этого программа теряет управление процессом печати. (Однако, пользователь по-прежнему может остановить печать из папки Printers.) Когда буферизация запрещена, программа может прервать печать в любое время, пока буфер принтера не получит последние данные для печати.

Для отмены задания на печать непосредственно из программы требуется какая-нибудь процедура прерывания. Процедура прерывания — это небольшая экспортируемая функция вашей программы. Адрес этой функции передается в Windows с помощью функции *SetAbortProc*; затем GDI периодически во время печати вызывает эту процедуру, спрашивая: "Можно ли продолжать печать?".

Сначала рассмотрим, что же необходимо, чтобы добавить к логике печати процедуру прерывания, а затем рассмотрим, к чему это ведет. Процедура прерывания обычно называется *AbortProc* и выглядит следующим образом:

```
BOOL CALLBACK AbortProc(HDC hdcPrn, int iCode)
{
       [другие строки программы]
}
```

Перед началом печати необходимо зарегистрировать процедуру прерывания с помощью вызова функции *SetAbortProc*:

```
SetAbortProc(hdcPrn, AbortProc);
```

Вызов этой функции делается перед вызовом функции *StartDoc*. Отменять регистрацию процедуры прерывания после окончания печати нет необходимости.

При обработке вызова функции *EndPage* (т. е. при проигрывании расширенного метафайла в драйвере устройства и создании временных файлов с выходными данными для принтера) GDI часто вызывает процедуру прерывания. Параметр *hdcPrn* — это описатель контекста принтера. Параметр *iCode* устанавливается в 0, если все идет нормально, или в SP_OUTOFDISK, если модуль GDI сталкивается с нехваткой свободного дискового пространства при создании временных файлов с выходными данными для принтера.

Возвращаемым значением процедуры *AbortProc* должно быть TRUE (ненулевое значение), если задание на печать следует продолжать или FALSE (0), если задание на печать следует прервать. Процедура прерывания может быть достаточно простой:

```
BOOL CALLBACK AbortProc(HDC hdcPrn, int iCode)
{
       MSG msg;

       while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
       {
              TranslateMessage(&msg);
              DispatchMessage(&msg);
```

```
        }
        return TRUE;
}
```

Эта функция может показаться несколько необычной. То есть, она подозрительно похожа на цикл обработки сообщений. Однако обратите внимание, что в этом цикле обработки сообщений вместо функции *GetMessage* вызывается функция *PeekMessage*. О функции *PeekMessage* было рассказано в главе 4 при рассмотрении программы RANDRECT. Вспомните, что функция *PeekMessage* возвращает управление программе не только при получении сообщения из очереди сообщений программы (точно также, как и функция *GetMessage*), но и в том случае, если в очереди сообщений программы нет сообщений.

Цикл обработки сообщений в функции *AbortProc* вызывает функцию *PeekMessage* до тех пор, пока возвращаемым значением функции *PeekMessage* остается TRUE. Величина TRUE означает, что функция *PeekMessage* извлекла из очереди сообщение, которое может быть передано одной из оконных процедур программы, используя функции *TranslateMessage* и *DispatchMessage*. При отсутствии сообщений в очереди сообщений программы, возвращаемым значением функции *PeekMessage* становится FALSE, поэтому функция *AbortProc* возвращает управление Windows.

Как Windows использует функцию *AbortProc*

Когда программа печатает, часть работы выполняется во время вызова функции *EndPage*. До этого вызова модуль GDI просто добавляет очередную запись к расположенному на диске расширенному метафайлу каждый раз, когда программа вызывает функции рисования GDI. При вызове функции *EndPage*, модуль GDI проигрывает этот метафайл в драйвере устройства для каждой полосы, которую драйвер устройства определяет на странице. Затем GDI сохраняет в файле выходные данные для принтера, созданные драйвером принтера. Если спулер не активен, то сам модуль GDI должен выводить эти данные на принтер.

При обработке вызова функции *EndPage*, модуль GDI вызывает вашу процедуру прерывания. Обычно параметр *iCode* равен 0, но если, из-за наличия других временных файлов, которые еще не напечатаны, модуль GDI сталкивается с нехваткой свободного дискового пространства, то тогда параметр *iCode* устанавливается равным SP_OUTOFDISK. (Это значение обычно не проверяется, но при желании это можно сделать.) Затем процедура прерывания входит в цикл *PeekMessage* для извлечения сообщений из очереди сообщений программы.

Если в очереди сообщений программы нет сообщений, возвращаемым значением функции *PeekMessage* является FALSE. Процедура прерывания выходит из цикла обработки сообщений и возвращает TRUE в модуль GDI показывая, что печать должна продолжаться. После этого модуль GDI продолжает обработку вызова функции *EndPage*.

Если случается ошибка, то модуль GDI останавливает процесс печати. Следовательно, главная цель процедуры прерывания заключается в том, чтобы дать возможность пользователю прервать печать. Для этого нам необходимо окно диалога, в котором на экран выводится кнопка Cancel. Давайте выполним два этих шага по очереди. Сначала добавим процедуру прерывания в создаваемую программу PRINT2, а затем в программу PRINT3 добавим окно диалога с кнопкой Cancel, чтобы процедуру прерывания можно было использовать.

Реализация процедуры прерывания

Давайте коротко вспомним организацию процедуры прерывания. Наша процедура прерывания выглядит так:

```
BOOL CALLBACK AbortProc(HDC hdcPrn, int iCode)
{
        MSG msg;


        while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
        }
        return TRUE;
}
```

Если необходимо что-то напечатать, мы передаем Windows указатель на эту функцию с помощью вызова:

```
SetAbortProc(hdcPrn, AbortProc);
```

Этот вызов делается перед вызовом функции *StartDoc*. И это все.

Все, но не совсем. Мы упустили из виду проблему, связанную с циклом *PeekMessage* в функции *AbortProc*, а это серьезная проблема. Функция *AbortProc* вызывается только в самом разгаре печати. Масса очень неприятных вещей может произойти, если вы извлечете сообщение из очереди в функции *AbortProc*, и передадите его в собственную оконную процедуру. Пользователь может повторно выбрать в меню опцию Print. Но программа уже находится в середине процесса печати. Пользователь может загрузить новый файл в программу в то время, пока

программа пытается напечатать предыдущий файл. Пользователь может даже прекратить работу с программой! Если это случается, все окна программы будут закрыты. В конце концов, произойдет возврат из функции печати, но оконной процедуры уже не будет.

Этот список проблем заставляет усомниться в правильности программы. Она к этому не готова. По этой причине, при установке процедуры прерывания сначала следует сделать окно программы недоступным, чтобы программа не могла получать сообщения клавиатуры и мыши. Это делается так:

```
EnableWindow(hwnd, FALSE);
```

Это предотвратит появление в очереди сообщений от клавиатуры и мыши. Таким образом, во время печати пользователь не сможет ничего сделать с программой. После окончания печати нужно разблокировать окно:

```
EnableWindow(hwnd, TRUE);
```

Тогда зачем мы должны использовать вызовы функций *TranslateMessage* и *DispatchMessage* в *AbortProc*, если ни одного сообщения от клавиатуры и мыши не окажется в очереди сообщений? Действительно, вызов функции *TranslateMessage*, строго говоря, больше не нужен (хотя он почти всегда присутствует). Однако на случай появления в очереди сообщений сообщения WM_PAINT, вызов функции *DispatchMessage* необходим. Если сообщение WM_PAINT должным образом не обрабатывается в оконной процедуре парой функций *BeginPaint* и *EndPaint*, то оно остается в очереди и останавливает всю работу, поскольку возвращаемое значение функции *PeekMessage* никогда не станет равным FALSE.

Если во время печати окно программы блокируется, программа остается на экране в инертном состоянии. Но пользователь имеет возможность переключаться на другие программы и что-то делать там, а спулер может продолжать отправлять на принтер выходные файлы.

В программе PRINT2, представленной на рис. 15.7, к логике программы PRINT1 добавлена процедура прерывания (и необходимая для этого поддержка). Более подробно, в программу PRINT2 добавлены процедура прерывания, вызов функции *SetAbortProc* и два вызова функции *EnableWindow*, первый — чтобы заблокировать окно, а второй — чтобы его разблокировать.

PRINT2.MAK

```
#----------------------
# PRINT2.MAK make file
#----------------------

print2.exe : print.obj print2.obj
     $(LINKER) $(GUIFLAGS) -OUT:print2.exe print.obj print2.obj \
     $(GUILIBS) winspool.lib

print.obj : print.c
     $(CC) $(CFLAGS) print.c

print2.obj : print2.c
     $(CC) $(CFLAGS) print2.c
```

PRINT2.C

```
/*----------------------------------------
   PRINT2.C -- Printing with Abort Function
               (c) Charles Petzold, 1996
  ----------------------------------------*/

#include <windows.h>

HDC  GetPrinterDC(void);              // in PRINT.C
void PageGDICalls(HDC, int, int);

HINSTANCE hInst;
char      szAppName[] = "Print2";
char      szCaption[] = "Print Program 2(Abort Function)";

BOOL CALLBACK AbortProc(HDC hdcPrn, int iCode)
     {
     MSG msg;

     while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
          {
```

```
          TranslateMessage(&msg);
          DispatchMessage(&msg);
          }
     return TRUE;
     }

BOOL PrintMyPage(HWND hwnd)
     {
     static DOCINFO di     = { sizeof(DOCINFO), "Print2: Printing", NULL };
     BOOL           bError = FALSE;
     HDC            hdcPrn;
     short          xPage, yPage;

     if(NULL ==(hdcPrn = GetPrinterDC()))
          return TRUE;

     xPage = GetDeviceCaps(hdcPrn, HORZRES);
     yPage = GetDeviceCaps(hdcPrn, VERTRES);

     EnableWindow(hwnd, FALSE);

     SetAbortProc(hdcPrn, AbortProc);

     if(StartDoc(hdcPrn, &di) > 0)
          {
          if(StartPage(hdcPrn) > 0)
               {
               PageGDICalls(hdcPrn, xPage, yPage);

               if(EndPage(hdcPrn) > 0)
                    bError = TRUE;
               }
          }
     else
          bError = TRUE;

     if(!bError)
          EndDoc(hdcPrn);

     EnableWindow(hwnd, TRUE);
     DeleteDC(hdcPrn);
     return bError;
     }
```

Рис. 15.7 Программа PRINT2

Добавление диалогового окна печати

Программа PRINT2 не вполне удовлетворительна. В программе непосредственно не показывается момент печати и его завершение. Если только нажать клавишу мыши в окне программы и обнаружить, что она ни на что не реагирует, можно определить, что функция *PrintMyPage* еще продолжает свою работу. В программе PRINT2 нет ничего, что дало бы пользователю возможность отменить задание на печать, пока идет буферизация.

Вы вероятно знаете, что большинство программ для Windows предоставляют пользователю возможность отменить уже запущенную операцию печати. На экране появляется маленькое окно диалога, в котором находится некоторый текст и кнопка Cancel. Это окно остается на экране все время, пока GDI записывает выходные данные для принтера в файл на диске, или, если спулер отключен, пока печатает принтер. Это немодальное окно диалога и, следовательно, вам необходима создать оконную процедуру для этого окна диалога.

Часто такое окно диалога называют диалоговым окном прерывания (abort dialog box), а процедуру диалога — диалоговой процедурой прерывания (abort dialog procedure). Чтобы проще было отличить ее от процедуры прерывания, назовем ее диалоговой процедурой печати. Процедура прерывания (*AbortProc*) и диалоговая процедура печати (*PrintDlgProc*) — это две разные экспортируемые функции. Если вы хотите печатать в профессиональной манере, как это делается в Windows, вы должны иметь обе эти функции.

Эти функции взаимодействуют следующим образом: цикл *PeekMessage* в процедуре *AbortProc* необходимо изменить так, чтобы он отправлял сообщения в оконную процедуру немодального окну диалога. Функция *PrintDlgProc*, чтобы контролировать состояние кнопки Cancel, должна обрабатывать сообщения WM_COMMAND. Если кнопка Cancel нажимается, значение переменной *bUserAbort* устанавливается в TRUE. Возвращаемым значением функции *AbortProc* является величина, обратная *bUserAbort*. Вспомните, что функция *AbortProc* возвращает TRUE для продолжения печати и FALSE для отказа от печати. В программе PRINT2 ее возвращаемое значение всегда было равно TRUE. Теперь, при нажатии в диалоговом окне печати кнопки Cancel, ее возвращаемым значением будет FALSE. Эта логика реализована в программе PRINT3, представленной на рис. 15.8.

При работе с программой PRINT3 понадобится временно отключить буферизацию печати. В противном случае, кнопка Cancel, которая появляется только пока спулер получает данные от программы PRINT3, может исчезнуть раньше, чем вы успеете щелкнуть на ней. Для отключения буферизации выберите из меню Start опцию Settings, выберите Printers, щелкните правой кнопкой на значке текущего принтера и щелкните на опции Properties. Установки спулера находятся на странице Details.

Не удивляйтесь, если при нажатии кнопки Cancel никакой остановки сразу не произойдет, особенно при наличии медленного принтера. У принтера имеется внутренний буфер, который перед остановкой принтера должен очиститься. Щелчок на кнопке Cancel только запрещает GDI посылать в буфер принтера какие-либо данные.

PRINT3.MAK

```
#----------------------
# PRINT3.MAK make file
#----------------------

print3.exe : print.obj print3.obj print.res
     $(LINKER) $(GUIFLAGS) -OUT:print3.exe print.obj print3.obj \
     print.res $(GUILIBS) winspool.lib
print.obj : print.c
     $(CC) $(CFLAGS) print.c

print3.obj : print3.c
     $(CC) $(CFLAGS) print3.c

print.res : print.rc
     $(RC) $(RCVARS) print.rc
```

PRINT3.C

```
/*--------------------------------------
   PRINT3.C -- Printing with Dialog Box
               (c) Charles Petzold, 1996
  --------------------------------------*/

#include <windows.h>

HDC  GetPrinterDC(void);              // in PRINT.C
void PageGDICalls(HDC, int, int);

HINSTANCE hInst;
char      szAppName[] = "Print3";
char      szCaption[] = "Print Program 3(Dialog Box)";

BOOL   bUserAbort;
HWND   hDlgPrint;

BOOL CALLBACK PrintDlgProc(HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam)
     {
     switch(msg)
          {
          case WM_INITDIALOG :
               SetWindowText(hDlg, szAppName);
               EnableMenuItem(GetSystemMenu(hDlg, FALSE), SC_CLOSE,
                                                          MF_GRAYED);
               return TRUE;
```

```
          case WM_COMMAND :
               bUserAbort = TRUE;
               EnableWindow(GetParent(hDlg), TRUE);
               DestroyWindow(hDlg);
               hDlgPrint = 0;
               return TRUE;
          }
     return FALSE;
     }

BOOL CALLBACK AbortProc(HDC hdcPrn, int iCode)
     {
     MSG   msg;
     while(!bUserAbort && PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
          {
          if(!hDlgPrint || !IsDialogMessage(hDlgPrint, &msg))
               {
               TranslateMessage(&msg);
               DispatchMessage(&msg);
               }
          }
     return !bUserAbort;
     }

BOOL PrintMyPage(HWND hwnd)
     {
     static DOCINFO di     = { sizeof(DOCINFO), "Print3: Printing", NULL };
     BOOL           bError = FALSE;
     HDC            hdcPrn;
     int            xPage, yPage;

     if(NULL ==(hdcPrn = GetPrinterDC()))
          return TRUE;

     xPage = GetDeviceCaps(hdcPrn, HORZRES);
     yPage = GetDeviceCaps(hdcPrn, VERTRES);

     EnableWindow(hwnd, FALSE);

     bUserAbort = FALSE;
     hDlgPrint = CreateDialog(hInst, "PrintDlgBox", hwnd, PrintDlgProc);

     SetAbortProc(hdcPrn, AbortProc);

     if(StartDoc(hdcPrn, &di) > 0)
          {
          if(StartPage(hdcPrn) > 0)
               {
               PageGDICalls(hdcPrn, xPage, yPage);

               if(EndPage(hdcPrn) > 0)
                    EndDoc(hdcPrn);
               else
                    bError = TRUE;
               }
          }
     else
          bError = TRUE;

     if(!bUserAbort)
          {
          EnableWindow(hwnd, TRUE);
          DestroyWindow(hDlgPrint);
          }
```

```
     DeleteDC(hdcPrn);

     return bError || bUserAbort;
     }
```

PRINT.RC

```
/*-------------------------
   PRINT.RC resource script
  -------------------------*/

#include <windows.h>

PrintDlgBox DIALOG 40, 40, 120, 40
     STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU | WS_VISIBLE
     {
     CTEXT          "Cancel Printing", -1,  4,  6, 120, 12
     DEFPUSHBUTTON  "Cancel",    IDCANCEL, 44, 22,  32, 14, WS_GROUP
     }
```

Рис. 15.8 Программа PRINT3

В программу PRINT3 добавлены две глобальные переменные: *bUserAbort* типа BOOL и описатель окна диалога *hDlgPrint*. Функция *PrintMyPage* инициализирует переменную *bUserAbort* значением FALSE и, как в программе PRINT2, главное окно программы блокируется. При вызове функции *SetAbortProc* используется указатель на *AbortProc*, а при вызове функции *CreateDialog* — указатель на *PrintDlgProc*. Возвращаемый функцией *CreateDialog* описатель окна хранится в переменной *hDlgPrint*.

Цикл обработки сообщений в *AbortProc* теперь становится таким:

```
while(!bUserAbort && PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
{
       if(!hDlgPrint || !IsDialogMessage(hDlgPrint, &msg))
       {
              TranslateMessage(&msg);
              DispatchMessage(&msg);
       }
}

return !bUserAbort;
```

Здесь функция *PeekMessage* вызывается только в том случае, если *bUserAbort* равно FALSE, то есть, если пользователь еще не отказался от печати. Функция *IsDialogMessage* нужна для отправки сообщения немодальному окну диалога. Как это обычно делается для немодальных окон диалога, описатель окна диалога проверяется перед этим вызовом. Возвращаемым значением функции *AbortProc* является величина, обратная *bUserAbort*. Сначала *bUserAbort* равно FALSE, следовательно возвращаемым значением функции *AbortProc* является TRUE, что показывает необходимость продолжения печати. Но в диалоговой процедуре печати *bUserAbort* может быть установлена в TRUE.

Функция *PrintDlgProc* совершенно проста. При обработке сообщения WM_INITDIALOG эта функция записывает в заголовок окна имя программы и делает недоступной опцию Close системного меню. Если пользователь нажимает кнопку Cancel, функция *PrintDlgProc* получает сообщение WM_COMMAND:

```
case WM_COMMAND :
       bUserAbort = TRUE;

       EnableWindow(GetParent(hDlg), TRUE);

       DestroyWindow(hDlg);
       hDlgPrint = 0;
       return TRUE;
```

Установка переменной *bUserAbort* в TRUE показывает, что пользователь решил отказаться от печати. Главное окно программы блокируется и окно диалога закрывается. (Очень важно, чтобы эти две операции выполнялись именно в таком порядке. В противном случае какие-то другие программы, работающие под Windows, станут активными, и ваша программа может перейти в фоновый режим.) Если все нормально, то *hDlgPrint* устанавливается в 0, чтобы предотвратить вызов функции *IsDialogMessage* из цикла обработки сообщений.

Окно диалога получает сообщения только тогда, когда процедура *AbortProc* с помощью функции *PeekMessage* получает сообщения и отправляет их оконной процедуре окна диалога с помощью функции *IsDialogMessage*. Процедура *AbortProc* вызывается только тогда, когда модуль GDI обрабатывает функцию *EndPage*. Если модуль

GDI обнаруживает, что возвращаемым значением функции *AbortProc* является FALSE, он возвращает управление функции из *EndPage* в *PrintMyPage*. Код ошибки она не возвращает. В этот момент функция *PrintMyPage* считает, что страница закончена и вызывает функцию *EndProc*. Однако, ничего на печать не выводится, поскольку модуль GDI не закончил обработку функции *EndPage*.

Осталось только кое-что очистить. Если пользователь не снимает задания на печать с помощью окна диалога, то окно диалога продолжает оставаться на экране. Функция *PrintMyPage* разблокирует главное окно и закрывает окно диалога:

```
if(!bUserAbort)
{
       EnableWindow(hwnd, TRUE);
       DestroyWindow(hDlgPrint);
}
```

О том, что произошло, информируют две переменные: переменная *bUserAbort* сообщает, что пользователь отказался от задания на печать, а переменная *bError* сообщает, что имела место ошибка. При желании, с этими переменными можно работать дальше. Чтобы вернуться в *WndProc*, в функции *PrintMyPage* просто реализована логическая операция OR:

```
return bError || bUserAbort;
```

Добавление печати к программе POPPAD

Теперь мы готовы добавить возможность печати к программам серии POPPAD. Можно сказать, что работа над программой POPPAD на этом завершается. Нам необходимы различные файлы программы POPPAD из главы 11, а также два новых файла, представленных на рис. 15.9.

POPPAD.MAK

```
#----------------------
# poppad.MAK make file
#----------------------

poppad.exe : poppad.obj popfile.obj popfind.obj \
             popfont.obj popprnt.obj poppad.res
     $(LINKER) $(GUIFLAGS) -OUT:poppad.exe poppad.obj popfile.obj \
     popfind.obj popfont.obj popprnt.obj poppad.res $(GUILIBS)

poppad.obj : poppad.c poppad.h
     $(CC) $(CFLAGS) poppad.c

popfile.obj : popfile.c
     $(CC) $(CFLAGS) popfile.c

popfind.obj : popfind.c
     $(CC) $(CFLAGS) popfind.c

popfont.obj : popfont.c
     $(CC) $(CFLAGS) popfont.c

popprnt.obj : popprnt.c
     $(CC) $(CFLAGS) popprnt.c

poppad.res : poppad.rc poppad.h poppad.ico
     $(RC) $(RCVARS) poppad.rc
```

POPPRNT.C

```
/*-------------------------------------------
   POPPRNT.C -- Popup Editor Printing Functions
  -------------------------------------------*/
#include <windows.h>
#include <commdlg.h>
#include <string.h>
#include "poppad.h"

BOOL bUserAbort;
HWND hDlgPrint;
```

```
BOOL CALLBACK PrintDlgProc(HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam)
     {
     switch(msg)
          {
          case WM_INITDIALOG :
               EnableMenuItem(GetSystemMenu(hDlg, FALSE), SC_CLOSE,
                                                          MF_GRAYED);
               return TRUE;

          case WM_COMMAND :
               bUserAbort = TRUE;
               EnableWindow(GetParent(hDlg), TRUE);
               DestroyWindow(hDlg);
               hDlgPrint = 0;
               return TRUE;
          }
     return FALSE;
     }

BOOL CALLBACK AbortProc(HDC hPrinterDC, int iCode)
     {
     MSG msg;

     while(!bUserAbort && PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
          {
          if(!hDlgPrint || !IsDialogMessage(hDlgPrint, &msg))
               {
               TranslateMessage(&msg);
               DispatchMessage(&msg);
               }
          }
     return !bUserAbort;
     }

BOOL PopPrntPrintFile(HINSTANCE hInst, HWND hwnd, HWND hwndEdit,
                                                  LPSTR szTitleName)
     {
     static DOCINFO  di = { sizeof(DOCINFO), "", NULL };
     static PRINTDLG pd;
     BOOL            bSuccess;
     LPCTSTR         pstrBuffer;
     int             yChar, iCharsPerLine, iLinesPerPage, iTotalLines,
                     iTotalPages, iPage, iLine, iLineNum;
     TEXTMETRIC      tm;
     WORD            iColCopy, iNoiColCopy;

     pd.lStructSize         = sizeof(PRINTDLG);
     pd.hwndOwner           = hwnd;
     pd.hDevMode            = NULL;
     pd.hDevNames           = NULL;
     pd.hDC                 = NULL;
     pd.Flags               = PD_ALLPAGES | PD_COLLATE | PD_RETURNDC;
     pd.nFromPage           = 0;
     pd.nToPage             = 0;
     pd.nMinPage            = 0;
     pd.nMaxPage            = 0;
     pd.nCopies             = 1;
     pd.hInstance           = NULL;
     pd.lCustData           = 0L;
     pd.lpfnPrintHook       = NULL;
     pd.lpfnSetupHook       = NULL;
     pd.lpPrintTemplateName = NULL;
     pd.lpSetupTemplateName = NULL;
     pd.hPrintTemplate      = NULL;
```

```
pd.hSetupTemplate      = NULL;

if(!PrintDlg(&pd))
     return TRUE;

iTotalLines =(short) SendMessage(hwndEdit, EM_GETLINECOUNT, 0, 0L);

if(iTotalLines == 0)
     return TRUE;

GetTextMetrics(pd.hDC, &tm);
yChar = tm.tmHeight + tm.tmExternalLeading;

iCharsPerLine = GetDeviceCaps(pd.hDC, HORZRES) / tm.tmAveCharWidth;
iLinesPerPage = GetDeviceCaps(pd.hDC, VERTRES) / yChar;
iTotalPages   =(iTotalLines + iLinesPerPage - 1) / iLinesPerPage;

pstrBuffer =(LPCTSTR) HeapAlloc(GetProcessHeap(),
                       HEAP_NO_SERIALIZE, iCharsPerLine + 1);

EnableWindow(hwnd, FALSE);

bSuccess   = TRUE;
bUserAbort = FALSE;

hDlgPrint = CreateDialog(hInst,(LPCTSTR) "PrintDlgBox", hwnd, PrintDlgProc);
SetDlgItemText(hDlgPrint, IDD_FNAME, szTitleName);
SetAbortProc(pd.hDC, AbortProc);

GetWindowText(hwnd,(PTSTR) di.lpszDocName, sizeof(PTSTR));

if(StartDoc(pd.hDC, &di) > 0)
     {
     for(iColCopy = 0;
          iColCopy <((WORD) pd.Flags & PD_COLLATE ? pd.nCopies : 1);
          iColCopy++)
          {
          for(iPage = 0; iPage < iTotalPages; iPage++)
               {
               for(iNonColCopy = 0;
                    iNonColCopy <(pd.Flags & PD_COLLATE ? 1 : pd.nCopies);
                    iNonColCopy++)
                    {

                    if(StartPage(pd.hDC) < 0)
                         {
                         bSuccess = FALSE;
                         break;
                         }

                    for(iLine = 0; iLine < iLinesPerPage; iLine++)
                         {
                         iLineNum = iLinesPerPage * iPage + iLine;

                         if(iLineNum > iTotalLines)
                              break;

                         *(int *) pstrBuffer = iCharsPerLine;

                         TextOut(pd.hDC, 0, yChar * iLine, pstrBuffer,
                             (int) SendMessage(hwndEdit, EM_GETLINE,
                             (WPARAM) iLineNum,(LPARAM) pstrBuffer));
                         }

                    if(EndPage(pd.hDC) < 0)
```

```
                         {
                         bSuccess = FALSE;
                         break;
                         }

                    if(bUserAbort)
                         break;
                    }

               if(!bSuccess || bUserAbort)
                    break;
               }
          if(!bSuccess || bUserAbort)
               break;
          }
     }
else
     bSuccess = FALSE;

if(bSuccess)
     EndDoc(pd.hDC);

if(!bUserAbort)
     {
     EnableWindow(hwnd, TRUE);
     DestroyWindow(hDlgPrint);
     }

HeapFree(GetProcessHeap(), 0,(LPVOID) pstrBuffer);
DeleteDC(pd.hDC);

return bSuccess && !bUserAbort;
}
```

Рис. 15.9 Новые файлы, добавляющие в программу POPPAD возможность печати

Пытаясь сохранить философию создания программ POPPAD, которая заключается в том, чтобы делать эти программы как можно проще, используя для этого преимущества, которые дает Windows, в файле POPPRNT.C демонстрируется как использовать функцию *PrintDlg*. Эта функция включена в библиотеку диалоговых окон общего пользования, и использует структуру типа PRNTDLG.

Как правило, опция Print включается в меню File программы. Если пользователь выбирает опцию Print, программа может инициализировать поля структуры PRNTDLG и вызвать функцию *PrintDlg*.

Функция *PrintDlg* выводит на экран окно диалога, которое дает пользователю возможность выбрать диапазон печатаемых страниц. Таким образом, это окно диалога особенно удобно для таких программ, как POPPAD, которые могут печатать многостраничные документы. В окне диалога имеется также поле редактирования для задания количества копий и флажок Collate (разобрать по копиям). Этот флажок определяет порядок страниц при печати многостраничных документов. Например, если документ состоит из трех страниц и пользователь хочет, чтобы были напечатаны три копии, то программа может их напечатать двумя способами. Если установлен флажок Collate, то копии печатаются в следующем порядке 1, 2, 3, 1, 2, 3, 1, 2, 3. Если же флажок Collate сброшен, то копии печатаются в следующем порядке: 1, 1, 1, 2, 2, 2, 3, 3, 3. Программа сама заботиться о соблюдении правильного порядка печати.

Окно диалога также позволяет пользователю выбрать принтер, отличный от заданного по умолчанию. В этом же диалоговом окне имеется кнопка Properties, нажатие которой приводит к появлению окна диалога настроек принтера. Это окно позволяет, как минимум, выбрать режим portrait и landscape.

После того, как функция *PrintDlg* возвращает управление, в полях структуры PRNTDLG установлен диапазон печатаемых страниц и состояние флажка Collate. Кроме этого в структуре находится готовый к использованию описатель контекста принтера.

В файле POPPRNT.C функция *PopPrntPrintFile* (которая вызывается из программы POPPAD, если пользователь выбирает в меню File опцию Print) вызывает функцию *PrintDlg* и затем продолжает печать файла. Далее функция *PopPrntPrintFile* выполняет кое-какие вычисления, чтобы определить количество символов, которые могут разместиться в строке и количество строк, которые могут разместиться на странице. Этот процесс включает в себя

вызов функции *GetDeviceCaps* для определения разрешающей способности страницы и функции *GetTextMetrics* для определения размеров символа.

Посылая сообщение EM_GETLINECOUNT дочернему окну редактирования, программа получает полное количество строк в документе (переменная *iTotalLines*). Буфер для хранения содержимого каждой строки выделяется в локальной памяти. Для каждой строки первое слово в этом буфере устанавливается равным числу символов строки. При посылке окну редактирования сообщения EM_GETLINE строка копируется в буфер, а затем, с помощью функции *TextOut*, отправляется в контекст принтера.

Обратите внимание, что логика печати документа для числа копий включает в себя два цикла *for*. В первом используется переменная *iColCopy*, и она используется в случае установленного флажка Collate, а во втором — переменная *iNonColCopy*, которая используется в противном случае.

Программа выходит из цикла *for*, инкрементирующего номер страницы, если функция *StartPage* или *EndPage* возвращает ошибку, либо переменная *bUserAbort* равна TRUE. Если возвращаемое значение процедуры прерывания равно FALSE, то функция *EndPage* не возвращает ошибку. Поэтому значение переменной *bUserAbort* проверяется непосредственно перед началом печати следующей страницы. Если ошибки нет, то делается вызов функции *EndDoc*:

```
if(!bError)   EndDoc(hdcPrn);
```

Вы можете поэкспериментировать с печатью многостраничных файлов в программе POPPAD. Процесс печати можно наблюдать в окне заданий на печать, которое выводится на экран при выборе опции Open меню File папки Printers. Файл, печатаемый первым, появляется в начале окна заданий на печать после того, как GDI заканчивает обработку первого вызова функции *EndPage*. В это время спулер начинает посылать файл на принтер. Если затем отказаться от задания на печать из программы POPPAD, спулер тоже прекратит печать. Это является результатом того, что возвращаемым значением процедуры прерывания становится FALSE. После того, как файл появляется в окне заданий на печать, отказаться от печати можно также выбрав в меню Document опцию Cancel Printing. В этом случае вызов функции *EndPage* в программе POPPAD возвращает ошибку SP_ERROR (равную —1).

Новички программирования под Windows часто излишне пугаются, когда имеют дело с функцией *AbortDoc*. При печати эта функция используется редко. Как видно из программы POPPAD, пользователь может почти всегда отказаться от задания на печать, либо в диалоговом окне печати программы POPPAD, либо в окне заданий на печать. Ни то, ни другое не требует использования функции *AbortDoc*. Вызов функции *AbortDoc* можно было бы вставить в программу POPPAD только в одном месте: между вызовом функции *StartDoc* и первым вызовом функции *EndPage*, но коды этих функций работают столь быстро, что функция *AbortDoc* не является необходимой.

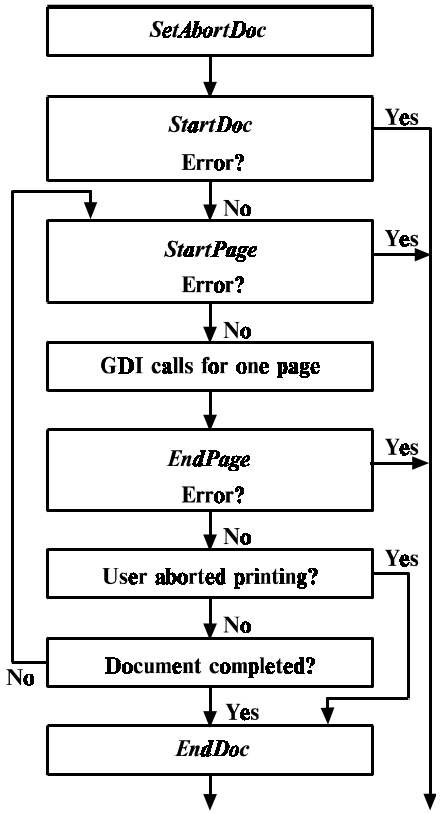


Рис. 15.10 Последовательность вызовов функций для печати документа

На рис. 15.10 показана правильная последовательность вызовов функций печати при печати многостраничного документа. Проверять, что значение переменной *bUserAbort* равно TRUE, лучше после каждого вызова функции *EndPage*. Функция *EndDoc* используется только тогда, когда предыдущие функции печати отработали без ошибок. На самом деле, при получении ошибки при вызове любой из показанных выше функций печати, печать прекращается.

Обработка кодов ошибок

До сих пор мы обрабатывали возвращаемые значения функций печати относительно просто: если функция печати (например, *StartDoc*) возвращает отрицательное значение, значит имеет место ошибка, и операция печати прерывается. Отрицательный код ошибки равен SP_ERROR (—1), и это единственное возвращаемое значение функций печати Win32, которое показывает наличие ошибки. Само по себе значение SP_ERROR о многом не говорит.

Более точно сообщить пользователю об ошибке можно с помощью функции *GetLastError*, вызывая ее сразу после ошибки. Если вызвать функцию *GetLastError* после того, как функция печати возвращает ошибку, то ее возвращаемым значением будет один из пяти возможных идентификаторов, которые определяются в файле WINGDI.H. Кроме них в файле WINGDI.H имеется идентификатор SP_NOTREPORTED, его значение равно 0x4000. Если результат поразрядной операции AND возвращаемого значения функции *GetLastError* и идентификатора SP_NOTREPORTED равен 0, то значит пользователь уже получил сообщение об ошибке. Результат поразрядной операции OR возвращаемого значения функции *GetLastError* и идентификатора SP_NOTREPORTED можно сравнить с пятью идентификаторами кодов ошибки для определения того, было ли уже выведено сообщение об ошибке или нет.

Следующий фрагмент программы показывает один из методов получения строки текста, идентифицирующей ошибку. Предполагается, что если возвращаемое значение функции печати отрицательно, то флаг *bError* устанавливается в TRUE. В представленном фрагменте этот флаг проверяется и, если имела место ошибка, вызывается функция *_GetErrorText*. Затем пользователю выводится окно сообщения об ошибке. Если пользователь к этому времени уже получил сообщение об ошибке, то возвращаемым значением функции *_GetErrorText* является NULL:

```
if( bError == TRUE )
{
       pszErrMsg = _GetErrorText();
       if(pszErrMsg) MessageBox(hwnd, pszErrMsg, NULL, MB_OK);
}
…
LPCTSTR _GetErrorText(void)
{
       static LPCTSTR pszErrorText[] = { "General error",
                                         "Canceled from program",
                                         "Canceled by user",
                                         "Out of disk space",
                                         "Out of memory" };
       DWORD dwError = GetLastError();

       if((dwError & SP_NOTREPORTED) == 0) return NULL;

       return pszErrorText[~dwError];
}
```

Далее приведены пять кодов ошибок с описанием возможных причин:

- SP_ERROR (0xFFFF или -1) — определяется как "general error" (общая ошибка). Это единственный код ошибки, являющийся возвращаемым значением функций печати в Win32. Он показывает, что модуль GDI или драйвер принтера не могут начать печать документа.
- SP_APPABORT (0xFFFE или -2) — этот код, как определено в документации, показывает, что процедура прерывания программы возвратила значение FALSE. Однако, это работает только при отключенном спулере. Если спулер работает, и если процедуре прерывания передается параметр *iCode* равный 0, а она возвращает значение FALSE, то возвращаемое значение функции *EndPage* будет положительно.
- SP_USERABORT (0xFFFD или -3) — этот код показывает, что пользователь отказался от задания на печать из окна папки Printers.
- SP_OUTOFDISK (0xFFFC или -4) — этот код показывает отсутствие свободного дискового пространства. Такой код ошибки появляется в том случае, если жесткий диск, на котором находится подкаталог TEMP, не может вместить ни одного временного файла спулера. Если в подкаталоге TEMP уже имеется

несколько временных файлов спулера, тогда во время вызова функции *EndPage* вызывается процедура прерывания с параметром *iCode* равным SP_OUTOFDISK. Если после этого возвращаемым значением процедуры прерывания является FALSE, то возвращаемым значением функции *EndPage* становится SP_ERROR. Очередной вызов функции *GetLastError* возвратит ошибку SP_OUTOFDISK.

- SP_OUTOFMEMORY (0xFFFB или -5) — этот код показывает нехватку памяти для печати.

Техника разбиения на полосы

Разбиение на полосы — это прием такого определения страницы графики, при котором страница разбивается на ряд отдельных прямоугольников, которые называются полосами. Такой подход освобождает драйвер принтера от необходимости создания в оперативной памяти битового образа целой страницы. Разбиение на полосы наиболее важно для таких растровых принтеров, в которых нет высокого уровня управления процессом создания страниц, например для матричных принтеров или некоторых лазерных принтеров.

Разбиение на полосы — это один из самых непонятных аспектов программирования для принтеров в Windows. Часть проблемы обнаруживается уже в документации, касающейся функции *GetDeviceCaps*. Бит RC_BANDING возвращаемого значения функции *GetDeviceCaps* с параметром RASTERCAPS, означает, как написано в документации, поддержку разбиения на полосы. Программисты, изучая эту документацию, считают, что в их приложениях для таких принтеров следует использовать разбиение на полосы. Но это совсем не так. Большая часть информации, возвращаемой функцией *GetDeviceCaps*, относится только к модулю GDI. Эта информация позволяет модулю GDI определять, что устройство может реализовать самостоятельно, а в чем ему необходима помощь. Возможность разбиения на полосы попадает в эту последнюю категорию.

Как правило, программа-приложение не должна содержать собственную логику разбиения на полосы. Как уже говорилось, когда вы делаете вызовы функций GDI, в которых определяется страница графики, то модуль GDI обычно сохраняет эти вызовы в расширенном метафайле, а затем, перед проигрыванием этого метафайла в драйвере принтера использует разбиение на полосы, устанавливая регион отсечения. От программы-приложения это скрыто. Однако, при определенных условиях, задачу разбиения на полосы могло бы взять на себя приложение. Если в приложении применяется разбиение на полосы, то модуль GDI не создает промежуточного расширенного метафайла. Вместо этого драйверу принтера для каждой полосы передаются команды рисования. У такого подхода есть два достоинства:

- Можно немного увеличить скорость печати. В приложении нужно вызывать только те функции GDI, которые что-то рисуют в каждой конкретной полосе, что быстрее, чем проигрывание всего расширенного метафайла в драйвере устройства для каждой полосы.
- Можно уменьшить требуемое в обычном случае дисковое пространство. Если приложение печатает битовые образы, но при этом не выполняет собственного разбиения на полосы, то эти битовые образы должны храниться в том метафайле, который создает GDI. Эта ситуация может привести к тому, что метафайл оказывается больше файла с выходными данными для принтера, который в конце концов создает GDI.

Разбиение на полосы особенно важно при печати битовых образов, поскольку они занимают много места в метафайле. (Для печати битового образа требуется выбрать битовый образ в контексте памяти и, используя функции *BitBlt* или *StretchBlt*, записать его в контексте принтера.) Но кроме этого, разбиение на полосы приводит к дальнейшему усложнению процесса печати, как это станет понятно при создании программы PRINT4, последней версии нашей программы печати.

Разбиение на полосы

Чтобы программа сама осуществляла разбиение страницы на полосы, необходимо использовать еще неизвестную нам функцию GDI, которая называется *ExtEscape*. Имя этой функции подразумевает, что она игнорируется модулем GDI и, что она вызывает непосредственно драйвер принтера. В некоторых случаях именно так и происходит, но часто, при вызове функции *ExtEscape*, GDI делает и кое-что еще.

Обычно, синтаксис функции *ExtEscape* следующий:

```
iResult = ExtEscape(hdcPrinter, iEscapeCode, cbCountIn, psDataIn, cbCountOut, psDataOut);
```

Параметр *iEscapeCode* — это код подфункции, который задается с помощью идентификатора, определяемого в файле WINGDI.H. Значения остальных четырех параметров зависят от кода подфункции. Хотя два параметра *psDataIn* и *psDataOut* объявляются указателями на символьные строки, иногда они становятся указателями на структуры. Чтобы указатели стали указателями на символьные строки, используйте (LPSTR).

Не все подфункции *ExtEscape* реализованы во всех драйверах устройств. Фактически, функция *ExtEscape* разработана таким образом, чтобы она была открытой для производителей мониторов, которые могли бы определять свои собственные подфункции *ExtEscape* для обеспечения каких-либо необычных возможностей своих

устройств. Если данная подфункция не реализована, то функция *ExtEscape* всегда возвращает 0, а при наличии ошибки — отрицательное значение. Положительное значение означает, что вызов функции прошел успешно.

Нас интересует только одна подфункция функции *ExtEscape*, которая определяется идентификатором NEXTBAND. Эта подфункция предназначена для печати страницы выходных данных как набора прямоугольных полос. Для использования функции с идентификатором *ExtEscape*, необходимо определить переменную типа RECT:

```
RECT rect;
```

Вспомните, что в структуре RECT имеется четыре поля: *left, top, right* и *bottom*. Начинать печать каждой страницы необходимо с вызова функции *ExtEscape* с параметром подфункции NEXTBAND, передавая ее указатель на структуру *rect*. После возвращения функцией своего значения, в *rect* содержатся координаты первой полосы. Эти координаты всегда задаются в единицах измерения устройства (пикселях) независимо от текущего режима отображения в контексте принтера. Вы осуществляете вызовы функций GDI для печати этой полосы. Затем, для получения координат следующей полосы, снова вызывается функция *ExtEscape* с параметром подфункции NEXTBAND, и вы печатаете уже в этой полосе. Когда структура RECT, переданная в функцию *ExtEscape*, после возвращения функцией своего значения оказывается пустой (со всеми полями равными 0), значит страница готова.

Далее представлен примерный код для печати одной страницы. Для простоты, здесь не проверяются ошибки, которые могут возвращать функции печати, и не выполняется проверка значения *bUserAbort*:

```
StartPage(hdcPrn);                  // Начало печати страницы

ExtEscape(hdcPrn, NEXTBAND, 0, NULL, sizeof(rect),(LPSTR)&rect);


while( !IsRectEmpty(&rect) )
{
       [вызов функций GDI для печати полосы]

       ExtEscape(hdcPrn, NEXTBAND, 0, NULL, sizeof(rect),(LPSTR)&rect);
}

EndPage(hdcPrn);                    // Конец печати страницы
```

Каждый вызов функции *ExtEscape* с параметром подфункции NEXTBAND (за исключением первого) реализует задачу, похожую на ту, которую выполняет вызов функции *EndPage*: он сообщает модулю GDI и драйверу принтера о том, что вся полоса определена и, что ее пора сохранить в файле на диске (или, если спулер отключен, выдать на принтер). Этот фрагмент начинается с вызова функции *StartPage* и, после окончания цикла, заканчивается вызовом функции *EndPage*.

Проще всего показать процесс разбиения страницы на полосы для матричного принтера. Перед тем, как это сделать, необходимо понять разницу между "началом бумаги" (которое всегда представляет из себя ту часть бумаги, с которой начинается печать) и "началом страницы" (которое зависит от того, какой режим задан драйверу принтера, portrait или landscape).

В режиме portrait начало страницы то же, что и начало бумаги. Полосы создаются сверху вниз. Значение *rect.left* структуры RECT, которое устанавливается вызовом функции *ExtEscape* с параметром подфункции NEXTBAND, всегда равно 0, а значение *rect.right* всегда равно ширине области печати в пикселях (значение, полученное при вызове функции *GetDeviceCaps* с параметром HORZRES). Для первой полосы значение *rect.top* равно 0. Для каждой из последующих полос значение *rect.top* равно значению *rect.bottom* предыдущей полосы. Для последней полосы значение *rect.bottom* равно высоте области печати в пикселях. (См. рис. 15.11.)

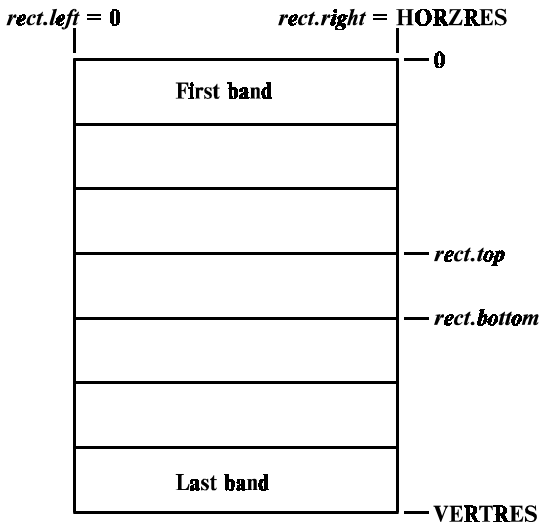


Рис. 15.11 Разбиение страницы на полосы для матричного принтера в режиме portrait

Таким образом, можно печатать в прямоугольной области с координатами от *rect.left* и *rect.top* до (но не включая) координат *rect.right* и *rect.bottom*. При вызове функции:

```
Rectangle(hdcPrn, rect.left, rect.top, rect.right, rect.bottom);
```

будет напечатан прямоугольник со сторонами, соответствующими границам полосы. (Вспомните, что правая и нижняя стороны прямоугольника фактически рисуются функцией *Rectangle* на один пиксель ближе точки, на которую указывают два последних параметра.)

В режиме landscape матричный принтер должен печатать документ повернутым, начиная с левой стороны страницы. Полосы располагаются на бумаге точно так же, но координаты прямоугольника отличаются, поскольку началом бумаги становится левая сторона страницы. В режиме landscape значение *rect.top* всегда равно 0, а значение *rect.bottom* постоянно равно высоте области печати в пикселях (значение, полученное при вызове функции *GetDeviceCaps* с параметром VERTRES). Для первой полосы, значение *rect.left* равно 0. Для последней полосы значение *rect.right* равно ширине области печати в пикселях. (См. рис. 15.12.)

Для лазерного принтера или плоттера процесс разбиения страницы на полосы может проходить иначе, поскольку порядок отправки на принтер или плоттер выходных данных может отличаться от последовательного, с начала страницы и далее до ее конца. Хотя на рис. 15.11 и 15.12 отражен вполне обычный случай, при программировании не следует предполагать, что разбиение страницы на прямоугольные полосы будет соответствовать этим двум образцам.

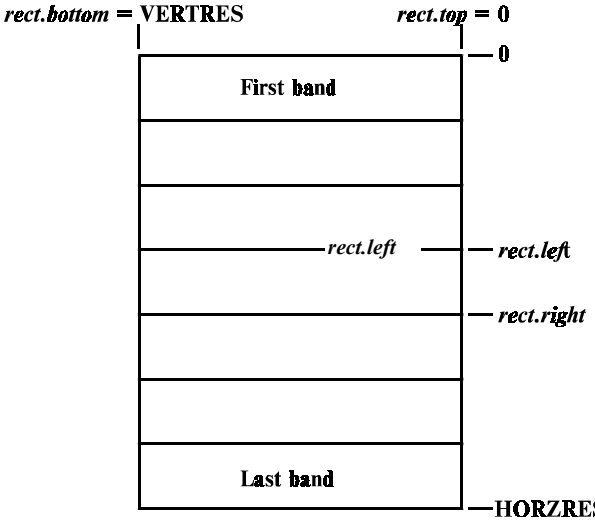


Рис. 15.12 Разбиение страницы на полосы для матричного принтера в режиме landscape

Разбиение выходных данных для принтера на полосы, как могло бы показаться, должно сильно усложнить программирование. Но, даже при использовании разбиения на полосы, совершенно не нужно включать в программу всю логику такого разбиения. Полоса — это регион отсечения. Вы можете делать любые вызовы функций GDI, а Windows проигнорирует все, за исключением того, что относится к внутренней области полосы. Это означает, что для каждой полосы все вызовы GDI можно делать как для страницы в целом.

Определить, требуется ли разбиение на полосы для конкретного драйвера, можно путем контроля бита RC_BANDING возвращаемого значения функции *GetDeviceCaps* с параметром RASTERCAPS. Как уже говорилось, эта информация относится только к GDI. Независимо от того, требуется для драйвера поддержка разбиения на полосы или нет, в модуле GDI всегда реализована возможность вызова функции *ExtEscape* с параметром подфункции NEXTBAND. Если для драйвера поддержка разбиения на полосы не требуется, возвращаемым значением первого для страницы вызова функции *ExtEscape* с параметром подфункции NEXTBAND станет прямоугольник, размер которого равен всей области печати. Возвращаемым значением повторного вызова функции *ExtEscape* с параметром подфункции NEXTBAND станет пустой прямоугольник.

Реализация разбиения страницы на полосы

В программе PRINT4, представленной на рис. 15.13, к логике печати программы PRINT3 добавлено разбиение страницы на полосы. Для программы PRINT4 также необходим представленный на рис. 15.8 файл PRINT.RC, а так же, как и для всех программ серии PRINT, представленный на рис. 15.5 файл PRINT.C.

PRINT4.MAK

```
#----------------------
# PRINT4.MAK make file
#----------------------

print4.exe : print.obj print4.obj print.res
     $(LINKER) $(GUIFLAGS) -OUT:print4.exe print.obj print4.obj \
     print.res $(GUILIBS) winspool.lib

print.obj : print.c
     $(CC) $(CFLAGS) print.c

print4.obj : print4.c
     $(CC) $(CFLAGS) print4.c

print.res : print.rc
     $(RC) $(RCVARS) print.rc
```

PRINT4.C

```
/*--------------------------------------
   PRINT4.C -- Printing with Banding
               (c) Charles Petzold, 1996
  --------------------------------------*/

#include <windows.h>

HDC  GetPrinterDC(void);              // in PRINT.C
void PageGDICalls(HDC, int, int);

HINSTANCE hInst;
char      szAppName[] = "Print4";
char      szCaption[] = "Print Program 4(Banding)";

BOOL   bUserAbort;
HWND   hDlgPrint;

BOOL CALLBACK PrintDlgProc(HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam)
     {
     switch(msg)
          {
          case WM_INITDIALOG :
               SetWindowText(hDlg, szAppName);
               EnableMenuItem(GetSystemMenu(hDlg, FALSE), SC_CLOSE,
                                                          MF_GRAYED);
               return TRUE;

          case WM_COMMAND :
               bUserAbort = TRUE;
               EnableWindow(GetParent(hDlg), TRUE);
```

```
               DestroyWindow(hDlg);
               hDlgPrint = 0;
               return TRUE;
          }
     return FALSE;
     }

BOOL CALLBACK AbortProc(HDC hdcPrn, int iCode)
     {
     MSG   msg;

     while(!bUserAbort && PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
          {
          if(!hDlgPrint || !IsDialogMessage(hDlgPrint, &msg))
               {
               TranslateMessage(&msg);
               DispatchMessage(&msg);
               }
          }
     return !bUserAbort;
     }

BOOL PrintMyPage(HWND hwnd)
     {
     static DOCINFO di     = { sizeof(DOCINFO), "Print4: Printing", NULL };
     BOOL           bError = FALSE;
     HDC            hdcPrn;
     RECT           rect;
     int            xPage, yPage;

     if(NULL ==(hdcPrn = GetPrinterDC()))
          return TRUE;

     xPage = GetDeviceCaps(hdcPrn, HORZRES);
     yPage = GetDeviceCaps(hdcPrn, VERTRES);

     EnableWindow(hwnd, FALSE);

     bUserAbort = FALSE;
     hDlgPrint = CreateDialog(hInst, "PrintDlgBox", hwnd, PrintDlgProc);

     SetAbortProc(hdcPrn, AbortProc);
     if(StartDoc(hdcPrn, &di) > 0 && StartPage(hdcPrn) > 0 &&
         ExtEscape(hdcPrn, NEXTBAND, 0, NULL, sizeof(rect),(LPSTR) &rect) > 0)
          {
          while(!IsRectEmpty(&rect) && !bUserAbort)
               {
               Rectangle(hdcPrn, rect.left, rect.top, rect.right,
                                                      rect.bottom);
               PageGDICalls(hdcPrn, xPage, yPage);

               if(ExtEscape(hdcPrn, NEXTBAND, 0, NULL, sizeof(rect),
                  (LPSTR) &rect) < 0)
                    {
                    bError = TRUE;          // If error, set flag and
                    break;                  //   break out of loop
                    }
               }
          }
     else
          bError = TRUE;

     if(!bError)
          {
          if(bUserAbort)
```

```
            AbortDoc(hdcPrn);
        else
            if(EndPage(hdcPrn))
                EndDoc(hdcPrn);
        }

    if(!bUserAbort)
        {
        EnableWindow(hwnd, TRUE);
        DestroyWindow(hDlgPrint);
        }

    DeleteDC(hdcPrn);

    return bError || bUserAbort;
    }
```

Рис. 15.13 Программа PRINT4

Программа PRINT4 отличается от программы PRINT3 только несколькими особенностями. Обратите внимание, что вместо прямоугольника, проходящего по границе страницы в целом, функция *Rectangle* печатает его для каждой полосы. Это позволяет вам увидеть то, как для конкретного принтера реализуется разбиение на полосы. Структура кода, выполняющего печать, такова:

```
if(StartDoc(hdcPrn, &di) > 0 && StartPage(hdcPrn) > 0 &&
    ExtEscape(hdcPrn, NEXTBAND, 0, NULL, sizeof(rect),(LPSTR) &rect) > 0)
{
       while(!IsRectEmpty(&rect) && !UserAbort)
       {
              [вызовы функций GDI]
              if(ExtEscape(hdcPrn, NEXTBAND, 0, NULL, sizeof(rect),(LPSTR)&rect) < 0)
              {
                     bError = TRUE;       // если ошибка, установить флаг и
                     break;               // выйти из цикла
              }
       }
}
else
       bError = TRUE;
```

Цикл *while* выполняется только в том случае, если прямоугольник не пуст и, если пользователь не отказался от печати в окне диалога. Программа PRINT4 должна проверять возвращаемое значение каждого вызова функции *ExtEscape* с параметром подфункции NEXTBAND и устанавливать флаг *bError*, если это возвращаемое значение отрицательно. Если ни один вызов функции *ExtEscape* не возвратит ошибки, то задание на печать должно быть закончено либо с помощью вызова функции *EndDoc*, либо прервано с помощью вызова функции *AbortDoc*. Этот код представлен ниже:

```
if(!bError)
{
       if(bUserAbort)
              AbortDoc(hdcPrn);
       else
       if(EndPage(hdcPrn))
              EndDoc(hdcPrn);
}
```

Принтер и шрифты

В главе 4 имеется программа JUSTIFY, в которой для вывода на экран отформатированного текста используются шрифты GDI. Программы, которые работают с форматированным текстом на экране, обычно имеют возможность печати этого текста. Действительно, в программах обработки текстов и настольных издательских системах экран используется главным образом только для предварительного просмотра выводимых на печать данных.

Шрифты типа TrueType предоставляют простейший путь точного представления текста на экране так, как если бы он был напечатан. Этот подход ограничивает возможности программы использованием только шрифтов типа True Type. Если выбор шрифта осуществляется с помощью функции *EnumFontFamilies*, то вы можете установить бит

TRUETYPE_FONTTYPE параметра *iFontType* функции обратного вызова. В этом случае будет использоваться окно списка только тех шрифтов, у которых установлен бит TRUETYPE_FONTTYPE.

Если в программе предоставить пользователю возможность выбора шрифта в диалоговом окне, которое отображается при вызове функции *ChooseFont* из библиотеки диалоговых окон общего пользования (как показано в программе JUSTIFY в главе 4), то для ограничения списка шрифтов только шрифтами типа TrueType можно включить в поле *Flags* структуры CHOOSEFONT флаг CF_TTONLY.

Однако, ограничение возможности выбора шрифта только шрифтами типа TrueType может не удовлетворить тех пользователей, кто вложил деньги в дополнительные принтерные шрифты, загружаемые или на картриджах. Такие шрифты не будут перечислены в предлагаемом пользователю списке. Для того, чтобы такие шрифты оказались в окне диалога, созданном с помощью функции *ChooseFont*, необходимо включить в поле *Flags* структуры CHOOSEFONT константу CF_PRINTERFONTS.

Необходимо также установить поле *hDC* структуры CHOOSEFONT равным описателю контекста принтера. Тогда список шрифтов, выводимых в окне списка будет ограничен внутренними шрифтами принтера, шрифтами типа TrueType и векторными шрифтами. (Чтобы исключить появление в списке векторных шрифтов, используйте флаг CF_NOVECTORFONTS.)

Если пользователю дается возможность выбора принтерных шрифтов, то вид шрифтов на экране будет только примерно соответствовать тому, что будет напечатано. Например, если для принтера предлагается 15-точечный шрифт Zapf Chancery, он может быть апроксимирован 15-точечным шрифтом типа TrueType, но при этом ширина символа шрифта типа TrueType будет отличаться от ширины символа шрифта Zapf Chancery. Даже для шрифтов типа TrueType ширина символа на экране и принтере может отличаться на величину ошибки округления, возникающую из-за отличий экрана и принтера.

Короче говоря, при написании программы, в которой на экране должен выводиться отформатированный текст, предназначенный для печати, придется сделать дополнительные усилия. Вот несколько указаний для начала.

При выводе отформатированного текста на экране желательно, чтобы текст выглядел так, как он в конце концов будет напечатан. Для определения размера бумаги и размера области печати можно использовать функции *GetDeviceCaps* и *DeviceCapabilities*. Например, если ширина бумаги равна 8,5 дюймов, и пользователь задает размер левого и правого отступов по одному дюйму, то желательно выводить текст в область экрана шириной 6,5 дюймов. Для такого вывода подходит режим отображения "logical twips", о котором говорилось в главе 4. Однако, будьте осторожны. Если пользователь выбирает 15-точечный шрифт, который поддерживает принтер, этот шрифт пришлось бы приспособить к ширине области экрана и подобрать вместо него, скажем, 14-точечный экранный шрифт, но такой 14-точечный шрифт нельзя использовать для определения того, сколько может занять одна напечатанная 15-точечным шрифтом строка. Это надо определять на основе принтерного шрифта. Подобным образом принтерный шрифт необходимо использовать для определения количества строк на странице.

Для форматирования текста на экране понадобится описатель контекста экрана и описатель информационного контекста принтера. До того, как фактически начать печатать, описатель контекста принтера не нужен. Проделайте следующие шаги:

1. В структуру логического шрифта объедините имя шрифта, его размер и выбранные пользователем атрибуты, и этот логический шрифт выберите в информационный контекст принтера.

2. Для определения действительных размера и характеристик выбранного шрифта для принтера используйте вызов функции *GetTextMetrics*. Для получения имени шрифта используйте вызов функции *GetTextFace*.

3. Информацию, полученную на втором шаге, используйте для создания другой структуры логического шрифта, основанного на размере и характеристиках принтерного шрифта, выберите этот логический шрифт в контекст экрана. Теперь шрифт, выбранный в контексте экрана, приблизительно соответствует шрифту, выбранному в информационном контексте принтера.

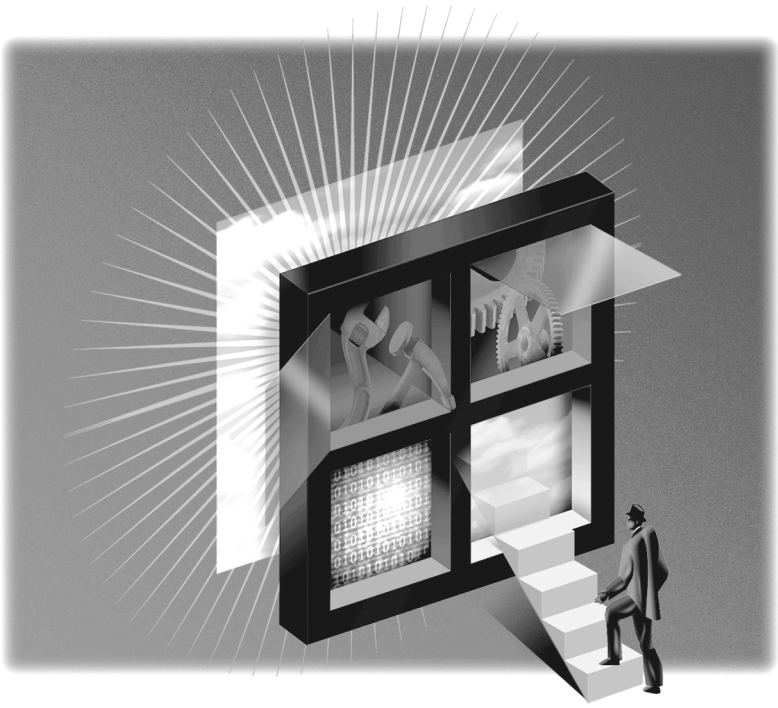
4. При отображении текста на экране следуйте общей схеме, которая использовалась в функции *Justify* программы JUSTIFY. Однако, при этом выполните функции *GetTextExtent* и *SetTextJustification*, используя информационный контекст принтера, но остановитесь перед функцией *TextOut*. Такой подход позволяет определить количество символов в строке и число строк на странице.

5. Когда для принтера сформирована каждая строка текста, то вызовите функцию *GetTextExtent* и (возможно) *SetTextJustification*, используя контекст экрана. Затем, для вывода строки на экран используйте вызов функции *TextOut*.

Для печати текста вы, вероятно, будете использовать код, похожий на код из файла POPPRINT.C в сочетании с логикой функции *Justify* файла JUSTIFY.C. Получите контекст принтера и снова выполните функции *GetTextExtent* и *SetTextJustification*, на этот раз для печати каждой строки, используя функцию *TextOut*.

Часть V

Связи и обмен данными



Глава 16 Буфер обмена

16

Буфер обмена (clipboard) в Windows дает возможность передавать данные от одной программы к другой. Это относительно простой механизм, не требующий больших добавлений ни к программе, которая помещает данные в буфер обмена, ни к программе, которая в дальнейшем их получает. В Windows 95 имеется программа просмотра, в которой показывается текущее состояние буфера обмена. В большинстве программ, имеющих дело с документами или другими данными, имеется меню Edit с опциями Cut, Copy и Paste. Если пользователь выбирает опции Cut или Copy, то программа передает данные в буфер обмена. Эти данные находятся в специальном формате, например в виде текста, битового образа или метафайла. Если пользователь выбирает в меню опцию Paste, программа проверяет имеются ли в буфере обмена данные в том формате, который программа может использовать, и если да, то эти данные передаются из буфера обмена в программу.

Программы не должны передавать данные в буфер обмена и получать их оттуда без совершенно точно определенной инструкции пользователя. Например, пользователь, выполняющий в одной программе операцию Cut или Copy, должен быть уверен, что эти данные будут оставаться в буфере обмена вплоть до следующей операции Cut или Copy.

Простое использование буфера обмена

Мы начнем с анализа программы для передачи данных в буфер обмена (Cut и Copy) и получения данных из буфера обмена (Paste).

Стандартные форматы данных буфера обмена

Windows поддерживает стандартные форматы данных буфера обмена, идентификаторы которых находятся в заголовочных файлах Windows. Наиболее часто используемыми из них являются:

- CF_TEXT — оканчивающаяся нулем группа символов из набора символов ASCII, в конце каждой строки которой имеются символы возврата каретки и перевода строки. Это простейший формат данных буфера обмена. Передаваемые в буфер обмена данные хранятся в области оперативной памяти, а передаются они с помощью описателя этой области памяти. Эта область памяти становится неотъемлемой частью буфера обмена, и программа, создавшая этот блок памяти, больше не должна его использовать.

- CF_BITMAP — зависящий от устройства битовый образ. Битовый образ передается в буфер обмена с помощью описателя битового образа. И в этом случае программа, после передачи битового образа в буфер обмена, не должна его больше использовать.

- CF_METAFILEPICT — "картинка метафайла" (metafile picture). Это не совсем то же самое, что метафайл (описанный в главе 4). Скорее это метафайл, содержащий дополнительную информацию в виде небольшой структуры типа METAFILEPICT. Программа передает картинку метафайла в буфер обмена с помощью описателя области памяти, содержащего эту структуру. В структуре METAFILEPICT имеется четыре поля: *mm* (LONG), режим отображения метафайла; *xExt* (LONG) и *yExt* (LONG), ширина и высота образа метафайла; *hMF* (HMETAFILE), описатель метафайла. (Далее в этой главе более подробно будет рассказано о полях *xExt* и *yExt*.) После того, как программа передаст в буфер обмена картинку метафайла, она не должна продолжать использовать ни область памяти, содержащую структуру METAFILEPICT, ни описатель метафайла, поскольку и тем, и другим теперь будет управлять Windows 95.

- CF_ENHMETAFILE — описатель расширенного метафайла (описанного в главе 4). В этом формате структура METAFILEPICT не используется.

- CF_SYLK — блок памяти, содержащий данные в формате Microsoft Symbolic Link (SYLK). Этот формат используется для обмена данными между программами Multiplan, Chart и Excel, созданными корпорацией

Microsoft. Это ASCII-формат, в котором каждая строка завершается символами возврата каретки и перевода строки.

- CF_DIF — блок памяти, содержащий данные в формате Data Interchange Format (DIF). Этот формат предложен компанией Software Arts для передачи данных в программу электронных таблиц VisiCalc. Теперь этот формат находится под управлением корпорации Lotus. Это также ASCII-формат, в котором каждая строка завершается символами возврата каретки и перевода строки.

Форматы CF_SYLK и CF_DIF концептуально похожи на формат CF_TEXT. Однако, символьные строки в форматах SYLK или DIF необязательно заканчиваются нулевым символом, поскольку эти форматы определяют конец данных.

- CF_TIFF — блок памяти, содержащий данные в формате Tag Image File Format (TIFF). Этот формат предложен корпорациями Microsoft, Aldus и компанией Hewlett-Packard в сотрудничестве с некоторыми производителями аппаратуры. Этот формат (описывающий данные битовых изображений) поддерживается компанией Hewlett-Packard.
- CF_OEMTEXT — блок памяти, содержащий текстовые данные (как в формате CF_TEXT), но из набора символов OEM.
- CF_DIB — блок памяти, определяющий независящий от устройства битовый образ (описывается в главе 4). Блок памяти начинается со структуры BITMAPINFO, за которой следуют биты битового образа.
- CF_PALETTE — описатель цветовой палитры. Обычно используется в сочетании с CF_DIB для определения цветовой палитры, используемой битовым образом.

Передача текста в буфер обмена

Предположим, что в буфер обмена необходимо передать символьную строку, и, что имеется указатель (*pString*) на эту строку. Передать необходимо *iLength* байт этой строки.

Во-первых, используя функцию *GlobalAlloc*, выделяем перемещаемый блок памяти размером *iLength*, резервируя место для символа окончания NULL:

```
hGlobalMemory = GlobalAlloc(GHND, iLength + 1);
```

Значение *hGlobalMemory* будет равно NULL, если память выделена быть не может. Если выделение памяти прошло удачно, то для получения указателя на выделенный блок необходимо его зафиксировать:

```
pGlobalMemory = GlobalLock(hGlobalMemory);
```

Теперь копируем символьную строку в зафиксированную область памяти:

```
for(i = 0; i < iLength; i ++)
     *pGlobalMemory++ = *pString++;
```

В конец строки NULL-символ помещать не надо, поскольку установка флага GHND в функции *GlobalAlloc* обнуляет всю выделяемую область памяти. После этого снимаем фиксацию области памяти:

```
GlobalUnlock(hGlobalMemory);
```

Теперь у нас имеется описатель области памяти, в которой находится оканчивающийся нулевым символом текст. Для помещения его в буфер обмена открываем и очищаем ее:

```
OpenClipboard(hwnd);
EmptyClipboard();
```

Используя идентификатор CF_TEXT, передаем буферу обмена описатель области памяти и закрываем буфер обмена:

```
SetClipboardData(CF_TEXT, hGlobalMemory);
CloseClipboard();
```

Готово.

Ниже приведено несколько правил, регулирующих этот процесс:

- Вызовы функций *OpenClipboard* и *CloseClipboard* делаются во время обработки одного сообщения. Буфер обмена не оставляют открытой дольше, чем это необходимо.
- Буферу обмена не передается описатель зафиксированной области памяти.
- После вызова функции *SetClipboardData* область памяти использовать нельзя. Она больше не принадлежит программе, и описатель следует считать недействительным. Если доступ к данным по-прежнему необходим, нужно сделать их копию или считать их из буфера обмена (как это описано в следующем

разделе). Между вызовами функций *SetClipboardData* и *CloseClipboard* можно ссылаться на выделенную область памяти, но при этом нужно использовать описатель, возвращаемый функцией *SetClipboardData*. Перед вызовом функции *CloseClipboard* необходимо освободить этот описатель.

Получение текста из буфера обмена

Получение текста из буфера обмена лишь чуть-чуть сложнее передачи текста в нее. Сначала необходимо определить, действительно ли в буфере обмена содержатся данные в формате CF_TEXT. Один из простейших способов сделать это заключается в вызове функции *IsClipboardFormatAvailable*:

```
bAvailable = IsClipboardFormatAvailable(CF_TEXT);
```

Функция возвращает TRUE (ненулевое значение), если в буфере обмена находятся данные в формате CF_TEXT. В программе POPPAD2 в главе 10 эта функция используется для определения того, нужно ли делать пункт Paste меню Edit доступным или недоступным. Функция *IsClipboardFormatAvailable* — это одна из нескольких функций буфера обмена, которую можно использовать без предварительного открытия буфера обмена. Однако, если позже для получения этого текста открыть буфер обмена, необходимо снова проверить (используя эту же функцию или другие способы) по-прежнему ли в буфере обмена находятся данные в формате CF_TEXT.

Для получения данных из буфера обмена сначала открываем ее:

```
OpenClipboard(hwnd);
```

Затем получаем описатель области памяти, содержащей текст:

```
hClipMemory = GetClipboardData(CF_TEXT);
```

Этот описатель будет равен NULL, если в буфере обмена отсутствуют данные в формате CF_TEXT. Это второй способ определения наличия текста в буфере обмена. Если возвращаемое значение функции *GetClipboardData* равно NULL, буфер обмена следует закрыть, ничего с ней не делая.

Описатель, возвращаемый функцией *GetClipboardData*, не принадлежит вашей программе — он принадлежит буферу обмена. Описателем можно пользоваться только между вызовами функций *GetClipboardData* и *CloseClipboard*. Нельзя освободить этот описатель или изменить данные, на которые он ссылается. Если доступ к этим данным необходим, то нужно сделать копию этой области памяти.

Далее предлагается один из способов копирования данных в программу. Сначала необходимо выделить область памяти точно такого же размера, как область памяти буфера обмена:

```
pMyCopy =(char *) malloc(GlobalSize(hClipMemory));
```

Теперь, для получения указателя на область памяти буфера обмена, необходимо зафиксировать описатель области памяти буфера обмена:

```
pClipMemory = GlobalLock(hClipMemory);
```

Теперь можно копировать данные:

```
strcpy(pMyCopy, pClipMemory);
```

Или можно использовать какой-нибудь простой код на языке C:

```
while(*pMyCopy++ = *pClipMemory++);
```

Перед закрытием буфера обмена необходимо снять фиксацию области памяти:

```
GlobalUnlock(hClipMemory);
CloseClipboard();
```

Итак, получен указатель *pMyCopy*, который позже можно зафиксировать для получения доступа к этим данным.

Открытие и закрытие буфера обмена

В любой момент времени только одна программа может держать буфер обмена открытой. Цель вызова функции *OpenClipboard* состоит в предотвращении возможности изменения содержимого буфера обмена в то время, пока программа ее использует. Типом возвращаемого значения функции *OpenClipboard* является BOOL, показывающее, была ли буфер обмена открыта успешно. Она не будет открыта, если в другом приложении она не закрыта. Если в каждой программе буфер обмена будет аккуратно открываться и закрываться настолько быстро, насколько это возможно, то вы никогда не столкнетесь с проблемой невозможности ее открыть.

Однако, в мире неаккуратных программ и вытесняющей многозадачности такие проблемы могут возникать. Даже если программа не теряет фокус ввода между временем помещения чего-нибудь в буфер обмена и тем временем, когда пользователь выбирает опцию Paste, не надейтесь, что в буфере обмена находится то, что было туда помещено. В это время доступ к буферу обмена *мог* иметь фоновый процесс.

Кроме этого, опасайтесь более хитрых проблем, связанных с окнами сообщений: если нельзя выделить достаточно памяти, чтобы скопировать что-нибудь в буфер обмена, то можно вывести на экран окно сообщения. Однако, если это окно сообщения не является системным модальным окном, пользователь может переключиться на другое приложение, пока окно сообщений остается на экране. Необходимо либо делать окно сообщения системы модальным, либо закрывать буфер обмена перед выводом на экран окна сообщения.

Также могут возникнуть проблемы, если оставить буфер обмена открытой при выводе на экран окна диалога. Поля редактирования в окне диалога используют буфер обмена для вырезания и вставки текста.

Использование буфера обмена с битовыми образами

При использовании формата CF_BITMAP при вызове функции *SetClipboardData* буферу обмена передается описатель зависящего от устройства битового образа. Возвращаемым значением функции *GetClipboardData* является описатель этого зависящего от устройства битового образа.

При наличии описателя битового образа, копировать этот битовый образ в буфер обмена можно с помощью следующего простого кода:

```
OpenClipboard(hwnd);

EmptyClipboard();

SetClipboardData(CF_BITMAP, hBitmap);

CloseClipboard();
```

Также легко получить доступ к битовому образу:

```
OpenClipboard(hwnd);

hBitmap = GetClipboardData(CF_BITMAP);
```

Если в буфере обмена битового образа нет, *hBitmap* будет равен NULL. Чтобы сделать копию битового образа, для определения размера в пикселях и цветовой организации битового образа используйте функцию *GetObject*:

```
GetObject(hBitmap, sizeof(BITMAP),(PSTR) &bm);
```

Теперь, используя вызов функции *CreateBitmapIndirect*, вы можете создать свой собственный битовый образ того же размера и цветовой организации:

```
hMyBitmap = CreateBitmapIndirect(&bm);
```

Выберите оба битовых образа в контексты памяти, создаваемые с помощью функции *CreateCopatibleDC*:

```
hdcMemSrc = CreateCompatibleDC(hdc);

SelectObject(hdcMemSrc, hBitmap);

hdcMemDst = CreateCompatibleDC(hdc);

SelectObject(hdcMemDst, hMyBitmap);
```

Для копирования битового образа из буфера обмена используйте функцию *BitBlt*:

```
BitBlt(hdcMemDst, 0, 0, bm.bmWidth, bm.bmHeight, hdcMemSrc, 0, 0, SRCCOPY);
```

Осталась только очистка, включающая в себя удаление контекстов памяти и закрытие буфера обмена:

```
DeleteDC(hdcMemSrc);
DeleteDC(hdcMemDst);
CloseClipboard();
```

Не удаляйте исходный битовый образ, поскольку он принадлежит буферу обмена.

Программы, которые работают с независящими от устройства битовыми образами (DIB), также могут передавать эти битовые образы в буфер обмена и получать их оттуда. Если в программе имеется глобальный описатель области памяти, в которой находится определение битового образа в формате DIB, то можно просто передать этот описатель функции *SetClipboardData* с идентификатором CF_DIB. После закрытия буфера обмена область памяти с определением битового образа больше не принадлежит вашей программе. Если необходимо сохранить копию области памяти с определением битового образа в формате DIB, можно сделать копию для буфера обмена, выделяя область памяти такого же размера и копируя туда данные DIB. Получение битового образа в формате DIB из буфера обмена похоже на получение текста из буфера обмена, за исключением того, что данные не заканчиваются нулем.

Если в программе необходимо сохранить собственную копию битового образа (либо в формате, зависящем от устройства, либо в формате, не зависящем от устройства), то после передачи копии в буфер обмена, для хранения этих двух копий вам возможно придется использовать большое пространство памяти. В этом случае, чтобы

избежать нехватки ресурсов памяти, вероятно, понадобится использовать прием, который называется "отложенное исполнение" (delayed rendering). Этот прием рассматривается далее в этой главе.

Метафайл и картина метафайла

Использовать буфер обмена для передачи расширенных метафайлов достаточно просто. Возвращаемым значением функции *GetClipboardData* является описатель расширенного метафайла; этот описатель метафайла также использует функция *SetClipboardData*. Необходима копия метафайла? Используйте функцию *CopyEnhMetaFile*. Как уже говорилось в главе 4 и демонстрировалось в главе 15, основное улучшение формата расширенного метафайла по сравнению с его старым форматом — это то, что вы можете легко определить размеры образа и вывести его в окно вашей программы соответствующим образом.

Старые, нерасширенные метафайлы создавали непростую проблему. Как можно определить величину изображения метафайла перед тем, как его проиграть? Пока вы не проанализируете весь метафайл, это невозможно.

Более того, если программа получает из буфера обмена метафайл старого типа, с ним можно работать более гибко, если метафайл был создан для проигрывания в режиме отображения MM_ISOTROPIC или MM_ANISOTROPIC. Программа, которая затем получает метафайл, может промасштабировать образ, просто установив перед запуском метафайла протяженность области вывода (viewport extent). Но если режим отображения устанавливается в MM_ISOTROPIC или MM_ANISOTROPIC внутри метафайла, то программа, получающая такой метафайл, зависнет. Программа может делать вызовы функций GDI только до или после того, как проигрывается метафайл. В процессе проигрывания метафайла делать вызовы функций GDI нельзя.

Для решения этих проблем описатели метафайлов старого типа не прямо помещаются в буфер обмена и извлекаются из нее, а с помощью других программ. Описатель метафайла становится частью "картинки метафайла", которая представляет собой структуру типа METAFILEPICT. Эта структура дает возможность программе, которая получает картину метафайла из буфера обмена, перед проигрыванием метафайла, устанавливать режим отображения и протяженность области вывода.

Длина структуры типа METAFILEPICT равна 16 байт, и в ней имеется четыре поля: *mm*, режим отображения метафайла; *xExt* и *yExt*, ширина и высота образа метафайла; и *hMF*, описатель метафайла. Для всех режимов отображения, за исключением MM_ISOTROPIC и MM_ANISOTROPIC, значения *xExt* и *yExt* — это размеры образа метафайла в единицах измерения, соответствующих режиму отображения, заданному в поле *mm*. Обладая такой информацией, программа, которая копирует структуру картины метафайла из буфера обмена, может определить размеры экранного пространства, которое займет метафайл при проигрывании. Программа, создающая метафайл, может установить эти размеры, присвоив им максимальные значения координат *x* и *y*, которые используются в функциях рисования GDI, входящих в состав этого метафайла.

Для режимов отображения MM_ISOTROPIC и MM_ANISOTROPIC назначение полей *xExt* и *yExt* структуры METAFILEPICT отличается. Из главы 4 известно, что программа имеет режим отображения MM_ISOTROPIC или MM_ANISOTROPIC, когда в функциях GDI необходимо использовать произвольные логические единицы измерения, вне зависимости от измеряемого размера образа. Программа работает в режиме отображения MM_ISOTROPIC, когда хочет сохранять коэффициент растяжения независимо от размера области вывода, а режим отображения MM_ANISOTROPIC, когда нет необходимости заботиться о коэффициенте растяжения. Из главы 4 также следует, что после того, как программа устанавливает режим отображения MM_ISOTROPIC или MM_ANISOTROPIC, как правило делаются вызовы функций *SetWindowExtEx* и *SetViewportExtEx*. Функция *SetWindowExtEx* использует логические единицы измерения для задания тех единиц, которые будет применять программа при рисовании. Функция *SetViewportExtEx* использует единицы измерения устройства, основанные на размере области вывода (например, размере рабочей области окна).

Если программа создает метафайл с режимом отображения MM_ISOTROPIC или MM_ANISOTROPIC для буфера обмена, то в самом метафайле не должно быть вызова функции *SetViewportExtEx*, поскольку единицы измерения устройства в таком вызове были бы основаны на размерах области вывода программы, создающей метафайл, а не на размерах области вывода программы, которая считывает метафайл из буфера обмена и проигрывает его. Вместо этого, значения *xExt* и *yExt* призваны помочь программе, получающей метафайл из буфера обмена, в установке соответствующей протяженности области вывода для проигрывания метафайла. Но в самом метафайле содержится вызов функции для установки протяженности окна, если режимом отображения является MM_ISOTROPIC или MM_ANISOTROPIC. Координаты функций рисования GDI внутри метафайла основаны на этой протяженности окна.

Программа, в которой создаются метафайл и картина метафайла, должна соответствовать следующим правилам:

- Поле *mm* структуры METAFILEPICT устанавливается для задания режима отображения.
- Для режимов отображения, отличных от MM_ISOTROPIC и MM_ANISOTROPIC, поля *xExt* и *yExt* устанавливаются равными ширине и высоте образа в единицах, соответствующих режиму отображения, указанному в поле *mm*. Для метафайлов, проигрываемых в режиме MM_ISOTROPIC или

MM_ANISOTROPIC, требуется несколько больше усилий. Для MM_ANISOTROPIC поля *xExt* и *yExt* устанавливаются в 0, если программа не предлагает ни размера, ни коэффициента растяжения образа. Для режимов MM_ISOTROPIC или MM_ANISOTROPIC положительные значения полей *xExt* и *yExt* показывают предлагаемые ширину и высоту изображения в единицах, равных 0.01 мм (единицы режима MM_HIMETRIC). Для MM_ISOTROPIC отрицательные значения полей *xExt* и *yExt* показывают предлагаемый коэффициент растяжения образа, а не его предлагаемый размер.

- Для режимов отображения MM_ISOTROPIC и MM_ANISOTROPIC в самом метафайле содержатся вызовы функций *SetWindowExtEx* и (возможно) *SetWindowOrgEx*. Таким образом, программа, в которой создается метафайл, вызывает эти функции в контексте метафайла. Как правило, вызовов функций *SetMapMode*, *SetViewportExtEx* и *SetWindowOrgEx* в метафайле не содержатся.
- Метафайл должен располагаться в оперативной, а не в дисковой памяти.

Далее представлен образец программы создания метафайла и его копирования в буфер обмена. Если метафайл использует режим отображения MM_ISOTROPIC или MM_ANISOTROPIC, то первый вызов функции в метафайле должен устанавливать протяженность окна. (Протяженность окна фиксирована в других режимах отображения.) Независимо от режима отображения, начало координат окна также может быть установлено:

```
hdcMeta = CreateMetaFile(NULL);

SetWindowExtEx(hdcMeta, ...);

SetWindowOrgEx(hdcMeta, ...);
```

Координаты функций рисования метафайла основываются на протяженности и начале координат окна. После того, как программа использует вызовы функций GDI для рисования в контексте метафайла, для получения описателя метафайла он закрывается:

```
hmf = CloseMetaFile(hdcMeta);
```

Программа также должна определить указатель на структуру типа METAFILEPICT и выделить область памяти для этой структуры:

```
HGLOBAL       hGMem;
LPMETAFILEPICT pMFP;
[другие строки программы]

hGMem = GlobalAlloc(GHND, sizeof(METAFILEPICT));


pMFP =(LPMETAFILEPICT) GlobalLock(hGMem);
```

Далее программа устанавливает четыре поля этой структуры:

```
pMFP -> mm    = MM_...      ;
pMFP -> xExt  = ...  ;
pMFP -> yExt  = ...  ;
pMFP -> hMF   = hmf  ;


GlobalUnlock(hGMem);
```

Затем программа передает в буфер обмена область памяти, в которой находится структура картинки метафайла:

```
OpenClipboard(hwnd);

EmptyClipboard();

SetClipboardData(CF_METAFILEPICT, hGMem);

CloseClipboard();
```

После вызовов этих функций, описатель *hGMem* (области памяти, в которой находится структура картинки метафайла), и описатель *hmf* (самого метафайла) становятся недействительными для программы, их создавшей.

Теперь более трудная часть. Когда программа получает метафайл из буфера обмена и проигрывает этот метафайл, должны быть сделаны следующие шаги:

1. Программа использует поле *mm* структуры картинки метафайла для задания режима отображения.
2. Для режимов отображения, отличных от MM_ISOTROPIC и MM_ANISOTROPIC, программа использует значения *xExt* и *yExt* для задания прямоугольника отсечения или просто для определения размера образа. Для режимов отображения MM_ISOTROPIC и MM_ANISOTROPIC программа использует *xExt* и *yExt* для задания протяженности области вывода.
3. Затем программа проигрывает метафайл.

Ниже приведен код, реализующий эти действия. Сначала вы открываете буфер обмена, получаете описатель структуры картины метафайла, и фиксируете его:

```
OpenClipboard(hwnd);
hGMem = GetClipboardData(CF_METAFILEPICT);

pMFP =(LPMETAFILEPICT) GlobalLock(hGMem);
```

Затем можно сохранить атрибуты текущего контекста устройства и установить режим отображения, заданный в поле *mm* структуры:

```
SaveDC(hdc);

SetMappingMode(pMFP -> mm);
```

Если режим отображения — не MM_ISOTROPIC или MM_ANISOTROPIC, можно задать прямоугольник отсечения с помощью значений *xExt* и *yExt*. Поскольку эти значения заданы в логических единицах, необходимо использовать функцию *LPtoDP* для преобразования координат в единицы измерения устройства для прямоугольника отсечения. Или можно просто сохранить значения, чтобы выяснить размер битового образа.

Для режимов отображения MM_ISOTROPIC или MM_ANISOTROPIC используйте *xExt* и *yExt* для задания протяженности области вывода. Ниже показана одна из возможных функций для выполнения этой задачи. В этой функции предполагается, что значения *cxClient* и *cyClient* отражают высоту и ширину в пикселях области, в которой должен появиться метафайл, если предлагаемый размер не задается с помощью значений *xExt* и *yExt*.

```
void PrepareMetaFile(HDC hdc, LPMETAFILEPICT pmfp, int cxClient, int cyClient)
{
       int xScale, yScale, iScale;

       SetMapMode(hdc, pmfp -> mm);


       if(pmfp -> mm == MM_ISOTROPIC || pmfp -> mm == MM_ANISOTROPIC)
       {
              if(pmfp -> xExt == 0)
                     SetViewportExtEx(hdc, cxClient, cyClient, NULL);
              else
if(pmfp -> xExt > 0)
                     SetViewportExtEx(hdc,
                            pmfp -> xExt * GetDeviceCaps(hdc, HORZRES) /
                            GetDeviceCaps(hdc, HORZSIZE) /100,
                            pmfp -> yExt * GetDeviceCaps(hdc, VERTRES) /
                            GetDeviceCaps(hdc, VERTSIZE) /100,
                            NULL);
              else
if(pmfp -> xExt < 0)
              {
                     xScale = 100 * cxClient * GetDeviceCaps(hdc, HORZSIZE) /
                            GetDeviceCaps(hdc, HORZRES) / -pmfp -> xExt;
                     yScale = 100 * cyClient * GetDeviceCaps(hdc,
                                               VERTSIZE) /
                            GetDeviceCaps(hdc, VERTRES) / -pmfp -> yExt;
                     iScale = min(xScale, yScale);

                     SetViewportExtEx(hdc,
                            - pmfp -> xExt * iScale * GetDeviceCaps(hdc, HORZRES) /
                            GetDeviceCaps(hdc, HORZSIZE) / 100,
                            - pmfp->yExt * iScale * GetDeviceCaps(hdc, VERTRES) /
                            GetDeviceCaps(hdc, VERTSIZE) / 100,
                            NULL);
              }
       }
}
```

В этой программе предполагается, что и *xExt*, и *yExt* равны 0, больше 0 или меньше 0. Если эти значения равны 0, значит ни размер, ни коэффициент растяжения не предлагаются. Протяженность области вывода устанавливается в соответствии с размерами области, в которой необходимо вывести метафайл. Положительные значения *xExt* и *yExt* являются предлагаемыми размерами битового образа в единицах, равных 0.01 мм. Функция *GetDeviceCaps* помогает определению числа пикселей в 0.01 мм, и это значение умножается на значения протяженностей в структуре картины метафайла. Отрицательные значения *xExt* и *yExt* показывают предлагаемый коэффициент растяжения, а не предлагаемый размер. Величина *iScale* рассчитывается на основе коэффициента растяжения размеров в миллиметрах, соответствующих значениям *cxClient* и *cyClient*. Этот масштабирующий множитель используется затем для задания протяженности области вывода в пикселях.

После окончания этой работы, можно, если нужно, установить начало координат области вывода, проиграть метафайл и вернуться к исходному контексту устройства:

```
PlayMetaFile(pMFP -> hMF);
RestoreDC(hdc, -1);
```

Теперь снимите фиксацию области памяти и закройте буфер обмена:

```
GlobalUnlock(hGMem);

CloseClipboard();
```

Более сложное использование буфера обмена

При использовании текста и графики было показано, что, после того как данные подготовлены, их передача в буфер обмена требует вызовов четырех функций:

```
OpenClipboard(hwnd);

EmptyClipboard();

SetClipboardData(iFormat, hHandle);

CloseClipboard();
```

Для получения доступа к этим данным требуются вызовы трех функций:

```
OpenClipboard(hwnd);

hHandle = GetClipboardData(iFormat);
[другие строки программы]

CloseClipboard();
```

Можно сделать копию данных буфера обмена или использовать их каким-то другим способом в промежутке между вызовами функций *GetClipboard* и *CloseClipboard*. В большинстве случаев такой подход позволит решить все поставленные задачи, но данные буфера обмена можно также использовать и более экзотическим образом.

Использование нескольких элементов данных

При открытии буфера обмена, для того чтобы поместить в нее данные, необходимо вызвать функцию *EmtyClipboard*, чтобы сигнализировать Windows о необходимости освободить или удалить содержимое буфера обмена. Нельзя что-то добавить к имеющемуся содержимому буфера обмена. Поэтому можно сказать, что в любой момент времени в буфере обмена хранится только один элемент данных.

Однако, в промежутке между вызовами функций *GetClipboard* и *CloseClipboard* можно несколько раз вызвать функцию *SetClipboardData* и при этом каждый раз использовать другой формат данных для буфера обмена. Например, если в буфере обмена нужно хранить небольшую строку текста, то можно создать контекст метафайла и записать этот текст в метафайл. Можно также создать достаточно большой для хранения символьной строки битовый образ, выбрать битовый образ в контексте памяти и записать строку в битовый образ. В таком случае символьная строка станет доступной не только для программ, которые могут считывать из буфера обмена текстовый формат, но также для программ, которые считывают из буфера обмена битовые образы и метафайлы. Более того, если перед выводом текста выбрать другой шрифт в контексте метафайла или контексте памяти, то программы, считывающие битовые образы или метафайлы, будут использовать строку, выведенную этим новым шрифтом. (Конечно, эти программы были бы не в состоянии определить, что битовый образ или метафайл фактически содержат символьную строку.)

При желании записать в буфер обмена несколько описателей, используйте для каждого из них вызов функции *SetClipboardData*:

```
OpenClipboard(hwnd);

EmptyClipboard();
```

```
SetClipboardData(CF_TEXT, hGMemText);

SetClipboardData(CF_BITMAP, hBitmap);

SetClipboardData(CF_METAFILEPICT, hGMemMFP);

CloseClipboard();
```

Пока данные этих трех форматов находятся в буфере обмена, возвращаемым значением функции *IsClipboardFormatAvailable* с аргументами CF_TEXT, CF_BITMAP или CF_METAFILEPICT будет TRUE. Получить эти описатели можно с помощью следующих вызовов:

```
hGMemText = GetClipboardData(CF_TEXT);
```

или:

```
hBitmap = GetClipboardData(CF_BITMAP);
```

или:

```
hGMemMFP = GetClipboardData(CF_METAFILEPICT);
```

В следующий раз, когда программа вызывает функцию *EmptyClipboard*, Windows освобождает или удаляет все три описателя, сохраняемые в буфере обмена, также как и метафайл, который является частью структуры METAFILEPICT.

Программа может определить все хранящиеся в буфере обмена форматы при первом открытии буфера обмена и последующем вызове функции *EnumClipboardFormats*. Начинать надо с установки в 0 переменной *iFormat*:

```
iFormat = 0;

OpenClipboard(hwnd);
```

Теперь последующий вызов функции *EnumClipboardFormats* начинается со значения 0. Возвращаемым значением функции будет положительное значение *iFormat* для каждого формата, который в данный момент имеется в буфере обмена. Когда возвращаемое значение равно 0, происходит выход из цикла:

```
while(iFormat = EnumClipboardFormats(iFormat))
{
        [логика обработки каждого значения iFormat]
}

CloseClipboard();
```

Число находящихся в данный момент в буфере обмена форматов можно получить с помощью вызова функции *CountClipboardFormats*:

```
iCount = CountClipboardFormats();
```

Отложенное исполнение

Когда данные помещаются в буфер обмена, то как правило делается копия данных и буферу обмена передается описатель на область памяти, в котором хранится эта копия. Для очень больших элементов данных такой подход может привести к непозволительному расходу памяти. Если пользователь никогда не вставит эти данные в другую программу, они будут продолжать занимать память до тех пор, пока их не заменит что-нибудь еще.

Можно избежать этой проблемы, если использовать прием, который называется "отложенное исполнение" (delayed rendering). При этом программа фактически не сохраняет данные до тех пор, пока они не потребуются другой программе. Вместо того, чтобы передавать Windows описатель этих данных, при вызове функции *SetClipboardData* просто используйте NULL:

```
OpenClipboard(hwnd);

EmptyClipboard();

SetClipboardData(iFormat, NULL);

CloseClipboard();
```

Можно делать несколько вызовов функции *SetClipboardData*, в которых использовать разные значения *iFormat*. При этом в некоторых из них можно задавать значение параметра, равное NULL, а в остальных использовать действительный описатель.

Это все достаточно просто, но теперь процесс слегка усложняется. Если другая программа вызывает функцию *GetClipboardData*, то Windows проверит, равен ли NULL описатель для указанного формата. Если да, то Windows отправит "владельцу буфера обмена" (вашей программе) сообщение с запросом о реальном описателе этих данных. Затем ваша программа должна сохранять этот описатель.

"Владелец буфера обмена" — это последнее окно, поместившее данные в буфер обмена. Когда программа вызывает функцию *OpenClipboard*, Windows сохраняет описатель окна, который имеется в этой функции. Этот описатель идентифицирует то окно, которое открыло буфер обмена. После вызова функции *EmptyClipboard*, Windows считает это окно новым владельцем буфера обмена.

Программа, которая использует отложенное исполнение, должна в своей оконной процедуре обрабатывать три сообщения: WM_RENDERFORMAT, WM_RENDERALLFORMATS и WM_DESTROYCLIPBOARD. Windows посылает вашей оконной процедуре сообщение WM_RENDERFORMAT, когда другая программа вызывает функцию *GetClipboardData*. Значение параметра *wParam* представляет из себя требуемый формат. При обработке сообщения WM_RENDERFORMAT нельзя открывать и очищать буфер обмена. Просто создайте область памяти того формата, который задан параметром *wParam*, передайте в нее данные и используйте вызов функции *SetClipboardData* с правильным форматом и незафиксированным описателем. Очевидно, что при обработке сообщения WM_RENDERFORMAT, необходимо иметь информацию в вашей программе, чтобы правильно создать эти данные. Когда другая программа вызывает функцию *EmptyClipboard*, Windows посылает вашей программе сообщение WM_DESTROYCLIPBOARD. Оно говорит вашей программе, что информация, необходимая для создания данных для буфера обмена, больше не нужна. Ваша программа перестает быть владельцем буфера обмена.

Если программа завершается в тот момент, когда она является владельцем буфера обмена, а в буфере обмена находится описатель данных NULL, который программа установила при вызове функции *SetClipboardData*, вы получите сообщение WM_RENDERALLFORMATS. Необходимо открыть буфер обмена, очистить ее, поместить данные в соответствующие области памяти и вызвать для каждого формата функцию *SetClipboardData*. Затем закрыть буфер обмена. Сообщение WM_RENDERALLFORMATS — это одно из последних сообщений, которое получает оконная процедура вашей программы. За ним следует сообщение WM_DESTROYCLIPBOARD (поскольку все данные переданы), а затем обычное сообщение WM_DESTROY.

Если программа может передавать в буфер обмена данные только одного формата (например, текст), то можно сочетать обработку сообщений WM_RENDERALLFORMATS и WM_RENDERFORMAT следующим образом:

```
case WM_RENDERALLFORMATS :
       OpenClipboard(hwnd);
       EmptyClipboard();      // fall through

case WM_RENDERFORMAT :
       [помещение текста в глобальную область памяти]
       SetClipboardData(CF_TEXT, hMem);

       if(iMsg == WM_RENDERALLFORMATS)
              CloseClipboard();
       return 0;
```

Если в программе используется несколько форматов для буфера обмена, то сообщение WM_RENDERFORMAT нужно обрабатывать только для формата заданного параметром *wParam*. Сообщение WM_DESTROYCLIPBOARD не надо обрабатывать в тех случаях, когда вашей программе необременительно сохранять информацию, необходимую для создания данных.

Нестандартные форматы данных

До сих пор мы имели дело только со стандартными, определяемыми Windows, форматами для буфера обмена. Однако, вам может понадобиться использовать буфер обмена для хранения данных в собственном, нестандартном формате (private data format). Во многих текстовых процессорах используется этот прием для хранения текста, содержащего информацию о шрифтах и форматах.

На первых порах эта концепция может показаться бессмысленной. Если целью буфера обмена является передача данных между приложениями, то зачем хранить в буфере обмена данные, которые понятны только одному приложению? Ответ прост: буфер обмена существует также и для передачи данных внутри одной программы (или, может быть, между различными экземплярами одной программы), а любая программа, очевидно, понимает свои собственные форматы.

Имеется несколько способов использования данных нестандартного формата. В простейшем, данные находятся как бы в одном из стандартных форматов для буфера обмена (текст, битовый образ или метафайл), но при этом они имеют значение только для вашей программы. В этом случае, при вызове функций *SetClipboardData* и *GetClipboardData*, используйте одно из следующих значений *iFormat* : CF_DSPTEXT, CF_DSPBITMAP, CF_DSPMETAFILEPICT или CF_DSPENHMETAFILE. Буквы "DSP" означают "display, вывод на экран". Эти форматы позволяют средствам просмотра буфера обмена Windows вывести эти данные на экран в виде текста, битового образа или метафайла. Однако, другая программа, которая вызывает функцию *GetClipboardData* с использованием обычных форматов CF_TEXT, CF_BITMAP, CF_METAFILEPICT или CF_ENHMETAFILE, не сможет получить эти данные.

Если при помещении данных в буфер обмена используется один из таких специальных форматов, то такой же формат необходимо использовать и при получении данных оттуда. Но как узнать, чьи это данные находятся в буфере обмена: одного из экземпляров вашей программы или совсем другой программы? Есть способ узнать это: сначала, вызывая функцию *GetClipboardOwner*, получите описатель окна владельца буфера обмена:

```
hwndClipOwner = GetClipboardOwner();
```

Теперь, для этого описателя окна можно получить имя класса окна:

```
char szClassName [16];
[другие строки программы]
GetClassName(hwndClipOwner, &szClassName, 16);
```

Если полученное имя класса окна совпадает с именем класса окна вашей программы, то данные были помещены в буфер обмена экземпляром вашей программы.

Второй способ использования данных нестандартного формата заключается в использовании флага CF_OWNERDISPLAY. Описателем памяти в вызове функции *SetClipboardData* является значение NULL:

```
SetClipboardData(CF_OWNERDISPLAY, NULL);
```

Это способ, который используется некоторыми текстовыми процессорами для отображения отформатированного текста в рабочей области, включенной в Windows 95, программы просмотра содержимого буфера обмена. Очевидно, что программа просмотра не знает, как выводить на экран этот отформатированный текст. Когда текстовый редактор задает формат CF_OWNERDISPLAY, он также принимает на себя заботу о рисовании в рабочей области программы просмотра буфера обмена.

Поскольку описатель памяти равен NULL, программа, которая вызывает функцию *SetClipboardData* с форматом CF_OWNERDISPLAY (владелец буфера обмена), должна обработать сообщение об отложенном исполнении, которое Windows отправляет владельцу буфера обмена, а также пять дополнительных сообщений. Эти пять сообщений посылаются владельцу буфера обмена программой просмотра буфера обмена:

- WM_ASKCBFORMATNAME — программа просмотра буфера обмена посылает это сообщение владельцу буфера обмена для получения имени формата данных. Параметр *lParam* является указателем на буфер, а параметр *wParam* — это максимальное число символов, которые можно поместить в этот буфер. Владелец буфера обмена должен скопировать имя формата буфера обмена в этот буфер.
- WM_SIZECLIPBOARD — это сообщение говорит владельцу буфера обмена, что размер рабочей области программы просмотра буфера обмена был изменен. Параметр *wParam* — это описатель окна программы просмотра буфера обмена, а *lParam* — указатель на структуру типа RECT, содержащую новый размер. Если в структуре RECT все поля равны нулю, значит окно программы просмотра буфера обмена закрыто или свернуто в значок. Хотя в Windows допускается запуск только одного экземпляра программы просмотра буфера обмена, другие средства просмотра буфера обмена могут также послать это сообщение владельцу буфера обмена. Работа с этими несколькими средствами просмотра буфера обмена не является для владельца буфера обмена чем-то невозможным (при условии, что параметр *wParam* идентифицирует каждую конкретную программу просмотра), но это, конечно, не просто.
- WM_PAINTCLIPBOARD — это сообщение говорит владельцу буфера обмена о необходимости обновления рабочей области программы просмотра буфера обмена. И снова, параметр *wParam* — это описатель окна программы просмотра буфера обмена. А параметр *lParam* — это описатель структуры PAINTSTRUCT. Владелец буфера обмена может получить описатель контекста программы просмотра буфера обмена из поля *hdc* этой структуры.
- WM_HSCROLLCLIPBOARD и WM_VSCROLLCLIPBOARD — эти сообщения информируют владельца буфера обмена о том, что пользователь прокручивает полосы прокрутки программы просмотра буфера обмена. Параметр *wParam* — это описатель окна программы просмотра буфера обмена, младшее слово параметра *lParam* — это запрос на прокрутку, а старшее слово — это положение бегунка, если младшее слово равно SB_THUMBPOSITION.

Обработка этих сообщений может показаться более сложной, чем на самом деле. Однако, это дает пользователю преимущество: при копировании текста из текстового редактора в буфер обмена, ему понравится, если он обнаружит, что в рабочей области программы просмотра текст остается отформатированным.

Третий способ использования данных нестандартного формата буфера обмена состоит в необходимости регистрировать имя этого формата. Вы передаете это имя в Windows, а Windows передает вам число для использования его в качестве параметра формата в функциях *SetClipboardData* и *GetClipboardData*. Программы, в которых используется этот способ, обычно также копируют данные в буфер обмена в одном из стандартных форматов. Такой подход позволяет программе просмотра выводить данные в своей рабочей области (без возни, связанной с форматом CF_OWNERDISPLAY) и допускает копирование данных из буфера обмена другими программами.

Рассмотрим пример: предположим, что мы написали программу рисования вектора, которая копирует данные в буфер обмена в формате битового образа, формате метафайла и в собственном зарегистрированном формате. Программа просмотра буфера обмена выведет метафайл на экран. Другие программы, которые могут считывать из буфера обмена битовые образы или метафайлы, получат данные в этих форматах. Однако, если самой программе рисования вектора понадобится считать данные из буфера обмена, то данные будут копироваться в собственном зарегистрированном формате, поскольку вероятнее всего в этом формате содержится больше информации, чем в битовом образе или метафайле.

Новый формат для буфера обмена регистрируется следующим образом:

```
iFormat = RegisterClipboardFormat(pszFormatName);
```

Значение *iFormat* находится в диапазоне от 0xC000 до 0xFFFF. Программа просмотра буфера обмена (или программа, которая с помощью вызова функции *EnumClipboardFormats* получает все текущие форматы данных буфера обмена) может получить имя этого формата в коде ASCII, используя вызов функции *GetClipboardFormatName*:

```
GetClipboardFormatName(iFormat, psBuffer, iMaxCount);
```

Windows копирует максимум *iMaxCount* символов в *psBuffer*.

Программисты, которые используют этот способ для копирования данных в буфер обмена, могли бы опубликовать имя формата и фактический формат данных. Если программа станет популярной, тогда другим программам можно будет копировать в этом формате данные из буфера обмена.

Соответствующая программа просмотра буфера обмена

Программа, которая уведомляется об изменении содержимого буфера обмена, называется программой просмотра буфера обмена (clipboard viewer). В Windows 95 имеется такая программа, но можно также написать собственную программу просмотра буфера обмена. Программы просмотра буфера обмена уведомляются об изменениях в буфере обмена с помощью сообщений, посылаемых оконным процедурам таких программ.

Цепочка программ просмотра буфера обмена

Одновременно в Windows может быть запущено любое число программ просмотра буфера обмена, и все они могут уведомляться об изменениях в буфере обмена. Однако, с точки зрения Windows, имеется только одна программа просмотра буфера обмена, которую мы назовем текущей программой просмотра буфера обмена (current clipboard viewer). Windows поддерживает только один описатель окна для идентификации текущей программы просмотра буфера обмена и при изменении содержимого буфера обмена посылает сообщения только этому окну.

Программы-приложения для просмотра буфера обмена могут участвовать в создании цепочки программ просмотра буфера обмена (clipboard viewer chain), при этом все программы цепочки получают сообщения, которые Windows отправляет текущей программе просмотра буфера обмена. Когда программа регистрирует себя в качестве программы просмотра буфера обмена, она становится текущей программой просмотра буфера обмена. Windows передает этой программе описатель окна предыдущей текущей программы просмотра буфера обмена, и программа сохраняет этот описатель. Когда программа получает сообщение просмотра буфера обмена, она посылает это сообщение оконной процедуре следующей в цепочке программы.

Функции и сообщения программы просмотра буфера обмена

Используя вызов функции *SetClipboardViewer*, программа может стать звеном цепочки программ просмотра буфера обмена. Если основной целью создания программы является ее использование в качестве программы просмотра буфера обмена, то эту функцию можно вызвать при обработке сообщения WM_CREATE. Возвращаемым значением функции является описатель окна предыдущей текущей программы просмотра буфера обмена. Программа должна сохранить этот описатель в статической переменной:

```
static HWND hwndNextViewer;
[другие строки программы]

case WM_CREATE :
       [другие строки программы]

       hwndNextViewer = SetClipboardViewer(hwnd);
```

Если программа становится первой программой просмотра буфера обмена в текущем сеансе работы Windows, то описатель *hwndNextViewer* будет равен NULL.

При изменении содержимого буфера обмена Windows посылает сообщение WM_DRAWCLIPBOARD текущей программе просмотра буфера обмена (последней, зарегистрировавшей себя в качестве программы просмотра

буфера обмена). Каждая программа в цепочке программ просмотра буфера обмена должна использовать функцию *SendMessage* для передачи этого сообщения следующей программе просмотра буфера обмена. Последняя программа цепочки (первое окно, зарегистрировавшее себя в качестве программы просмотра буфера обмена) в качестве значения описателя *hwndNextViewer* будет хранить NULL. Если значение описателя *hwndNextViewer* будет равно NULL, то программа просто возвращает управление без посылки сообщения другой программе. (Не путайте сообщения WM_DRAWCLIPBOARD и WM_PAINTCLIPBOARD. Сообщение WM_PAINTCLIPBOARD посылается программой просмотра буфера обмена той программе, в которой для буфера обмена используется формат CF_OWNERDISPLAY. Сообщение WM_DRAWCLIPBOARD посылается Windows текущей программе просмотра буфера обмена.)

Простейший способ обработки сообщения WM_DRAWCLIPBOARD — это отправка его следующей программе просмотра буфера обмена (пока значение описателя *hwndNextViewer* не будет равно NULL) и превращение рабочей области окна вашей программы в недействительную:

```
case WM_DRAWCLIPBOARD :
        if(hwndNextViewer)
                SendMessage(hwndNextViewer, iMsg, wParam, lParam);

        InvalidateRect(hwnd, NULL, TRUE);
        return 0;
```

При обработке сообщения WM_PAINT можно прочитать содержимое буфера обмена, используя вызовы обычных функций *OpenClipboard*, *GetClipboardData* и *CloseClipboard*.

Если необходимо удалить программу из цепочки программ просмотра буфера обмена, нужно использовать для этого вызов функции *ChangeClipboardChain*. Параметрами этой функции являются описатель окна программы, удаляемой из цепочки, и описатель окна следующей программы просмотра буфера обмена:

```
ChangeClipboardChain(hwnd, hwndNextViewer);
```

Когда программа вызывает функцию *ChangeClipboardChain*, Windows посылает сообщение WM_CHANGECBCHAIN текущей программе просмотра буфера обмена. Параметром *wParam* этого сообщения является описатель окна программы, удаляемой из цепочки (первый параметр функции *ChangeClipboardChain*), а параметром *lParam* — описатель окна следующей программы просмотра буфера обмена (второй параметр функции *ChangeClipboardChain*), после удаления программы из цепочки.

Когда программа получает сообщение WM_CHANGECBCHAIN, необходимо проверить, равен ли параметр *wParam* значению *hwndNextViewer*, которое хранится в программе. Если да, программа должна сделать значение *hwndNextViewer* равным *lParam*. Это действие гарантирует, что ни одно из будущих сообщений WM_DRAWCLIPBOARD не будет отправлено окну изъятой из цепочки программы. Если параметр *wParam* не равен *hwndNextViewer*, а *hwndNextViewer* не равно NULL, то сообщение посылается следующей программе просмотра буфера обмена:

```
case WM_CHANGECBCHAIN :
        if((HWND) wParam == hwndNextViewer)
                hwndNextViewer =(HWND) lParam;

        else if(hwndNextViewer)
                SendMessage(hwndNextViewer, iMsg, wParam, lParam);
        return 0;
```

Инструкцию *else if*, которая проверяет *hwndNextViewer* на ненулевое значения, включать в программу не обязятельно. Значение NULL описателя *hwndNextViewer* показало бы, что программа, выполняющая эту инструкцию, является последней в цепочке, а этого не может быть.

Если программа перед завершением еще остается в цепочке, то ее необходимо удалить оттуда. Это можно сделать при обработке сообщения WM_DESTROY, используя вызов функции *ChangeClipboardChain*:

```
case WM_DESTROY :
        ChangeClipboardChain(hwnd, hwndNextViewer);
        PostQuitMessage(0);
        return 0;
```

В Windows также имеется функция, позволяющая программе получить описатель окна первой программы просмотра буфера обмена:

```
hwndViewer = GetClipboardViewer();
```

Обычно эта функция не нужна. Ее возвращаемое значение может быть равно NULL при отсутствии текущей программы просмотра буфера обмена.

В следующем примере показана работа цепочки программ просмотра буфера обмена. При запуске Windows описатель окна текущей программы просмотра буфера обмена равен NULL:

Описатель окна текущей программы просмотра буфера обмена:	NULL

Программа с описателем окна *hwnd1* вызывает функцию *SetClipboardViewer*. Функция возвращает NULL. Это значение заносится в поле *hwndNextViewer* программы:

Описатель окна текущей программы просмотра буфера обмена:	*hwnd1*
Описатель окна следующей, после hwnd1, программы:	NULL

Вторая программа с описателем окна *hwnd2* теперь вызывает функцию *SetClipboardViewer* и получает в ответ *hwnd1*:

Описатель окна текущей программы просмотра буфера обмена:	*hwnd2*
Описатель окна следующей, после hwnd2, программы:	*hwnd1*
Описатель окна следующей, после hwnd1, программы:	NULL

Третья программа (*hwnd3*), и затем четвертая (*hwnd4*) также вызывают функцию *SetClipboardViewer* и получают в ответ *hwnd2* и *hwnd3*:

Описатель окна текущей программы просмотра буфера обмена:	*hwnd4*
Описатель окна следующей, после hwnd4, программы:	*hwnd3*
Описатель окна следующей, после hwnd3, программы:	*hwnd2*
Описатель окна следующей, после hwnd2, программы:	*hwnd1*
Описатель окна следующей, после hwnd1, программы:	NULL

Когда содержимое буфера обмена меняется, Windows посылает сообщение WM_DRAWCLIPBOARD программе с описателем окна *hwnd4*, программа с описателем окна *hwnd4* посылает сообщение программе с описателем окна *hwnd3*, программа с описателем окна *hwnd3* посылает его программе с описателем окна *hwnd2*, программа с описателем окна *hwnd2* — программе с описателем окна *hwnd1*, а программа с описателем окна *hwnd1* возвращает управление.

Теперь, если необходимо удалить программу с описателем окна *hwnd2* из цепочки, делается вызов функции *ChangeClipboardChain*:

```
ChangeClipboardChain(hwnd2, hwnd1);
```

Windows посылает программе с описателем окна *hwnd4* сообщение WM_CHANGECBCHAIN с параметром *wParam* равным *hwnd2* и параметром *lParam* равным *hwnd1*. Поскольку следующей, после программы с описателем окна *hwnd4*, является программа с описателем окна *hwnd3*, программа с описателем окна *hwnd4* отправляет это сообщение программе с описателем окна *hwnd3*. Теперь программа с описателем *hwnd3* обнаруживает, что параметр *wParam* равен описателю окна следующей программы (*hwnd2*), поэтому она устанавливает описатель окна следующей программы равным параметру *lParam* (*hwnd1*) и возвращает управление. Задача решена. Теперь цепочка программ просмотра буфера обмена выглядит так:

Описатель окна текущей программы просмотра буфера обмена:	*hwnd4*
Описатель окна следующей, после hwnd4, программы:	*hwnd3*
Описатель окна следующей, после hwnd3, программы:	*hwnd1*
Описатель окна следующей, после hwnd1, программы:	NULL

Простая программа просмотра буфера обмена

Собственная программа просмотра буфера обмена не должна быть столь изощренной, как соответствующая программа, входящая в состав Windows 95. Например, программа просмотра буфера обмена может выводить данные буфера обмена на экран в одном формате. Программа просмотра буфера обмена CLIPVIEW, представленная на рис. 16.1, выводит данные буфера обмена на экран только в формате CF_TEXT.

CLIPVIEW.MAK

```
#------------------------
# CLIPVIEW.MAK make file
#------------------------

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

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

CLIPVIEW.C

```
/*---------------------------------------
   CLIPVIEW.C -- Simple Clipboard Viewer
                 (c) Charles Petzold, 1996
  ---------------------------------------*/

#include <windows.h>

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
     {
     static char  szAppName[] = "ClipView";
     HWND         hwnd;
     MSG          msg;
     WNDCLASSEX   wndclass;
     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = NULL;
     wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wndclass.hbrBackground =(HBRUSH) GetStockObject(WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;
     wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "Simple Clipboard Viewer(Text Only)",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

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

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HWND hwndNextViewer;
     HGLOBAL     hGMem;
```

```
HDC         hdc;
PSTR        pGMem;
PAINTSTRUCT ps;
RECT        rect;

switch(iMsg)
     {
     case WM_CREATE :
          hwndNextViewer = SetClipboardViewer(hwnd);
          return 0;

     case WM_CHANGECBCHAIN :
          if((HWND) wParam == hwndNextViewer)
               hwndNextViewer =(HWND) lParam;
          else if(hwndNextViewer)
               SendMessage(hwndNextViewer, iMsg, wParam, lParam);

          return 0;

     case WM_DRAWCLIPBOARD :
          if(hwndNextViewer)
               SendMessage(hwndNextViewer, iMsg, wParam, lParam);

          InvalidateRect(hwnd, NULL, TRUE);
          return 0;

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

          hGMem = GetClipboardData(CF_TEXT);

          if(hGMem != NULL)
               {
               pGMem =(PSTR) GlobalLock(hGMem);
               DrawText(hdc, pGMem, -1, &rect, DT_EXPANDTABS);
               GlobalUnlock(hGMem);
               }

          CloseClipboard();
          EndPaint(hwnd, &ps);
          return 0;

     case WM_DESTROY :
          ChangeClipboardChain(hwnd, hwndNextViewer);
          PostQuitMessage(0);
          return 0;
     }
return DefWindowProc(hwnd, iMsg, wParam, lParam);
}
```

Рис. 16.1 Программа CLIPVIEW

Программа CLIPVIEW обрабатывает сообщения WM_CREATE, WM_CHANGECBCHAIN, WM_DRAWCLIPBOARD и WM_DESTROY так, как это было показано ранее. Сообщение WM_PAINT просто открывает буфер обмена и использует функцию *GetClipboardData* с форматом CF_TEXT. Если функция возвращает описатель памяти, программа CLIPVIEW фиксирует его и использует функцию *DrawText* для вывода текста в своей рабочей области.

Программа просмотра буфера обмена, которая работает более чем с пятью стандартными форматами данных (как это делает соответствующая программа Windows 95), вынуждена выполнять дополнительную работу, например, отображать имена всех находящихся в данный момент в буфере обмена форматов данных. Это можно сделать, используя вызов функции *EnumClipboardFormats*, и получить имена нестандартных форматов можно с помощью

вызова функции *GetClipboardFormatName*. Программа просмотра буфера обмена, которая использует формат CF_OWNERDISPLAY, должна посылать в буфер обмена для вывода данных следующие четыре сообщения:

WM_PAINTCLIPBOARD	WM_VSCROLLCLIPBOARD
WM_SIZECLIPBOARD	WM_HSCROLLCLIPBOARD

При желании написать программу просмотра буфера обмена такого типа, необходимо, с помощью функции *GetClipboardOwner*, получить описатель окна владельца буфера обмена и отправить этому окну указанные сообщения, когда необходимо обновить рабочую область программы просмотра буфера обмена.

Глава 17 Динамический обмен данными

17

Динамический обмен данными (Dynamic Data Exchange, DDE) является одним из нескольких механизмов связи между процессами (interprocess communication, IPC), поддерживаемых в Windows 95. Тремя другими механизмами, о которых рассказывается в этой книге, являются папка обмена Windows (о которой говорилось в главе 16), разделение памяти в динамически подключаемых библиотеках (dynamic link libraries, DLL) (глава 19) и связь и внедрение объектов (object linking and embedding, OLE) (глава 20). DDE менее амбициозен (и обладает меньшими возможностями), чем OLE, но, как правило, легче реализуется.

DDE базируется на встроенной в Windows системе сообщений. Две программы Windows с помощью DDE поддерживают между собой диалог, посылая друг другу сообщения. Эти две программы называют "сервером" и "клиентом". Сервер DDE — это программа, имеющая доступ к данным, которые могут оказаться полезными другим программам Windows. Клиент DDE — это программа, получающая такие данные от сервера.

В Windows 95 программы могут использовать управляющую библиотеку DDE (DDE Management Library, DDEML), в которой имеются средства, позволяющие упростить использование DDE. DDEML изолирует программу от системы сообщений DDE посредством группы функций высокого уровня. Программы, использующие DDEML, отвечают на сообщения DDE с помощью функций обратного вызова (call-back function). Оба метода использования DDE вполне совместимы, поскольку DDEML является надстройкой над системой сообщений DDE.

В этой главе сначала будет рассказано о традиционном подходе к DDE, а затем продемонстрировано, как с помощью DDEML упростить решение проблемы связи между процессами.

Диалог DDE инициируется программой клиентом. Клиент рассылает всем работающим в данный момент в Windows программам сообщение WM_DDE_INITIATE. Это сообщение идентифицирует основную категорию необходимых клиенту данных. Сервер DDE, имеющий эти данные, может ответить на это сообщение. В этом случае диалог устанавливается.

Одна и та же программа в Windows может быть клиентом для одной программы и сервером для другой, но для этого требуются два разных диалога DDE. Сервер может передавать данные многим клиентам, и клиенты могут получать данные от многих серверов, но, опять же, для этого требуется много диалогов DDE. Чтобы эти диалоги оставались уникальными и независимыми, для каждого диалога (и со стороны клиента, и со стороны сервера) используется отдельное окно. Как правило, программа, работающая с DDE, создает невидимое дочернее окно для каждого диалога, который она поддерживает.

Программы, вовлеченные в диалог DDE, не нужно программировать каким-то особым образом для взаимодействия друг с другом. Как будет показано в следующем разделе этой главы, обычно разработчик программы сервера DDE полностью документирует то, как идентифицируются данные. Пользователь программы, которая может работать в качестве клиента DDE (например, Microsoft Excel), может использовать эту информацию для установки диалога между этими двумя программами.

При написании семейства двух или более программ для Windows, которые должны взаимодействовать друг с другом, но не с другими программами Windows, можно рассмотреть вопрос определения собственного протокола обмена сообщениями. Однако, если две или более программы должны как записывать, так и читать разделяемые данные, необходимо разместить эти данные в файле, проецируемом в память (memory-mapped file). Как будет показано в этой главе, клиенты DDE никогда не записывают данные, а только читают их. Следовательно, сервер DDE может безопасно выделять память с помощью функции *GlobalAlloc* или *HeapAlloc*.

Поскольку механизм DDE использует встроенную в Windows систему сообщений, он очень легко и естественно вписывается в систему. Правда, это не значит, что DDE легко реализовать. В протоколе имеется множество опций,

и программисту нужно быть готовым к тому, что придется столкнуться с некоторыми достаточно изощренными проблемами.

Основные концепции

Когда клиент запрашивает у сервера данные, он должен иметь возможность идентифицировать запрашиваемые данные. Это делается с помощью трех символьных строк, которые называются "приложение" (application), "раздел" (topic) данных и "элемент" (item) данных.

Приложение, раздел и элемент

Назначение приложения, раздела и элемента станет понятнее из примера. В первой половине этой главы будет показано, как написать Windows-программу-сервер DDE, которая называется DDEPOP1. В этой программе содержатся данные о населении Соединенных Штатов, полученные по итогам переписей населения в 1970, 1980 и 1990 годах. На основе квадратичной экстраполяции программа может рассчитать численность населения любого штата или Соединенных Штатов в целом в любой момент времени.

Любой, кто пишет программу сервера DDE должен документировать то, как с помощью трех символьных строк идентифицировать данные сервера:

- Имя приложения сервера. В данном примере им является просто "DDEPOP1". Каждый сервер имеет только одно имя приложения, каковым является имя программы.
- Имя раздела. Все серверы DDE поддерживают по меньшей мере один раздел. В случае программы DDEPOP1, поддерживается один раздел, он идентифицируется строкой "US_Population". Очевидно, что программа DDEPOP1 могла бы быть дополнена информацией о площади штатов в квадратных милях. В таком случае поддерживался бы второй раздел с именем "US_Area".
- Имя элемента. Внутри каждого раздела сервер DDE поддерживает один или более элементов данных. В программе DDEPOP1 элемент идентифицирует штат с помощью стандартной двухсимвольной почтовой аббревиатуры, например "NY" для Нью-Йорка, "CA" для Калифорнии и "US" для страны в целом. В программе DDEPOP1 поддерживается 52 элемента — 50 штатов, округ Колумбия ("DC") и страна в целом.

Описание этих данных важно при использовании сервера с другими Windows-программами, которые могут выступать в качестве клиента, например, Microsoft Excel. Для использования программы DDEPOP1 вместе с Microsoft Excel, в ячейке электронной таблицы программы Microsoft Excel можно набрать следующую строку:

```
=DDEPOP1 | US_Population ! US
```

В этих трех словах заданы приложение, раздел и элемент (в данном случае все население Соединенных Штатов). Если файл DDEPOP1.EXE еще не запущен, Microsoft Excel попытается запустить его. (Программа DDEPOP1 должна находиться в текущем каталоге или в списке каталогов переменной окружения PATH.) При удачном запуске Microsoft Excel инициирует с программой DDEPOP1 диалог DDE, получит данные о населении и выведет эти данные в виде числа в ячейке электронной таблицы. Этот результат, обозначающий численность населения, может быть отформатирован, представлен в графическом виде, использован в вычислениях.

Самым замечательным является то, что цифры, обозначающие в электронной таблице численность населения, будут периодически обновляться. Этот процесс называется горячей связью (hot link) или (в более простом варианте) теплой связью (warm link). Каждые 5 секунд программа DDEPOP1 пересчитывает данные о населении и уведомляет клиента при изменении элемента. В случае с населением США, число будет увеличиваться на 1 почти каждые 15 секунд.

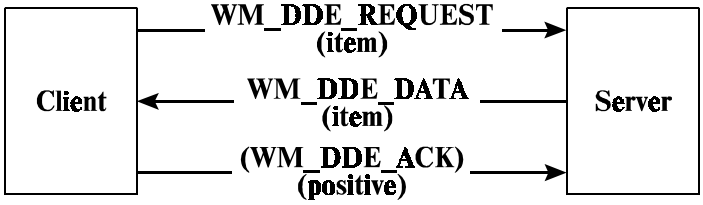
Типы диалогов

В DDE имеется три основных типа диалога — холодная связь (cold link), горячая связь и теплая связь. В этих диалогах используются сообщения DDE, определенные в заголовочном файле DDE.H. Простейшим из трех диалогов является холодная связь.

Холодная связь

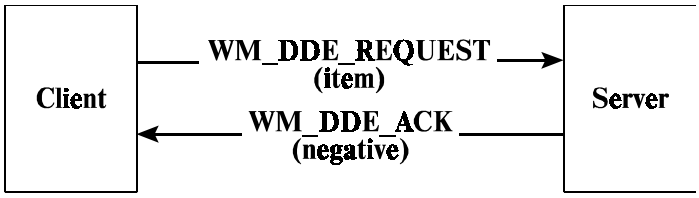
Диалог холодной связи начинается, когда клиент широковещательно рассылает сообщение WM_DDE_INITIATE, идентифицирующее приложение и раздел, который требуется клиенту. (Чтобы начать диалог с любым приложением-сервером или любым разделом данных, приложение и раздел, соответственно, могут быть равны NULL.) Приложение-сервер, которое поддерживает заданный раздел, отвечает клиенту сообщением WM_DDE_ACK (подтверждение, acknowledge):

Затем клиент запрашивает конкретный элемент данных, посылая сообщение WM_DDE_REQUEST. Если у сервера имеется этот элемент данных, то он отвечает, отправляя клиенту сообщение WM_DDE_DATA:

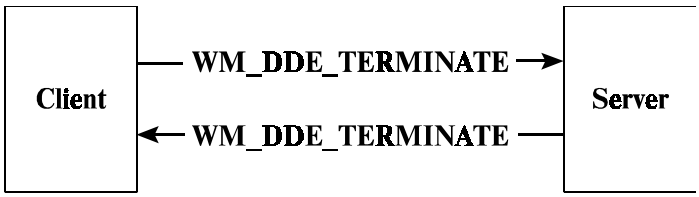


Здесь также показано, что клиент может уведомить сервер о получении сообщения WM_DDE_DATA. Это не обязательно (что обозначено посредством заключения в скобки сообщения WM_DDE_ACK). Необходимость подтверждения сервер показывает с помощью флага, передаваемого с сообщением WM_DDE_DATA. Флаг, передаваемый с сообщением WM_DDE_ACK, показывает, что данные получены успешно.

Если клиент посылает серверу сообщение WM_DDE_REQUEST, а у сервера нет запрашиваемого элемента данных, то сервер посылает клиенту негативное сообщение WM_DDE_ACK:



Диалог DDE продолжается до тех пор, пока клиент отправляет серверу сообщения WM_DDE_REQUEST — с запросами о том же самом или другом элементе данных — и сервер отвечает сообщениями WM_DDE_DATA или WM_DDE_ACK. Диалог заканчивается, когда клиент и сервер посылают друг другу сообщения WM_DDE_TERMINATE:



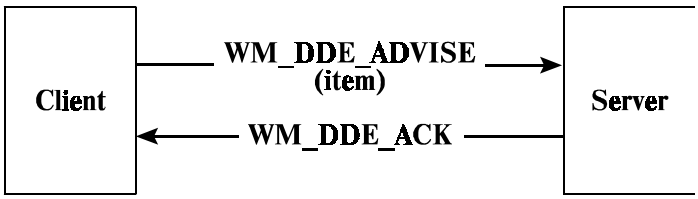
Хотя здесь показано, что клиент первым посылает сообщение WM_DDE_TERMINATE, это не всегда так. Сервер может первым послать сообщение WM_DDE_TERMINATE, и клиент должен ответить на него.

Горячая связь

Одной из проблем, связанной с холодной связью, является то, что данные, к которым сервер имеет доступ, во время передачи могут меняться. (Это происходит в случае программы DDEPOP1, рассчитывающей прирост населения.) При использовании холодной связи клиент не знает, когда изменяются данные. Эту проблему решает горячая связь.

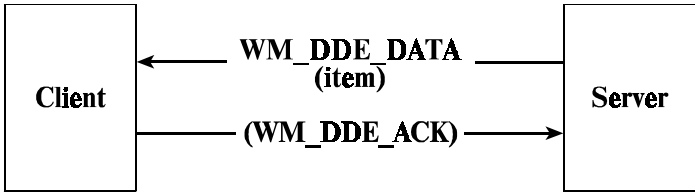
Также, как при холодной связи диалог DDE начинается с сообщения WM_DDE_INITIATE и сообщения WM_DDE_ACK:

Клиент задает требуемый ему элемент данных, посылая серверу сообщение WM_DDE_ADVISE. Отвечая сообщением WM_DDE_ACK, сервер показывает, есть ли у него доступ к этому элементу:

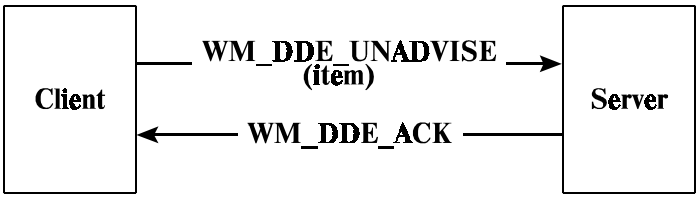


Позитивное подтверждение показывает, что сервер может дать клиенту нужные данные, негативное — что не может.

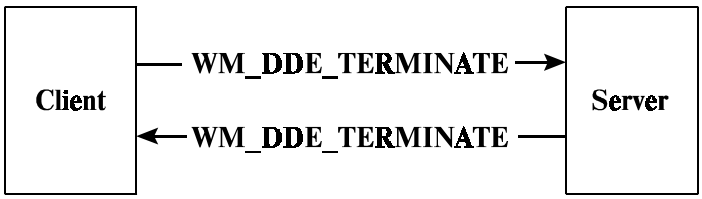
С этого момента сервер обязан извещать клиента об изменении значения элемента данных, когда бы оно не произошло. Для этого извещения используется сообщение WM_DDE_DATA, на которое клиент (в зависимости от состояния флага в сообщении WM_DDE_DATA) может отвечать или не отвечать сообщением WM_DDE_ACK:



Если клиенту больше не нужно знать об изменении элемента данных, он посылает серверу сообщение WM_DDE_UNADVISE, и сервер отвечает подтверждением:



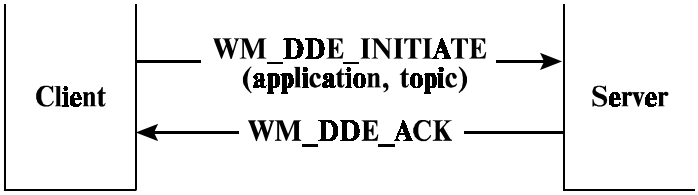
Диалог завершается после обмена сообщениями WM_DDE_TERMINATE:



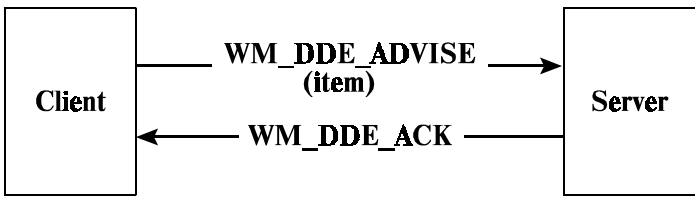
Холодная связь и горячая связь не являются взаимоисключающими. В рамках одного диалога DDE, клиент может запросить какие-либо данные с помощью сообщения WM_DDE_REQUEST (холодная связь), а другие данные с помощью сообщения WM_DDE_ADVISE (горячая связь).

Теплая связь

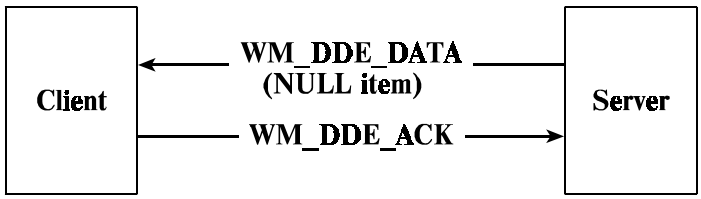
Теплая связь сочетает в себе элементы холодной и горячей связи. Диалог начинается как обычно:



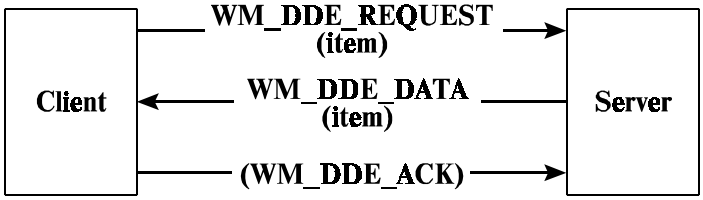
Как и в случае горячей связи клиент посылает серверу сообщение WM_DDE_ADVISE, а сервер отвечает на него либо позитивно, либо негативно:



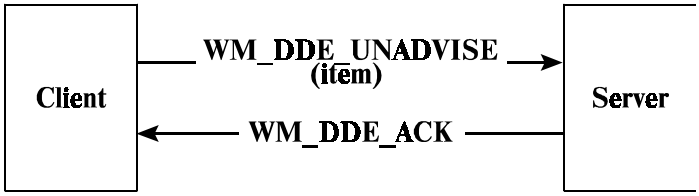
Однако, флаг, передаваемый с сообщением WM_DDE_ADVISE, показывает, что клиенту достаточно только узнавать о каждом изменении данных, а немедленно получать новые данные не нужно. Поэтому сервер, при любых изменениях данных, посылает клиенту сообщение WM_DDE_DATA с элементом данных равным NULL:



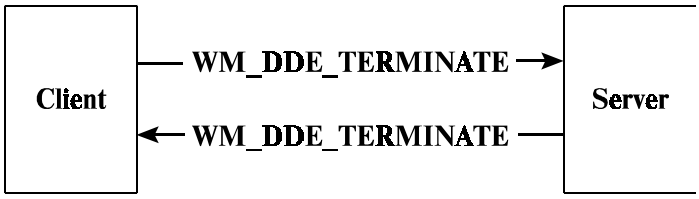
Теперь клиент знает, что конкретный элемент данных изменился. Для получения этого элемента, точно также, как и при холодной связи, клиент использует сообщение WM_DDE_REQUEST:



Также, как при горячей связи клиент может остановить сообщения, извещающие его об изменении элементов данных, отправив серверу сообщение WM_DDE_UNADVISE:



Диалог завершается после обмена сообщениями WM_DDE_TERMINATE:



В этих трех типах диалога используются все сообщения DDE, за исключением двух: WM_DDE_POKE (в котором клиент передает серверу данные, которые сервер не запрашивал) и WM_DDE_EXECUTE (в котором клиент посылает серверу командную строку). В этой главе эти сообщения рассматриваться не будут.

В заголовочном файле DDE.H также определены четыре структуры:

- DDEACK (используется в сообщении WM_DDE_ACK)
- DDEADVISE (используется в сообщении WM_DDE_ADVISE)
- DDEDATA (используется в сообщении WM_DDE_DATA)
- DDEPOKE (используется в сообщении WM_DDE_POKE)

В примерах программ этой главы мы коснемся первых трех представленных здесь структур.

Символьные строки и атомы

Ранее было показано, как клиент и сервер DDE идентифицируют данные с помощью трех символьных строк — приложения, раздела и элемента. Но в реальных сообщениях между клиентом и сервером эти символьные строки не появляются; вместо них используются атомы (atoms).

Атомы — это значения типа WORD, которые соответствуют символьным строкам, причем регистр строк значения не имеет. Атомы можно использовать в программе для работы с символьными строками, в этом случае таблица атомов (таблица соответствия значений атомов символьным строкам) хранится в области данных программы.

Атом определяется следующим образом:

```
ATOM aAtom;
```

С помощью функции *AddAtom* к таблице атомов можно добавить строку:

```
aAtom = AddAtom(pString);
```

Если такой символьной строки в таблице атомов еще нет, функция добавит ее туда, а ее возвращаемым значением будет уникальное число, идентифицирующее строку. Каждый атом имеет счетчик ссылок (reference count); это то число, которое показывает сколько раз для данной строки вызывалась функция *AddAtom*. Начальным значением счетчика ссылок является 1. Если в таблице атомов уже существует данная символьная строка (то есть, если это второй или последующий вызов функции *AddAtom* для данной строки), то функция возвращает число, идентифицирующее строку и увеличивает на 1 значение счетчика ссылок.

Функция *DeleteAtom* уменьшает на 1 значение счетчика ссылок:

```
DeleteAtom(aAtom);
```

Если значение счетчика ссылок становится равным 0, атом и символьная строка удаляются из таблицы атомов.

Возвращаемым значением функции *FindAtom* является атом, соответствующей заданной строке (или 0, если такой строки нет в таблице атомов):

```
aAtom = FindAtom(pString);
```

На состояние счетчика ссылок эта функция не влияет.

Символьную строку, соответствующую данному атому, можно определить с помощью функции *GetAtomName*:

```
iBytes = GetAtomName(aAtom, pBuffer, iBufferSize);
```

Последним параметром является размер буфера, на который указывает второй параметр. Функция возвращает число скопированных в буфер байтов и не влияет на состояние счетчика ссылок.

Эти четыре функции (остальные не столь важны) позволяют работать с атомами внутри программы. Однако, поскольку таблица атомов хранится в области данных конкретной программы, то атомы уникальны только для нее. Чтобы использовать атомы с DDE, нужны четыре функции, похожие на только что описанные:

```
aAtom = GlobalAddAtom(pString);

GlobalDeleteAtom(aAtom);

aAtom = GlobalFindAtom(pString);

iBytes = GlobalGetAtomName(aAtom, pBuffer, iBufferSize);
```

Таблица для таких атомов хранится в разделяемой области памяти в динамически подключаемой библиотеке внутри Windows и, следовательно, доступна всем программам Windows. Можно использовать функцию *GlobalAddAtom* в одной программе, чтобы добавить строку к таблице атомов, и передать этот атом другой программе. Эта другая программа может использовать функцию *GlobalGetAtomName* для получения символьной строки, соответствующей переданному атому. Таким образом программы Windows идентифицируют приложение, раздел и элемент DDE.

Правила, определяющие использование атомов с DDE, чрезвычайно важны: нехорошо, когда нужный в одной программе атом, другая программа удаляет из таблицы атомов. Также плохо, если атомы, которые больше не нужны, не удаляются из таблицы атомов. По этой причине при управлении атомами нужно быть очень внимательным.

Атомы используются для строк приложений, разделов и элементов DDE. Память для структур данных, которые передаются от одной программы Windows к другой, может быть либо выделена из динамической области с помощью функций *GlobalAlloc* или *HeapAlloc*, либо реализована через механизм файлов, проецируемых в память. Каждый из этих методов позволяет разделять данные между несколькими программами Windows. Однако, использовать файлы, проецируемые в память, для диалогов DDE не рекомендуется, поскольку это дает возможность клиенту перезаписать или исказить данные. При распределении памяти из динамической области сервер гарантирует, что для всех клиентов данные предоставляются только для чтения. Правила DDE, которые позволяют определить, какая программа ответственна за выделение и освобождение таких блоков памяти, также достаточно строги.

В Win32 функция *GlobalAlloc* является как бы упаковкой для функции *HeapAlloc*, и память, которую она резервирует, становится частной памятью того, кто вызывает функцию. Другие процессы не имеют к ней доступа. Такое положение не меняют даже флаги GMEM_DDESHARE (и GMEM_SHARE), поскольку функция *GlobalAlloc* их игнорирует. Как клиенту DDE увидеть данные, хранящиеся в частной памяти сервера? А никак. Когда сервер посылает клиенту сообщение WM_DDE_DATA, Windows делает копию соответствующей структуры DDEDATA, в которой эти данные хранятся. Клиент видит не сам выделенный блок памяти, а его копию.

Программа сервер DDE

Теперь мы готовы начать изучение программы DDEPOP1, программу сервер DDE, в которой для клиента DDE поддерживаются данные о темпе роста населения штатов. Эта программа представлена на рис. 17.1. Обратите внимание, что хотя для функции *GlobalAlloc* флаг GMEM_DDESHARE не нужен, для ясности этот флаг включен в представленный ниже исходный текст программы.

DDEPOP1.MAK

```
#----------------------
# DDEPOP1.MAK make file
#----------------------

ddepop1.exe : ddepop1.obj ddepop1.res
     $(LINKER) $(GUIFLAGS) -OUT:ddepop1.exe ddepop1.obj ddepop1.res $(GUILIBS)

ddepop1.obj : ddepop1.c ddepop.h
     $(CC) $(CFLAGS) ddepop1.c

ddepop1.res : ddepop1.rc ddepop.ico
     $(RC) $(RCVARS) ddepop1.rc
```

DDEPOP1.C

```
/*-----------------------------------------
   DDEPOP1.C -- DDE Server for Population Data
              (c) Charles Petzold, 1996
  -----------------------------------------*/

#include <windows.h>
#include <dde.h>
#include <string.h>
#include "ddepop.h"

typedef struct
     {
     unsigned int fAdvise:1;
     unsigned int fDeferUpd:1;
     unsigned int fAckReq:1;
     unsigned int dummy:13;
     long         lPopPrev;
     }
     POPADVISE;

#define ID_TIMER    1
#define DDE_TIMEOUT 3000

LRESULT CALLBACK WndProc        (HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK ServerProc     (HWND, UINT, WPARAM, LPARAM);
BOOL    CALLBACK TimerEnumProc  (HWND, LONG);
BOOL    CALLBACK CloseEnumProc  (HWND, LONG);
BOOL             PostDataMessage(HWND, HWND, int, BOOL, BOOL, BOOL);

char      szAppName[]     = "DdePop1";
char      szServerClass[] = "DdePop1.Server";
HINSTANCE hInst;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
     {
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

     hInst = hInstance;

               // Register window class

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = 0;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
```

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

     RegisterClassEx(&wndclass);

               // Register window class for DDE Server

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = 0;
     wndclass.lpfnWndProc   = ServerProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 2 * sizeof(DWORD);
     wndclass.hInstance     = hInstance;
     wndclass.hIcon         = NULL;
     wndclass.hCursor       = NULL;
     wndclass.hbrBackground = NULL;
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szServerClass;
     wndclass.hIconSm       = NULL;

     RegisterClassEx(&wndclass);
     hwnd = CreateWindow(szAppName, "DDE Population Server",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     InitPops();                       // initialize 'pop' structure

     SetTimer(hwnd, ID_TIMER, 5000, NULL);

     ShowWindow(hwnd, SW_SHOWMINNOACTIVE);
     UpdateWindow(hwnd);

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

     KillTimer(hwnd, ID_TIMER);

     return msg.wParam;
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char szTopic[] = "US_Population";
     ATOM        aApp, aTop;
     HWND        hwndClient, hwndServer;

     switch(iMsg)
          {
          case WM_DDE_INITIATE :

                    // wParam          -- sending window handle
                    // LOWORD(lParam) -- application atom
                    // HIWORD(lParam) -- topic atom

               hwndClient =(HWND) wParam;
```

```
               aApp = GlobalAddAtom(szAppName);
               aTop = GlobalAddAtom(szTopic);

                    // Check for matching atoms, create window, and acknowledge

               if((LOWORD(lParam) == NULL || LOWORD(lParam) == aApp) &&
                  (HIWORD(lParam) == NULL || HIWORD(lParam) == aTop))
                    {
                    hwndServer = CreateWindow(szServerClass, NULL,
                                              WS_CHILD, 0, 0, 0, 0,
                                              hwnd, NULL, hInst, NULL);
                    SetWindowLong(hwndServer, 0,(LONG) hwndClient);
                    SendMessage((HWND) wParam, WM_DDE_ACK,
                                (WPARAM) hwndServer,
                                 MAKELPARAM(aApp, aTop));
                    }

                    // Otherwise, delete the atoms just created

               else
                    {
                    GlobalDeleteAtom(aApp);
                    GlobalDeleteAtom(aTop);
                    }

               return 0;

          case WM_TIMER :
          case WM_TIMECHANGE :

                    // Calculate new current populations

               CalcPops();

                    // Notify all child windows

               EnumChildWindows(hwnd, &TimerEnumProc, 0L);
               return 0;

          case WM_QUERYOPEN :
               return 0;

          case WM_CLOSE :

                    // Notify all child windows

               EnumChildWindows(hwnd, &CloseEnumProc, 0L);

               break;                  // for default processing

          case WM_DESTROY :
               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }

LRESULT CALLBACK ServerProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     ATOM          aItem;
     char          szItem[10];
     DDEACK        DdeAck;
     DDEADVISE    *pDdeAdvise;
     DWORD         dwTime;
```

```
GLOBALHANDLE  hPopAdvise, hDdeAdvise, hCommands, hDdePoke;
int           i;
UINT          uiLow, uiHi;
HWND          hwndClient;
MSG           msg;
POPADVISE    *pPopAdvise;
WORD          cfFormat, wStatus;

switch(iMsg)
     {
     case WM_CREATE :

               // Allocate memory for POPADVISE structures

          hPopAdvise = GlobalAlloc(GHND, NUM_STATES * sizeof(POPADVISE));

          if(hPopAdvise == NULL)
               DestroyWindow(hwnd);
          else
               SetWindowLong(hwnd, 4,(LONG) hPopAdvise);

          return 0;

     case WM_DDE_REQUEST :

               // wParam          -- sending window handle
               // LOWORD(lParam) -- data format
               // HIWORD(lParam) -- item atom

          hwndClient =(HWND) wParam;
          cfFormat   = LOWORD(lParam);
          aItem      = HIWORD(lParam);

               // Check for matching format and data item

          if(cfFormat == CF_TEXT)
               {
               GlobalGetAtomName(aItem, szItem, sizeof(szItem));

               for(i = 0; i < NUM_STATES; i++)
                    if(strcmp(szItem, pop[i].szState) == 0)
                         break;

               if(i < NUM_STATES)
                    {
                    GlobalDeleteAtom(aItem);
                    PostDataMessage(hwnd, hwndClient, i,
                                    FALSE, FALSE, TRUE);
                    return 0;
                    }
               }

               // Negative acknowledge if no match

          DdeAck.bAppReturnCode = 0;
          DdeAck.reserved       = 0;
          DdeAck.fBusy          = FALSE;
          DdeAck.fAck           = FALSE;

          wStatus = *((WORD *) &DdeAck);

          if(!PostMessage(hwndClient, WM_DDE_ACK,(WPARAM) hwnd,
                          PackDDElParam(WM_DDE_ACK, wStatus, aItem)))
               {
               GlobalDeleteAtom(aItem);
```

```
                }

            return 0;

        case WM_DDE_ADVISE :

                // wParam -- sending window handle
                // lParam -- DDEADVISE memory handle & item atom

            UnpackDDElParam(WM_DDE_ADVISE, lParam, &uiLow, &uiHi);
            FreeDDElParam(WM_DDE_ADVISE, lParam);

            hwndClient =(HWND) wParam;
            hDdeAdvise =(GLOBALHANDLE) uiLow;
            aItem      =(ATOM) uiHi;

            pDdeAdvise =(DDEADVISE *) GlobalLock(hDdeAdvise);

                // Check for matching format and data item

            if(pDdeAdvise->cfFormat == CF_TEXT)
                {
                GlobalGetAtomName(aItem, szItem, sizeof(szItem));

                for(i = 0; i < NUM_STATES; i++)
                    if(strcmp(szItem, pop[i].szState) == 0)
                        break;

                    // Fill in the POPADVISE structure and acknowledge

                if(i < NUM_STATES)
                    {
                    hPopAdvise =(GLOBALHANDLE) GetWindowLong(hwnd, 4);
                    pPopAdvise =(POPADVISE *)
                                        GlobalLock(hPopAdvise);

                    pPopAdvise[i].fAdvise   = TRUE;
                    pPopAdvise[i].fDeferUpd = pDdeAdvise->fDeferUpd;
                    pPopAdvise[i].fAckReq   = pDdeAdvise->fAckReq;
                    pPopAdvise[i].lPopPrev  = pop[i].lPop;

                    GlobalUnlock(hDdeAdvise);
                    GlobalFree(hDdeAdvise);

                    DdeAck.bAppReturnCode = 0;
                    DdeAck.reserved       = 0;
                    DdeAck.fBusy          = FALSE;
                    DdeAck.fAck           = TRUE;

                    wStatus = *((WORD *) &DdeAck);

                    if(!PostMessage(hwndClient, WM_DDE_ACK,
                                    (WPARAM) hwnd,
                                     PackDDElParam(WM_DDE_ACK,
                                                   wStatus, aItem)))
                        {
                        GlobalDeleteAtom(aItem);
                        }
                    else
                        {
                        PostDataMessage(hwnd, hwndClient, i,
                                        pPopAdvise[i].fDeferUpd,
                                        pPopAdvise[i].fAckReq,
                                        FALSE);
                        }
```

```
                    GlobalUnlock(hPopAdvise);
                    return 0;
                    }
               }

                    // Otherwise, post a negative WM_DDE_ACK

          GlobalUnlock(hDdeAdvise);

          DdeAck.bAppReturnCode = 0;
          DdeAck.reserved       = 0;
          DdeAck.fBusy          = FALSE;
          DdeAck.fAck           = FALSE;

          wStatus = *((WORD *) &DdeAck);

          if(!PostMessage(hwndClient, WM_DDE_ACK,(WPARAM) hwnd,
                           PackDDElParam(WM_DDE_ACK, wStatus, aItem)))
               {
               GlobalFree(hDdeAdvise);
               GlobalDeleteAtom(aItem);
               }

          return 0;

     case WM_DDE_UNADVISE :

               // wParam          -- sending window handle
               // LOWORD(lParam) -- data format
               // HIWORD(lParam) -- item atom

          hwndClient =(HWND) wParam;
          cfFormat   = LOWORD(lParam);
          aItem      = HIWORD(lParam);

          DdeAck.bAppReturnCode = 0;
          DdeAck.reserved       = 0;
          DdeAck.fBusy          = FALSE;
          DdeAck.fAck           = TRUE;

          hPopAdvise  =(GLOBALHANDLE) GetWindowLong(hwnd, 4);
          pPopAdvise =(POPADVISE *) GlobalLock(hPopAdvise);

               // Check for matching format and data item

          if(cfFormat == CF_TEXT || cfFormat == NULL)
               {
               if(aItem ==(ATOM) NULL)
                    for(i = 0; i < NUM_STATES; i++)
                         pPopAdvise[i].fAdvise = FALSE;
               else
                    {
                    GlobalGetAtomName(aItem, szItem, sizeof(szItem));

                    for(i = 0; i < NUM_STATES; i++)
                         if(strcmp(szItem, pop[i].szState) == 0)
                              break;

                    if(i < NUM_STATES)
                         pPopAdvise[i].fAdvise = FALSE;
                    else
                         DdeAck.fAck = FALSE;
                    }
               }
```

```
          else
               DdeAck.fAck = FALSE;

               // Acknowledge either positively or negatively
          wStatus = *((WORD *) &DdeAck);

          if(!PostMessage(hwndClient, WM_DDE_ACK,(WPARAM) hwnd,
                          PackDDElParam(WM_DDE_ACK, wStatus, aItem)))
               {
               if(aItem !=(ATOM) NULL)
                    GlobalDeleteAtom(aItem);
               }

          GlobalUnlock(hPopAdvise);
          return 0;

     case WM_DDE_EXECUTE :

               // Post negative acknowledge

          hwndClient =(HWND) wParam;
          hCommands  =(GLOBALHANDLE) lParam;

          DdeAck.bAppReturnCode = 0;
          DdeAck.reserved       = 0;
          DdeAck.fBusy          = FALSE;
          DdeAck.fAck           = FALSE;

          wStatus = *((WORD *) &DdeAck);

          if(!PostMessage(hwndClient, WM_DDE_ACK,(WPARAM) hwnd,
                          PackDDElParam(WM_DDE_ACK,
                                        wStatus,(UINT) hCommands)))
               {
               GlobalFree(hCommands);
               }
          return 0;

     case WM_DDE_POKE :

               // Post negative acknowledge

          UnpackDDElParam(WM_DDE_POKE, lParam, &uiLow, &uiHi);
          FreeDDElParam(WM_DDE_POKE, lParam);
          hwndClient =(HWND) wParam;
          hDdePoke   =(GLOBALHANDLE) uiLow;
          aItem      =(ATOM) uiHi;

          DdeAck.bAppReturnCode = 0;
          DdeAck.reserved       = 0;
          DdeAck.fBusy          = FALSE;
          DdeAck.fAck           = FALSE;

          wStatus = *((WORD *) &DdeAck);
          if(!PostMessage(hwndClient, WM_DDE_ACK,(WPARAM) hwnd,
                          PackDDElParam(WM_DDE_ACK, wStatus, aItem)))
               {
               GlobalFree(hDdePoke);
               GlobalDeleteAtom(aItem);
               }

          return 0;

     case WM_DDE_TERMINATE :
```

```
                    // Respond with another WM_DDE_TERMINATE iMsg

               hwndClient =(HWND) wParam;
               PostMessage(hwndClient, WM_DDE_TERMINATE,(WPARAM) hwnd, 0L);
               DestroyWindow(hwnd);
               return 0;

          case WM_TIMER :

                    // Post WM_DDE_DATA iMsgs for changed populations

               hwndClient =(HWND) GetWindowLong(hwnd, 0);
               hPopAdvise =(GLOBALHANDLE) GetWindowLong(hwnd, 4);
               pPopAdvise =(POPADVISE *) GlobalLock(hPopAdvise);

               for(i = 0; i < NUM_STATES; i++)
                    if(pPopAdvise[i].fAdvise)
                         if(pPopAdvise[i].lPopPrev != pop[i].lPop)
                              {
                              if(!PostDataMessage(hwnd, hwndClient, i,
                                                  pPopAdvise[i].fDeferUpd,
                                                  pPopAdvise[i].fAckReq,
                                                  FALSE))
                                   break;

                              pPopAdvise[i].lPopPrev = pop[i].lPop;
                              }

               GlobalUnlock(hPopAdvise);
               return 0;

          case WM_CLOSE :

                    // Post a WM_DDE_TERMINATE iMsg to the client

               hwndClient =(HWND) GetWindowLong(hwnd, 0);
               PostMessage(hwndClient, WM_DDE_TERMINATE,(WPARAM) hwnd, 0L);

               dwTime = GetCurrentTime();
               while(GetCurrentTime() - dwTime < DDE_TIMEOUT)
                    if(PeekMessage(&msg, hwnd, WM_DDE_TERMINATE,
                                   WM_DDE_TERMINATE, PM_REMOVE))
                         break;

               DestroyWindow(hwnd);
               return 0;

          case WM_DESTROY :
               hPopAdvise =(GLOBALHANDLE) GetWindowLong(hwnd, 4);
               GlobalFree(hPopAdvise);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }

BOOL CALLBACK TimerEnumProc(HWND hwnd, LPARAM lParam)
     {
     SendMessage(hwnd, WM_TIMER, 0, 0L);

     return TRUE;
     }

BOOL CALLBACK CloseEnumProc(HWND hwnd, LPARAM lParam)
     {
     SendMessage(hwnd, WM_CLOSE, 0, 0L);
```

```
     return TRUE;
     }

BOOL PostDataMessage(HWND hwndServer, HWND hwndClient, int iState,
                     BOOL fDeferUpd, BOOL fAckReq, BOOL fResponse)
     {
     ATOM         aItem;
     char         szPopulation[16];
     DDEACK       DdeAck;
     DDEDATA     *pDdeData;
     DWORD        dwTime;
     GLOBALHANDLE hDdeData;
     MSG          msg;
     WORD         wStatus;

     aItem = GlobalAddAtom(pop[iState].szState);

          // Allocate a DDEDATA structure if not deferred update

     if(fDeferUpd)
          {
          hDdeData = NULL;
          }
     else
          {
          wsprintf(szPopulation, "%ld\r\n", pop[iState].lPop);

          hDdeData = GlobalAlloc(GHND | GMEM_DDESHARE,
                                 sizeof(DDEDATA) + strlen(szPopulation));

          pDdeData =(DDEDATA *) GlobalLock(hDdeData);

          pDdeData->fResponse = fResponse;
          pDdeData->fRelease  = TRUE;
          pDdeData->fAckReq   = fAckReq;
          pDdeData->cfFormat  = CF_TEXT;

          lstrcpy((PSTR) pDdeData->Value, szPopulation);

          GlobalUnlock(hDdeData);
          }

          // Post the WM_DDE_DATA iMsg

     if(!PostMessage(hwndClient, WM_DDE_DATA,(WPARAM) hwndServer,
                      PackDDElParam(WM_DDE_DATA,(UINT) hDdeData, aItem)))
          {
          if(hDdeData != NULL)
               GlobalFree(hDdeData);

          GlobalDeleteAtom(aItem);
          return FALSE;
          }

          // Wait for the acknowledge iMsg if it's requested

     if(fAckReq)
          {
          DdeAck.fAck = FALSE;

          dwTime = GetCurrentTime();

          while(GetCurrentTime() - dwTime < DDE_TIMEOUT)
               {
```

```
               if(PeekMessage(&msg, hwndServer, WM_DDE_ACK, WM_DDE_ACK,
                              PM_REMOVE))
                    {
                    wStatus = LOWORD(msg.lParam);
                    DdeAck = *((DDEACK *) &wStatus);
                    aItem  = HIWORD(msg.lParam);
                    GlobalDeleteAtom(aItem);
                    break;
                    }
               }

          if(DdeAck.fAck == FALSE)
               {
               if(hDdeData != NULL)
                    GlobalFree(hDdeData);

               return FALSE;
               }
          }

     return TRUE;
     }
```

DDEPOP.H

```
/*---------------------------------------------------------------
   DDEPOP.H header file

   Data from "The World Almanac and Book of Facts 1992," page 75
  ---------------------------------------------------------------*/

#include <time.h>

struct
     {
     char       *szState;
     long        lPop70;
     long        lPop80;
     long        lPop90;
     long        lPop;
     long        lPopLast;
     long double a;
     long double b;
     long double c;
     }
     pop[] = {
               "AL",   3444354,   3894025,   4040587, 0, 0, 0.0, 0.0, 0.0,
               "AK",    302583,    401851,    550043, 0, 0, 0.0, 0.0, 0.0,
               "AZ",   1775399,   2716546,   3665228, 0, 0, 0.0, 0.0, 0.0,
               "AR",   1923322,   2286357,   2350725, 0, 0, 0.0, 0.0, 0.0,
               "CA",  19971069,  23667764,  29760021, 0, 0, 0.0, 0.0, 0.0,
               "CO",   2209596,   2889735,   3294394, 0, 0, 0.0, 0.0, 0.0,
               "CT",   3032217,   3107564,   3287116, 0, 0, 0.0, 0.0, 0.0,
               "DE",    548104,    594338,    666168, 0, 0, 0.0, 0.0, 0.0,
               "DC",    756668,    638432,    606900, 0, 0, 0.0, 0.0, 0.0,
               "FL",   6791418,   9746961,  12937926, 0, 0, 0.0, 0.0, 0.0,
               "GA",   4587930,   5462982,   6478216, 0, 0, 0.0, 0.0, 0.0,
               "HI",    769913,    964691,   1108229, 0, 0, 0.0, 0.0, 0.0,
               "ID",    713015,    944127,   1006749, 0, 0, 0.0, 0.0, 0.0,
               "IL",  11110285,  11427409,  11430602, 0, 0, 0.0, 0.0, 0.0,
               "IN",   5195392,   5490214,   5544159, 0, 0, 0.0, 0.0, 0.0,
               "IA",   2825368,   2913808,   2776755, 0, 0, 0.0, 0.0, 0.0,
               "KS",   2249071,   2364236,   2477574, 0, 0, 0.0, 0.0, 0.0,
               "KY",   3220711,   3660324,   3685296, 0, 0, 0.0, 0.0, 0.0,
               "LA",   3644637,   4206116,   4219973, 0, 0, 0.0, 0.0, 0.0,
               "ME",    993722,   1125043,   1227928, 0, 0, 0.0, 0.0, 0.0,
```

```
            "MD",   3923897,   4216933,   4781468, 0, 0, 0.0, 0.0, 0.0,
            "MA",   5689170,   5737093,   6016425, 0, 0, 0.0, 0.0, 0.0,
            "MI",   8881826,   9262044,   9295297, 0, 0, 0.0, 0.0, 0.0,
            "MN",   3806103,   4075970,   4375099, 0, 0, 0.0, 0.0, 0.0,
            "MS",   2216994,   2520770,   2573216, 0, 0, 0.0, 0.0, 0.0,
            "MO",   4677623,   4916766,   5117073, 0, 0, 0.0, 0.0, 0.0,
            "MT",    694409,    786690,    799065, 0, 0, 0.0, 0.0, 0.0,
            "NE",   1485333,   1569825,   1578385, 0, 0, 0.0, 0.0, 0.0,
            "NV",    488738,    800508,   1201833, 0, 0, 0.0, 0.0, 0.0,
            "NH",    737681,    920610,   1109252, 0, 0, 0.0, 0.0, 0.0,
            "NJ",   7171112,   7365011,   7730188, 0, 0, 0.0, 0.0, 0.0,
            "NM",   1017055,   1303302,   1515069, 0, 0, 0.0, 0.0, 0.0,
            "NY",  18241391,  17558165,  17990455, 0, 0, 0.0, 0.0, 0.0,
            "NC",   5084411,   5880095,   6628637, 0, 0, 0.0, 0.0, 0.0,
            "ND",    617792,    652717,    638800, 0, 0, 0.0, 0.0, 0.0,
            "OH",  10657423,  10797603,  10847115, 0, 0, 0.0, 0.0, 0.0,
            "OK",   2559463,   3025487,   3145585, 0, 0, 0.0, 0.0, 0.0,
            "OR",   2091533,   2633156,   2842321, 0, 0, 0.0, 0.0, 0.0,
            "PA",  11800766,  11864720,  11881643, 0, 0, 0.0, 0.0, 0.0,
            "RI",    949723,    947154,   1003464, 0, 0, 0.0, 0.0, 0.0,
            "SC",   2590713,   3120729,   3486703, 0, 0, 0.0, 0.0, 0.0,
            "SD",    666257,    690768,    696004, 0, 0, 0.0, 0.0, 0.0,
            "TN",   3926018,   4591023,   4877185, 0, 0, 0.0, 0.0, 0.0,
            "TX",  11198655,  14225513,  16986510, 0, 0, 0.0, 0.0, 0.0,
            "UT",   1059273,   1461037,   1722850, 0, 0, 0.0, 0.0, 0.0,
            "VT",    444732,    511456,    562758, 0, 0, 0.0, 0.0, 0.0,
            "VA",   4651448,   5346797,   6187358, 0, 0, 0.0, 0.0, 0.0,
            "WA",   3413244,   4132353,   4866692, 0, 0, 0.0, 0.0, 0.0,
            "WV",   1744237,   1950186,   1793477, 0, 0, 0.0, 0.0, 0.0,
            "WI",   4417821,   4705642,   4891769, 0, 0, 0.0, 0.0, 0.0,
            "WY",    332416,    469557,    453588, 0, 0, 0.0, 0.0, 0.0,
            "US", 203302031, 226542203, 248709873, 0, 0, 0.0, 0.0, 0.0
            };

#define NUM_STATES(sizeof(pop) / sizeof(pop[0]))

void CalcPops(void)
     {
     int    i;
     time_t lTime;

     time(&lTime);                    // time in seconds since 1/1/70
     lTime -= 92L * 24 * 60 * 60;      // time in seconds since 4/1/70

     for(i = 0; i < NUM_STATES; i++)
          pop[i].lPop =(long)(pop[i].a * lTime * lTime +
                              pop[i].b * lTime +
                              pop[i].c);
     }

void InitPops(void)
     {
     int         i;
     long double ldSec80, ldSec90;

     ldSec80 = 3653.0 * 24 * 60 * 60;  // seconds from 4/1/70 to 4/1/80
     ldSec90 = 7305.0 * 24 * 60 * 60;  // seconds from 4/1/70 to 4/1/90

     for(i = 0; i < NUM_STATES; i++)
          {
          pop[i].a =(ldSec90 *(pop[i].lPop80 - pop[i].lPop70) +
                     ldSec80 *(pop[i].lPop70 - pop[i].lPop90)) /
                    (ldSec90 * ldSec80 *(ldSec80 - ldSec90));

          pop[i].b =(ldSec90 * ldSec90 *(pop[i].lPop70 - pop[i].lPop80) +
```

```
                    ldSec80 * ldSec80 *(pop[i].lPop90 - pop[i].lPop70)) /
                  (ldSec90 * ldSec80 *(ldSec80 - ldSec90));

          pop[i].c = pop[i].lPop70;
          }

     CalcPops();
     }
```

DDEPOP1.RC

```
/*---------------------------
   DDEPOP1.RC resource script
  ---------------------------*/

DdePop1 ICON ddepop.ico
```

DDEPOP1.ICO

Рис. 17.1 Программа DDEPOP1

Ранее описывалось, как работает этот сервер с Microsoft Excel. Далее в этой главе будет показана программа-клиент DDE (названная SHOWPOP1), которая также использует программу DDEPOP1 в качестве сервера.

Программа DDEPOP1

Обратите внимание, что в начале листинга файла DDEPOP1.C имеется строка:

```
#include <dde.h>
```

Этот заголовочный файл содержит определения сообщений DDE и структур данных.

Далее, в программе с помощью инструкции *typedef* определяется структура (названная POPADVISE). Ниже будет показано, как используется эта структура.

В функции *WinMain* программа регистрирует два класса окна. Имя первого класса окна "DdePop1", и он используется для главного окна программы. Имя второго — "DdePop1.Server". Второй класс используется для дочерних окон, которые создаются для поддержки нескольких диалогов DDE. Для каждого диалога требуется собственное дочернее окно, созданное на основе этого класса.

Во втором классе окна поле *cbWndExtra* структуры WNDCLASSEX устанавливается так, чтобы любое окно хранило два двойных слова. Как потом станет ясно, первое будет использоваться для хранения описателя окна клиента, с которым связывается окно сервера. Вторым будет описатель блока памяти, в котором содержатся NUM_STATE структур типа POPADVISE.

После того, как создано главное окно программы DDEPOP1, вызывается функция *InitPops*. Эта функция находится в файле DDEPOP.H, вместе с действительными данными о населении и функцией *CalcPops*, которая используется для расчета прироста населения. Также, в программе DDEPOP1 вызывается функция *SetTimer*, которая устанавливает 5-секундный интервал времени для периодического обновления поля *lPop* структуры *pop* в файле DDEPOP.H.

Обратите внимание, что функция *ShowWindow* вызывается с параметром SW_SHOWMINNOACTIVE и, что *WndProc* возвращает 0 в ответ на сообщение WM_QUERYOPEN. Это приводит к тому, что DDEPOP выводится в виде кнопки в панели задач Windows 95.

Сообщение WM_DDE_INITIATE

Диалог DDE инициируется клиентом путем рассылки сообщения WM_DDE_INITIATE всем окнам верхнего уровня. (Далее, при изучении в этой главе программы-клиента, будет показано, как это реализуется путем вызова функции *SendMessage* с первым параметром HWND_BROADCAST.)

Сообщение WM_DDE_INITIATE обрабатывается сервером DDE в оконной процедуре главного окна. Как и в любом сообщении DDE, параметр *wParam* этого сообщения является описателем окна, пославшего сообщение. Им является описатель окна клиента. *WndProc* сохраняет его в переменной *hwndClient*.

В сообщении WM_DDE_INITIATE младшим словом параметра *lParam* является атом, идентифицирующий необходимое клиенту приложение. Он может быть равен NULL, если клиента устраивает ответ от любого сервера. Старшим словом параметра *lParam* является атом, идентифицирующий необходимый клиенту раздел. И снова, он может быть равен NULL, если клиента устраивает любой раздел, поддерживаемый сервером.

WndProc обрабатывает сообщение WM_DDE_INITIATE следующим образом: вызывается функция *GlobalAddAtom*, чтобы добавить в таблицу атомов атомы для имени своего приложения ("DdePop1"), и имени раздела ("US_Population"). Затем проверяется, являются ли атомы, которые находятся в младшем и старшем слове параметра *lParam*, парой этих атомов, или они равны NULL.

Если атомы совпадают, то *WndProc* на основе класса окна "DdePop1.Server" создает невидимое дочернее окно. Это окно (его оконной процедурой является *ServerProc*) будет обрабатывать все последующие DDE-сообщения диалога DDE. В первое из двух двойных слов, зарезервированных для окна, с помощью функции *SetWindowLong* заносится описатель окна клиента. Затем *WndProc* подтверждает получение сообщения WM_DDE_INITIATE, отправляя клиенту сообщение WM_DDE_ACK. Параметром *wParam* этого сообщения является описатель только что созданного сервером окна, а параметр *lParam* содержит атомы, идентифицирующие имя сервера и имя раздела. (Если клиенту требуются все разделы, и сервер поддерживает несколько разделов, то сервер должен послать клиенту несколько сообщений WM_DDE_ACK, по одному сообщению на каждый поддерживаемый сервером раздел.)

Программа, которая получает сообщение WM_DDE_ACK, ответственна за удаление всех атомов, сопровождающих это сообщение. *WndProc* вызывает функцию *GlobalDeleteAtom* для двух созданных ею атомов только в том случае, если она не посылает клиенту сообщения WM_DDE_ACK.

Сообщение WM_DDE_INITIATE и сообщение WM_DDE_ACK (в ответ на сообщение WM_DDE_INITIATE) — это единственные два сообщения DDE, которые, вместо использования функции *PostMessage*, посылаются с помощью функции *SendMessage*. Как будет показано далее в этой главе, это означает, что клиент, посылающий такое асинхронное сообщение WM_DDE_INITIATE, получает ответное асинхронное сообщение WM_DDE_ACK раньше, чем завершится исходный вызов функции *SendMessage*.

Оконная процедура *ServerProc*

С отправкой сообщения WM_DDE_ACK в ответ на сообщение WM_DDE_INITIATE начинается диалог DDE. Как уже говорилось, когда *WndProc* посылает обратно клиенту сообщение WM_DDE_ACK, параметром *wParam* этого сообщения становится описатель дочернего окна, которое она создает для диалога. Это означает, что все последующие сообщения DDE проходят между клиентом и этим дочерним окном, оконной процедурой которого является *ServerProc*.

ServerProc при обработке сообщения WM_CREATE выделяет память, необходимую для хранения NUM_STATES структур типа POPADVISE. (Об их использовании скоро будет рассказано.) Описатель этого блока памяти с помощью функции *SetWindowLong* сохраняется во втором зарезервированном двойном слове. Этот блок памяти освобождается, когда *ServerProc* получает сообщение WM_DESTROY.

Сообщение WM_DDE_REQUEST

Клиент посылает серверу синхронное сообщение WM_DDE_REQUEST, когда ему нужны данные, которые соответствуют конкретному элементу. Этот тип транзакции известен как холодная связь. Сервер отвечает, посылая клиенту синхронное сообщение WM_DDE_DATA с данными или сообщение WM_DDE_ACK, если он не может удовлетворить запрос. Рассмотрим, как *ServerProc* обрабатывает сообщение WM_DDE_REQUEST.

Как обычно у сообщений DDE, параметр *wParam* сообщения WM_DDE_REQUEST является описателем окна, пославшим синхронное сообщение, в данном случае клиента. Младшее слово параметра *lParam* представляет из себя запрашиваемый формат данных. Старшее слово параметра *lParam* представляет из себя атом, идентифицирующий запрашиваемый элемент данных.

Форматы данных DDE аналогичны форматам папки обмена, поэтому младшее слово параметра *lParam* чаще всего будет одним из идентификаторов, начинающихся с префикса CF. Клиент может послать серверу несколько сообщений WM_DDE_REQUEST для одного и того же элемента, но с разными форматами. Сервер должен ответить, используя сообщение WM_DDE_DATA только для тех форматов, которые он поддерживает. Поскольку самым используемым форматом DDE является CF_TEXT, то только этот формат поддерживает программа DDEPOP1.

Таким образом, при обработке сообщения WM_DDE_REQUEST *ServerProc* сначала проверяет, является ли запрашиваемый формат — форматом CF_TEXT. Затем *ServerProc* для получения символьной строки, соответствующей атому, переданному в старшем слове параметра *lParam*, вызывает функцию *GlobalGetAtomName*. Если клиент знает, что это должна быть двухсимвольная строка, идентифицирующая штат. Цикл *for* просматривает имеющиеся штаты с целью найти совпадение между этой строкой и полем *szState* структуры *pop*. Если это удается, то *ServerProc*, используя вызов функции *GlobalDeleteAtom*, удаляет атом, а затем вызывает функцию

PostDataMessage (функция, находящаяся в конце программы DDEPOP1, которая посылает синхронное сообщение WM_DDE_DATA. Эту функцию мы еще рассмотрим). Затем *ServerProc* завершается.

Если запрашиваемый формат не является форматом CF_TEXT, или, если не удалось найти совпадения между атомом элемента и одним из названий штатов, то *ServerProc* отправляет негативное синхронное сообщение WM_DDE_ACK, означающее, что данные не были обнаружены. Это делается путем задания в поле *fAck* структуры DDEACK (определенной в файле DDE.H) значения FALSE. Структура DDEACK преобразуется в слово, которое становится младшим словом параметра *lParam*. Старшим словом параметра *lParam* является атом запрашиваемого элемента. Вызывая функцию *PostMessage*, клиенту посылается синхронное сообщение WM_DDE_ACK.

Обратите внимание, как здесь обрабатывается атом. В документации о сообщении WM_DDE_REQUEST говорится: "Отвечая на сообщение WM_DDE_DATA или на сообщение WM_DDE_REQUEST, приложение-сервер может либо повторно использовать атом *aItem*, либо оно может удалить атом и создать новый." Это означает то, что состояние глобальной таблицы атомов не должно изменяться сервером, т. е. счетчик ссылок для атома *aItem* не должен ни увеличиваться, ни уменьшаться.

Существует три возможных варианта:

- Если требуемым форматом является CF_TEXT, и атом соответствует одному из названий штатов, то *ServerProc* перед тем, как вызвать функцию *PostDataMessage* из файла DDEPOP1.C, вызывает функцию *GlobalDeleteAtom*. Функция *PostDataMessage* (как мы скоро увидим) при отправке клиенту синхронного сообщения WM_DDE_DATA воссоздает удаленный атом.
- Если требуемый формат отличен от CF_TEXT, или, если атом не совпадает ни одним из названий штатов, то *ServerProc* для отправки клиенту негативного сообщения WM_DDE_ACK вызывает функцию *PostMessage*. В этом сообщении атом просто используется повторно.
- Однако, при неудачном вызове функции *PostMessage* (что возможно, если клиент неожиданно завершится) *ServerProc* удаляет атом, поскольку клиент этого сделать не может.

Мы еще не до конца разобрались с сообщением WM_DDE_REQUEST, поскольку еще не исследовали, как программа DDEPOP1, используя вызов функции *PostDataMessage*, посылает ответное сообщение WM_DDE_DATA. Об этом в следующем разделе.

Функция *PostDataMessage* программы DDEPOP1

Функция *PostDataMessage*, которая находится в конце файла DDEPOP1.C, предназначена для посылки клиенту синхронного сообщения WM_DDE_DATA. Кроме этого она используется для обработки сообщений WM_DDE_ADVISE (об этом будет рассказано ниже), поэтому она несколько сложнее, чем необходимо для обработки только сообщений WM_DDE_REQUEST.

У функции *PostDataMessage* имеется шесть параметров:

- *hwndServer* — описатель окна сервера
- *hwndClient* — описатель окна клиента
- *iState* — индекс массива *pop*, идентифицирующий штат, для которого требуется получить данные о населении
- *fDeferUpd* — этому параметру *ServerProc* присваивает значение FALSE при ответе на сообщение WM_DDE_REQUEST
- *fAckReq* — этому параметру *ServerProc* также, в этом случае, присваивает значение FALSE
- *fResponse* — этому параметру *ServerProc* присваивает значение TRUE для обозначения ответа на сообщение WM_DDE_REQUEST

(О параметрах *fDeferUpd* и *fAckReq* скоро будет рассказано при изучении сообщения WM_DDE_ADVISE. А пока будем игнорировать все те ситуации, в которых какому-либо из этих двух параметров функции *PostDataMessage* присваивается значение TRUE.)

Функция *PostDataMessage* начинается с вызова функции *GlobalAddItem*, которая используется для создания атома двухсимвольного имени штата. (Вспомните, что перед вызовом функции *PostDataMessage*, *ServerProc* удаляет атом.) Затем вызывается функция *wsprintf* для преобразования значения населения штата (обновляемого процедурой *ServerProc* каждые 5 секунд) в символьную строку, оканчивающуюся символами возврата каретки и перевода строки.

Затем в функции *PostDataMessage* вызывается функция *GlobalAlloc* с опцией GMEM_DDESHARE, что позволяет выделить блок памяти, достаточный для структуры DDEDATA (определенной в файле DDE.H) и реальных данных

(символьной строки *szPopulation*). В случае использования функции *PostDataMessage* при ответе на сообщение WM_DDE_REQUEST, полям структуры DDEDATA присваиваются следующие значения:

- Полю *fResponse* структуры DDEDATA присваивается значение TRUE, означающее, что данные готовятся в ответ на сообщение WM_DDE_DATA.
- Полю *fRelease* также присваивается значение TRUE, означающее, что клиент обязан освободить только что выделенный блок памяти.
- Полю *fAckReq* присваивается значение FALSE, означающее, что сообщение WM_DDE_ACK от клиента не требуется.
- Полю *cfFormat* присваивается значение CF_TEXT, означающее, что данные представлены в текстовом формате.
- Массив *szPopulation* копируется в выделенный блок памяти, начиная с поля *Value* структуры DDEDATA.

Далее в функции *PostDataMessage* используется функция *PostMessage*, которая отправляет клиенту синхронное сообщение WM_DDE_DATA. Как обычно, параметр *wParam* является описателем окна (сервера), посылающего сообщение. Параметр *lParam* чуть сложнее. Он идентифицирует объект памяти, в котором хранятся два значения: описатель блока памяти, содержащего структуру DDEDATA, и атом, идентифицирующий элемент данных (двухсимвольное имя штата). Значение параметра *lParam* формируется с помощью вызова функции *PackDDElParam*. Процедура, которая получает сообщение WM_DDE_DATA, должна извлечь эти два элемента данных из объекта памяти *lParam*. Это делается с помощью вызова функции *UnpackDDElParam*, и затем, вызов функции *FreeDDElParam* освобождает объект.

Если вызов функции *PostMessage* проходит удачно, значит задача выполнена. За освобождение блока памяти и удаление атома отвечает клиент. Если вызов функции *PostMessage* неудачен (возможно из-за того, что клиента больше в системе нет), функция *PostDataMessage* освобождает блок памяти, который она выделила, и удаляет атом.

Сообщение WM_DDE_ADVISE

Теперь, вероятно, вы начинаете понимать некоторые сложности, связанные с DDE. При работе с сообщением WM_DDE_ADVISE и горячей связью они еще более увеличатся.

Сообщение WM_DDE_REQUEST, как уже говорилось, позволяет клиенту получить данные от сервера. Но если эти данные меняются (как это происходит при приросте населения), то у клиента нет способа узнать об этом. Дать клиенту возможность узнать, что данные были изменены, является основной задачей сообщения WM_DDE_ADVISE. После получения этого сообщения, сервер становится ответственным за извещение клиента об изменении данных. (Это извещение сервер реализует путем отправки клиенту синхронного сообщения WM_DDE_DATA.) Это может быть непросто, поскольку сервер должен "запомнить", об изменении каких элементов нужно уведомлять клиента.

В сообщении WM_DDE_ADVISE параметр *lParam* идентифицирует объект, в котором хранятся, во-первых, описатель глобального блока памяти, содержащего структуру DDEDATA, как это определено в файле DDE.H, во-вторых, атом, идентифицирующий элемент данных. При обработке сообщения WM_DDE_ADVISE *ServerProc* вызывает функцию *UnpackDDElParam* для извлечения описателя и атома, а затем функцию *FreeDDElParam* для освобождения объекта *lParam*. Затем проверяется поле *cfFormat* структуры DDEADVISE на соответствие значению CF_TEXT. Если оно соответствует, то функция получает текстовую строку, на которую ссылается атом, и ищет ее совпадение с содержимым поля *szState* массива структур *pop*.

Если совпадение обнаружено, то *ServerProc* получает указатель на массив структур POPADVISE, память для которого выделена при обработке сообщения WM_CREATE. В этом массиве для каждого штата имеется структура POPADVISE. Кроме этого для каждого окна DDE имеется свой массив. Этот массив используется для хранения всей информации процедуры *ServerProc*, которая будет необходима при обновлении элементов для клиента.

Полям структуры POPADVISE, соответствующей выбранному штату присваиваются следующие значения:

- Полю *fAdvise* присваивается значение TRUE. Это просто флаг, означающий, что клиент желает обновить информацию по данному штату.
- Полю *fDeferUpd* (отсроченное обновление, deferred update) присваивается значение, содержащееся в аналогичном поле структуры DDEADVISE. Значение TRUE означает, что клиент вместо горячей связи хочет установить теплую связь. Клиент будет уведомлен об изменении данных, но без немедленной отправки ему этих данных. (В этом случае сервер посылает синхронное сообщение WM_DDE_DATA, где вместо описателя глобального блока памяти, содержащего структуру DDEDATA, указывается значение NULL. Позже клиент отправляет обычное синхронное сообщение WM_DDE_REQUEST для фактического

получения данных.) Значение FALSE означает, что клиент хочет получить данные в сообщении WM_DDE_DATA.

- Полю *fAckReq* (необходимо подтверждение, acknowledgment requested) присваивается значение, содержащееся в аналогичном поле структуры DDEADVISE. Это очень интересное значение. Значение TRUE извещает сервер о необходимости отправить синхронное сообщение WM_DDE_DATA, где полю *fAckReq* структуры DDEDATA присвоить значение TRUE. Это означает, что клиенту требуется подтвердить получение сообщения WM_DDE_DATA сообщением WM_DDE_ACK. Значение TRUE не означает, что клиенту требуется от сервера сообщение WM_DDE_ACK; оно означает, что сервер требует от клиента отправить ему сообщение WM_DDE_ACK, когда он позднее отправит клиенту синхронное сообщение WM_DDE_DATA.
- Полю *lPopPrev* присваивается значение, соответствующее текущему населению штата. *ServerProc* использует это поле для определения необходимости извещения клиента об изменении численности населения штата.

Процедура *ServerProc* заканчивает работу со структурой DDEADVISE и освобождает блок памяти, как это описано в документации о сообщении WM_DDE_ADVISE. Теперь *ServerProc* должна подтвердить получение сообщения WM_DDE_ADVISE путем отправки позитивного синхронного сообщения WM_DDE_ACK. При этом полю *fAck* структуры DDEACK присваивается значение TRUE. Если вызов функции *PostMessage* неудачен, то *ServerProc* удаляет атом.

Если формат данных — не CF_TEXT, или если не удалось подобрать совпадающего имени штата, то *ServerProc* посылает негативное синхронное сообщение WM_DDE_ACK. В этом случае, если вызов функции *PostMessage* неудачен, то *ServerProc* и удаляет атом, и освобождает блок памяти структуры DDEADVISE.

Теоретически, обработка сообщения WM_DDE_ADVISE на этом завершается. Однако, клиент просил, чтобы его известили о любых изменениях элемента данных. Если предположить, что клиенту не известно ни одного значения элемента данных, то необходимо, чтобы *ServerProc* отправила клиенту синхронное сообщение WM_DDE_DATA.

Это делается с помощью функции *PostDataMessage*, но при этом третьему параметру присваивается значение поля *fDeferUpd* структуры POPADVISE, четвертому параметру — значение поля *fAckReq* структуры POPADVISE, и последнему параметру — значение FALSE (означающее, что синхронное сообщение WM_DDE_DATA послано в ответ на сообщение WM_DDE_ADVISE, а не в ответ на сообщение WM_DDE_REQUEST).

Настало время вернуться к функции *PostDataMessage*. Обратите внимание на начало функции, если параметру *fDeferUpd* присваивается значение TRUE, то функция не выделяет память для описателя *hDdeData*, а просто устанавливает его в NULL.

Если параметр *fAckReq* равен TRUE, то функция *PostDataMessage*, после отправки синхронного сообщения WM_DDE_DATA, ждет от клиента сообщения WM_DDE_ACK. Это делается при помощи вызова функции *PeekMessage*. Далее функция *PostDataMessage* удаляет атом в сообщении WM_DDE_ACK. Если сообщение WM_DDE_ACK не приходит в течение трех секунд, или если сообщение содержит негативное подтверждение, то функция *PostDataMessage* освобождает блок данных, содержащий структуру DDEDATA.

Если вы думаете, что вам удастся избежать части этой работы, если предположить, что клиент никогда не пошлет сообщения WM_DDE_ADVISE с полями "отсроченное обновление" и "необходимо подтверждение" структуры POPADVISE, установленными в TRUE, то оставьте эти мысли. Именно это делает Microsoft Excel, устанавливая теплую связь с подтверждением получения сообщений WM_DDE_DATA.

Обновление элементов данных

После обработки сообщения WM_DDE_ADVISE, серверу требуется известить клиента о том, что элемент данных был изменен. То, как это реализовано, зависит от сервера. В программе DDEPOP1 для того, чтобы каждые пять секунд пересчитывать численность населения, используется таймер. Пересчет происходит при обработке в *WndProc* сообщения WM_TIMER.

Затем *WndProc* вызывает функцию *EnumChildWindows* с функцией *TimerEnumProc* (расположенной в файле DDEPOP1.C после процедуры *ServerProc*). Функция *TimerEnumProc* посылает всем дочерним окнам асинхронные сообщения WM_TIMER. Все эти окна будут использовать оконную процедуру *ServerProc*.

ServerProc обрабатывает сообщение WM_TIMER, просматривая все имена штатов и проверяя, присвоено ли полю *fAdvise* структуры POPADVISE значение TRUE и изменилась ли численность населения. Если да, то для отправки клиенту синхронного сообщения WM_DDE_DATA вызывается функция *PostDataMessage*.

Сообщение WM_DDE_UNADVISE

Сообщение WM_DDE_UNADVISE требует от сервера прекратить отправку синхронных сообщений WM_DDE_DATA при изменении элемента данных. Младшим словом параметра *lParam* этого сообщения является

либо формат данных, либо NULL, что означает — данные всех форматов. Старшим словом параметра *lParam* этого сообщения является либо элемент ATOM, либо NULL, что означает — все элементы.

В программе DDEPOP1 сообщение WM_DDE_UNADVISE обрабатывается путем присвоения соответствующим полям *fAdvise* структуры POPADVISE значения FALSE и последующим позитивным или негативным подтверждением путем посылки сообщения WM_DDE_ACK.

Сообщение WM_DDE_TERMINATE

Когда клиент решает закончить диалог, он посылает серверу синхронное сообщение WM_DDE_TERMINATE. Сервер просто отвечает клиенту собственным сообщением WM_DDE_TERMINATE. Кроме этого, *ServerProc* после получения сообщения WM_DDE_TERMINATE удаляет дочернее окно, поскольку оно больше не нужно, и диалог, который это окно поддерживало, завершается.

ServerProc также обрабатывает сообщения WM_DDE_POKE и WM_DDE_EXECUTE, но в обоих случаях она просто отвечает негативным подтверждением получения.

Если программа DDEPOP1 закрывается из собственного системного меню, то она должна завершить все диалоги со своими клиентами. Поэтому, когда *WndProc* получает сообщение WM_CLOSE, она вызывает функцию *EnumChildWindows* с функцией *CloseEnumProc*. Функция *CloseEnumProc* посылает всем дочерним окнам асинхронное сообщение WM_CLOSE.

ServerProc отвечает на сообщение WM_CLOSE путем отправки клиенту синхронного сообщения WM_DDE_TERMINATE и затем ждет от клиента ответного сообщения WM_DDE_TERMINATE.

Программа-клиент DDE

Теперь, после того, как мы рассмотрели программу-сервер DDE, которую можно использовать для работы с Microsoft Excel, рассмотрим программу-клиент DDE, сервером для которой является программа DDEPOP1. Эта программа, названная SHOWPOP1, представлена на рис. 17.2.

SHOWPOP1.MAK

```
#------------------------
# SHOWPOP1.MAK make file
#------------------------

showpop1.exe : showpop1.obj
     $(LINKER) $(GUIFLAGS) -OUT:showpop1.exe showpop1.obj $(GUILIBS)

showpop1.obj : showpop1.c showpop.h
     $(CC) $(CFLAGS) showpop1.c
```

SHOWPOP1.C

```
/*---------------------------------------
   SHOWPOP1.C -- DDE Client using DDEPOP1
                 (c) Charles Petzold, 1996
  ---------------------------------------*/

#include <windows.h>
#include <dde.h>
#include <stdlib.h>
#include <string.h>
#include "showpop.h"

#define WM_USER_INITIATE(WM_USER + 1)
#define DDE_TIMEOUT      3000

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

char szAppName[] = "ShowPop1";

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
     {
     HWND       hwnd;
     MSG        msg;
```

```
     WNDCLASSEX wndclass;

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

     hwnd = CreateWindow(szAppName, "DDE Client - US Population",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

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

     SendMessage(hwnd, WM_USER_INITIATE, 0, 0L);

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

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static BOOL   fDoingInitiate = TRUE;
     static char   szServerApp[]  = "DdePop1",
                   szTopic[]      = "US_Population";
     static HWND   hwndServer = NULL;
     static long   cxChar, cyChar;
     ATOM          aApp, aTop, aItem;
     char          szBuffer[24], szPopulation[16], szItem[16];
     DDEACK        DdeAck;
     DDEDATA      *pDdeData;
     DDEADVISE    *pDdeAdvise;
     DWORD         dwTime;
     GLOBALHANDLE  hDdeAdvise, hDdeData;
     HDC           hdc;
     MSG           msg;
     PAINTSTRUCT   ps;
     short         i;
     long          x, y;
     TEXTMETRIC    tm;
     WORD          wStatus;
     UINT          uiLow, uiHi;

     switch(iMsg)
          {
          case WM_CREATE :
               hdc = GetDC(hwnd);
               GetTextMetrics(hdc, &tm);
               cxChar = tm.tmAveCharWidth;
               cyChar = tm.tmHeight + tm.tmExternalLeading;
```

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

     case WM_USER_INITIATE :

               // Broadcast WM_DDE_INITIATE iMsg

          aApp = GlobalAddAtom(szServerApp);
          aTop = GlobalAddAtom(szTopic);

          SendMessage(HWND_BROADCAST, WM_DDE_INITIATE,(WPARAM) hwnd,
                      MAKELONG(aApp, aTop));

               // If no response, try loading DDEPOP first

          if(hwndServer == NULL)
               {
               WinExec(szServerApp, SW_SHOWMINNOACTIVE);

               SendMessage(HWND_BROADCAST, WM_DDE_INITIATE,(WPARAM) hwnd,
                           MAKELONG(aApp, aTop));
               }

               // Delete the atoms

          GlobalDeleteAtom(aApp);
          GlobalDeleteAtom(aTop);
          fDoingInitiate = FALSE;

               // If still no response, display message box

          if(hwndServer == NULL)
               {
               MessageBox(hwnd, "Cannot connect with DDEPOP1.EXE!",
                          szAppName, MB_ICONEXCLAMATION | MB_OK);

               return 0;
               }

               // Post WM_DDE_ADVISE iMsgs

          for(i = 0; i < NUM_STATES; i++)
               {
               hDdeAdvise = GlobalAlloc(GHND | GMEM_DDESHARE,
                                        sizeof(DDEADVISE));

               pDdeAdvise =(DDEADVISE *) GlobalLock(hDdeAdvise);

               pDdeAdvise->fAckReq   = TRUE;
               pDdeAdvise->fDeferUpd = FALSE;
               pDdeAdvise->cfFormat  = CF_TEXT;

               GlobalUnlock(hDdeAdvise);

               aItem = GlobalAddAtom(pop[i].szAbb);

               if(!PostMessage(hwndServer, WM_DDE_ADVISE,(WPARAM) hwnd,
                               PackDDElParam(WM_DDE_ADVISE,
                                             (UINT) hDdeAdvise, aItem)))
                    {
                    GlobalFree(hDdeAdvise);
                    GlobalDeleteAtom(aItem);
                    break;
                    }
```

```
                DdeAck.fAck = FALSE;

                dwTime = GetCurrentTime();

                while(GetCurrentTime() - dwTime < DDE_TIMEOUT)
                     {
                     if(PeekMessage(&msg, hwnd, WM_DDE_ACK,
                                     WM_DDE_ACK, PM_REMOVE))
                          {
                          GlobalDeleteAtom(HIWORD(msg.lParam));

                          wStatus = LOWORD(msg.lParam);
                          DdeAck = *((DDEACK *) &wStatus);

                          if(DdeAck.fAck == FALSE)
                               GlobalFree(hDdeAdvise);

                          break;
                          }
                     }

                if(DdeAck.fAck == FALSE)
                     break;

                while(PeekMessage(&msg, hwnd, WM_DDE_FIRST,
                                  WM_DDE_LAST, PM_REMOVE))
                     {
                     DispatchMessage(&msg);
                     }
                }

           if(i < NUM_STATES)
                {
                MessageBox(hwnd, "Failure on WM_DDE_ADVISE!",
                            szAppName, MB_ICONEXCLAMATION | MB_OK);
                }
           return 0;

      case WM_DDE_ACK :

                // In response to WM_DDE_INITIATE, save server window

           if(fDoingInitiate)
                {
                UnpackDDElParam(WM_DDE_ACK, lParam, &uiLow, &uiHi);
                FreeDDElParam(WM_DDE_ACK, lParam);
                hwndServer =(HWND) wParam;
                GlobalDeleteAtom((ATOM) uiLow);
                GlobalDeleteAtom((ATOM) uiHi);
                }
           return 0;

      case WM_DDE_DATA :

                // wParam -- sending window handle
                // lParam -- DDEDATA memory handle & item atom

           UnpackDDElParam(WM_DDE_DATA, lParam, &uiLow, &uiHi);
           FreeDDElParam(WM_DDE_DATA, lParam);

           hDdeData  =(GLOBALHANDLE) uiLow;
           pDdeData =(DDEDATA *) GlobalLock(hDdeData);
           aItem     =(ATOM) uiHi;

                // Initialize DdeAck structure
```

```
          DdeAck.bAppReturnCode = 0;
          DdeAck.reserved       = 0;
          DdeAck.fBusy          = FALSE;
          DdeAck.fAck           = FALSE;

               // Check for matching format and data item

          if(pDdeData->cfFormat == CF_TEXT)
               {
               GlobalGetAtomName(aItem, szItem, sizeof(szItem));

               for(i = 0; i < NUM_STATES; i++)
                    if(strcmp(szItem, pop[i].szAbb) == 0)
                         break;

               if(i < NUM_STATES)
                    {
                    strcpy(szPopulation,(char *) pDdeData->Value);
                    pop[i].lPop = atol(szPopulation);
                    InvalidateRect(hwnd, NULL, FALSE);

                    DdeAck.fAck = TRUE;
                    }
               }

               // Acknowledge if necessary

          if(pDdeData->fAckReq == TRUE)
               {
               wStatus = *((WORD *) &DdeAck);

               if(!PostMessage((HWND) wParam, WM_DDE_ACK,(WPARAM) hwnd,
                                PackDDElParam(WM_DDE_ACK,
                                             wStatus, aItem)))
                    {
                    GlobalDeleteAtom(aItem);
                    GlobalUnlock(hDdeData);
                    GlobalFree(hDdeData);
                    return 0;
                    }
               }
          else
               {
               GlobalDeleteAtom(aItem);
               }

               // Clean up

          if(pDdeData->fRelease == TRUE || DdeAck.fAck == FALSE)
               {
               GlobalUnlock(hDdeData);
               GlobalFree(hDdeData);
               }
          else
               {
               GlobalUnlock(hDdeData);
               }

          return 0;

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

          for(i = 0; i < NUM_STATES; i++)
```

```
                {
                if(i <(NUM_STATES + 1) / 2)
                     {
                     x = cxChar;
                     y = i * cyChar;
                     }
                else
                     {
                     x = 44 * cxChar;
                     y =(i -(NUM_STATES + 1) / 2) * cyChar;
                     }

                TextOut(hdc, x, y, szBuffer,
                         wsprintf(szBuffer, "%-20s",
                                  (PSTR) pop[i].szState));

                x += 36 * cxChar;

                SetTextAlign(hdc, TA_RIGHT | TA_TOP);

                TextOut(hdc, x, y, szBuffer,
                         wsprintf(szBuffer, "%10ld", pop[i].lPop));

                SetTextAlign(hdc, TA_LEFT | TA_TOP);
                }

           EndPaint(hwnd, &ps);
           return 0;

      case WM_DDE_TERMINATE :

                // Respond with another WM_DDE_TERMINATE iMsg

           PostMessage(hwndServer, WM_DDE_TERMINATE,(WPARAM) hwnd, 0L);
           hwndServer = NULL;
           return 0;

      case WM_CLOSE :
           if(hwndServer == NULL)
                break;

                // Post WM_DDE_UNADVISE iMsg

           PostMessage(hwndServer, WM_DDE_UNADVISE,(WPARAM) hwnd,
                       MAKELONG(CF_TEXT, NULL));

           dwTime = GetCurrentTime();

           while(GetCurrentTime() - dwTime < DDE_TIMEOUT)
                {
                if(PeekMessage(&msg, hwnd, WM_DDE_ACK,
                               WM_DDE_ACK, PM_REMOVE))
                     break;
                }
                // Post WM_DDE_TERMINATE iMsg

           PostMessage(hwndServer, WM_DDE_TERMINATE,(WPARAM) hwnd, 0L);

           dwTime = GetCurrentTime();

           while(GetCurrentTime() - dwTime < DDE_TIMEOUT)
                {
                if(PeekMessage(&msg, hwnd, WM_DDE_TERMINATE,
                               WM_DDE_TERMINATE, PM_REMOVE))
                     break;
```

```
                    }

                break;             // for default processing

           case WM_DESTROY :
                PostQuitMessage(0);
                return 0;
           }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

SHOWPOP.H

```
/*----------------------
   SHOWPOP.H header file
  ----------------------*/

struct
     {
     char *szAbb;
     char *szState;
     long  lPop;
     }
     pop[] = {
             "AL", "Alabama",             0, "AK", "Alaska",             0,
             "AZ", "Arizona",             0, "AR", "Arkansas",           0,
             "CA", "California",          0, "CO", "Colorado",           0,
             "CT", "Connecticut",         0, "DE", "Delaware",           0,
             "DC", "Dist. of Columbia",   0, "FL", "Florida",            0,
             "GA", "Georgia",             0, "HI", "Hawaii",             0,
             "ID", "Idaho",               0, "IL", "Illinois",           0,
             "IN", "Indiana",             0, "IA", "Iowa",               0,
             "KS", "Kansas",              0, "KY", "Kentucky",           0,
             "LA", "Louisiana",           0, "ME", "Maine",              0,
             "MD", "Maryland",            0, "MA", "Massachusetts",      0,
             "MI", "Michigan",            0, "MN", "Minnesota",          0,
             "MS", "Mississippi",         0, "MO", "Missouri",           0,
             "MT", "Montana",             0, "NE", "Nebraska",           0,
             "NV", "Nevada",              0, "NH", "New Hampshire",      0,
             "NJ", "New Jersey",          0, "NM", "New Mexico",         0,
             "NY", "New York",            0, "NC", "North Carolina",     0,
             "ND", "North Dakota",        0, "OH", "Ohio",               0,
             "OK", "Oklahoma",            0, "OR", "Oregon",             0,
             "PA", "Pennsylvania",        0, "RI", "Rhode Island",       0,
             "SC", "South Carolina",      0, "SD", "South Dakota",       0,
             "TN", "Tennessee",           0, "TX", "Texas",              0,
             "UT", "Utah",                0, "VT", "Vermont",            0,
             "VA", "Virginia",            0, "WA", "Washington",         0,
             "WV", "West Virginia",       0, "WI", "Wisconsin",          0,
             "WY", "Wyoming",             0, "US", "United States Total", 0
             };

#define NUM_STATES(sizeof(pop) / sizeof(pop[0]))
```

Рис. 17.2 Программа SHOWPOP1

Эта программа отображает в своем окне названия штатов с обновленной численностью населения из программы DDEPOP1, используя возможности сообщения WM_DDE_ADVISE. Обратите внимание, что в программе SHOWPOP1 имеется точно такая же структура *pop*, как и в программе DDEPOP1, но в данной версии содержатся поля для двухбуквенных аббревиатур штатов, для названия штатов, и поле *lPop* (инициализируемое нулем), в которое будет заноситься обновленная численность населения, получаемая из программы DDEPOP1.

В программе SHOWPOP1 поддерживается только один диалог DDE, поэтому для этого диалога ей достаточно одного окна. Для него она использует оконную процедуру *WndProc*.

Инициирование диалога DDE

В программе SHOWPOP1 диалог инициируется после вызова в *WinMain* функции *UpdateWindow* путем отправки из *WndProc* асинхронного, определяемого пользователем сообщения WM_USER_INITIATE. Как правило клиент инициирует диалог с помощью команды меню.

В ответ на это, определяемое пользователем сообщение, *WndProc* вызывает функцию *GlobalAddAtom* для создания атомов для имени приложения сервера ("DdePop1") и имени раздела ("US_Population"). *WndProc* посылает широковещательное асинхронное сообщение WM_DDE_INITIATE с помощью вызова функции *SendMessage* с описателем окна HWND_BROADCAST.

Как мы знаем, сервер, который ищет совпадения атомов приложения и раздела, должен отправить клиенту ответное асинхронное сообщение WM_DDE_ACK. Поскольку это сообщение асинхронное, то есть оно посылается функцией *SendMessage*, а не *PostMessage*, то клиент получит сообщение WM_DDE_ACK раньше, чем завершится исходный вызов функции *SendMessage* для посылки сообщения WM_DDE_INITIATE. *WndProc* обрабатывает сообщение WM_DDE_ACK, сохраняя описатель окна сервера в переменной *hwndServer*, и удаляя атомы, сопутствующие сообщению.

Если клиент рассылает широковещательное сообщение WM_DDE_INITIATE с именами приложения и раздела равными NULL, он должен быть готов к получению нескольких сообщений WM_DDE_ACK от каждого сервера, который может удовлетворить его запрос. В этом случае клиент должен решить, услугами какого сервера ему воспользоваться. Остальным нужно отправить синхронные сообщения WM_DDE_TERMINATE для завершения диалога.

Возможна ситуация, когда *hwndServer* будет равным NULL и после вызова функции *SendMessage* для посылки сообщения WM_DDE_INITIATE. Это означает, что программа DDEPOP1 не запущена в системе. В этом случае *WndProc* пытается запустить программу DDEPOP1, используя для этого вызов функции *WinExec*. Вызов функции *WinExec* для загрузки программы DDEPOP1 ищет ее в текущем каталоге и в каталогах, указанных в переменной окружения PATH. Затем *WndProc* снова рассылает асинхронное сообщение WM_DDE_INITIATE. Если *hwndServer* по-прежнему равен NULL, то *WndProc* выводит на экран окно сообщений, извещающее пользователя о наличии ошибки.

Далее, для каждого штата, из перечисленных в структуре *pop*, *WndProc* с помощью вызова функции *GlobalAlloc* выделяет структуру DDEADVISE. Флагу *fAckReq* присваивается значение TRUE (означающее, что сервер должен послать синхронное сообщение WM_DDE_DATA с полем *fAckReq* структуры DDEDATA равным NULL). Флагу *fDeferUpd* ("отсроченное обновление — deferred update") присваивается значение FALSE (означающее не теплую связь, а горячую), и полю *cfFormat* присваивается значение CF_TEXT. Функция *GlobalAddAtom* добавляет атом для двухбуквенной аббревиатуры штата.

Эта структура и атом передаются серверу, когда программа SHOWPOP1 отправляет ему синхронное сообщение WM_DDE_ADVISE. Если вызов функции *PostMessage* неудачен (что может случиться, если программа DDEPOP1 внезапно завершилась), то программа SHOWPOP1 освобождает блок памяти, удаляет атом и покидает цикл. В противном случае программа SHOWPOP1 ожидает сообщения WM_DDE_ACK, используя для этого функцию *PeekMessage*. Как сказано в документации по DDE, клиент удаляет атом, сопутствующий сообщению, и, если сервер отвечает негативным подтверждением, освобождает также блок памяти.

Вполне вероятно, что за сообщением WM_DDE_ACK от сервера последует сообщение WM_DDE_DATA для элемента данных. По этой причине, программа SHOWPOP1 вызывает функцию *PeekMessage* и функцию *DispatchMessage* для извлечения сообщений DDE из очереди сообщений и передачи их в *WndProc*.

Сообщение WM_DDE_DATA

Следом за сообщениями WM_DDE_ADVISE *WndProc* будет получать от сервера сообщения WM_DDE_DATA, содержащие обновленные данные о населении. Как ранее отмечалось, *lParam* является объектом памяти, в котором находятся описатель блока памяти со структурой DDEDATA и атом, идентифицирующий элемент данных. Эта информация извлекается из сообщения функцией *UnpackDDElParam*.

Программа SHOWPOP1 проверяет, равно ли поле *cfFormat* структуры DDEDATA значению CF_TEXT. (Нам конечно известно, что в программе DDEPOP1 используется только формат CF_TEXT, но для завершенности программы такая проверка необходима.) Затем с помощью функции *GlobalGetAtomName* она получает текстовую строку, соответствующую атому элемента. Эта текстовая строка представляет собой двухбуквенную аббревиатуру штата.

Используя цикл *for* программа SHOWPOP1 просматривает список штатов в поисках совпадения. Если таковое обнаруживается, она копирует данные о населении из структуры DDEDATA в массив *szPopulation*, преобразуя его в длинное целое с помощью функции языка C *atol*, сохраняет его в структуре pop и делает окно недействительным.

Теперь осталось только освободить захваченные ресурсы. Если от клиента требуется подтверждение сообщения WM_DDE_DATA, *WndProc* его посылает. Если подтверждения не требуется (или вызов функции *PostMessage*

неудачен), атом *aItem* удаляется. Если вызов функции *PostMessage* неудачен, или если не удалось найти соответствующий штат (что обозначается негативным подтверждением), или если флаг *fRelease* структуры DDEDATA имеет значение TRUE, то программа SHOWPOP1 освобождает блок памяти.

Первоначально, программа SHOWPOP1 была написана так, что она отправляла синхронные сообщения WM_DDE_ADVISE с полем *fAckReq* структуры DDEADVISE равным FALSE. Это показывало серверу, что ему нужно отправлять синхронные сообщения WM_DDE_DATA с полем *fAckReq* структуры DDEDATA равным FALSE, что, в свою очередь, показывало клиенту, что не нужно отправлять серверу синхронных сообщений WM_DDE_ACK, подтверждающих получение сообщений WM_DDE_DATA. Это хорошо работало при обычном обновлении данных. Однако, если во время работы программы SHOWPOP1 изменить системное время Windows, то программа DDEPOP1 посылает программе SHOWPOP1 52 синхронных сообщения WM_DDE_DATA, не ожидая их подтверждения. Это приводило к переполнению очереди сообщений программы SHOWPOP1 и она теряла большую часть обновленных данных о населении.

Урок ясен: если клиент хочет, чтобы его извещали об изменении нескольких элементов данных одновременно, он должен присвоить полю *fAckReq* структуры DDEADVISE значение TRUE. Это единственный безопасный вариант.

Сообщение WM_DDE_TERMINATE

Обработка отправленного сервером синхронного сообщения WM_DDE_TERMINATE достаточно проста: программа SHOWPOP1 просто отправляет обратно другое синхронное сообщение WM_DDE_TERMINATE и присваивает переменной *hwndServer* значение NULL (означающее окончание диалога).

Если программа SHOWPOP1 завершается (что инициируется сообщением WM_CLOSE), то сначала она посылает серверу синхронное сообщение WM_DDE_UNADVISE, чтобы предотвратить все будущие обновления данных. Это достигается присвоением атому элемента значения NULL, означающее все элементы данных. Затем программа SHOWPOP1 посылает серверу синхронное сообщение WM_DDE_TERMINATE и ожидает от него ответного сообщения WM_DDE_TERMINATE.

Управляющая библиотека DDE

Из-за сложностей, с которыми столкнулись многие программисты при использовании DDE, Microsoft, начиная с версии Windows 3.1, решила предоставить им новые средства. Управляющая библиотека DDE (DDE management library, DDEML) преодолевает многие сложности DDE, путем инкапсуляции сообщений, управления атомами и управления памятью в теле функций.

При первом знакомстве DDEML может обескуражить. В заголовочных файлах DDEML.H и DDE.H определено 30 функций, начинающихся с префикса *Dde*. Для программы, использующей DDEML, также требуется функция обратного вызова, которая может обрабатывать 16 типов транзакций. Они представляют из себя константы, определенные в файле DDEML.H и начинающиеся с префикса XTYP.

Вся прелесть в том, что DDEML действительно упрощает программирование DDE. Это станет очевидным при сравнении размеров исходного текста функционально эквивалентных программ. В файлах DDEPOP2.C и SHOWPOP2.C приведены исходные тексты программ, использующих DDEML, а в приведенных ранее файлах DDEPOP1 и SHOWPOP1 — исходные тексты программ, не использующих DDEML.

Концептуальные различия

Имеется несколько терминологических и концептуальных различий при использовании классического DDE и DDEML. Во-первых, хотя транзакции DDE по-прежнему основаны на именах приложения, раздела и элемента, в DDEML имя приложения называется сервисом (service).

Любая программа, использующая DDEML, должна сначала зарегистрировать себя в управляющей библиотеке DDE, что делается с помощью вызова функции *DdeInitialize*. Параметром функции является идентификатор экземпляра приложения, который передается всем остальным функциям DDEML. Когда программа завершается, вызывается функция *DdeUninitialize*.

Важнейшее отличие между DDEML и классическим DDE состоит в том, что программа обрабатывает транзакции DDE в функции обратного вызова, а не в оконной процедуре. Эта функция обратного вызова регистрируется в системе при вызове функции *DdeInitialize*.

При передаче данных от одного приложения к другому программа прямо не выделяет совместно используемую память. Вместо этого программа с помощью вызова функции *DdeCreateDataHandle* создает описатель данных буфера, содержащего данные. Программа, которой предназначены данные, получает эти данные из описателя, используя для этого функцию *DdeAccessData* или *DdeGetData*. Описатель данных освобождается при вызове функции *DdeFreeDataHandle*.

Вместо использования атомов, программа DDEML использует описатели строк, которые хранятся в переменных типа HSZ. Описатель символьной строки создает функция *DdeCreateStringHandle*; функция *DdeQueryString* получат строку, соответствующую описателю; функция *DdeCmpStringHandles* сравнивает строки, представленные двумя описателями строк; функция *DdeFreeStringHandle* освобождает описатель строки. Иногда описатели строк освобождаются автоматически. Программа, с помощью вызова функции *DdeKeepStringHandle*, может оставить описатель строки действительным.

Реализация DDE с помощью DDEML

Для того чтобы увидеть, как DDEML работает в реальном приложении, рассмотрим пример. На рис. 17.3 представлен исходный текст программы сервера DDEML, которая называется DDEPOP2, а на рис. 17.4 представлен исходный текст программы клиента DDEML, которая называется SHOWPOP2. Для программы DDEPOP2 также требуются файлы DDEPOP.H и DDEPOP.ICO, представленные на рис. 17.1, а для программы SHOWPOP2 — файл SHOWPOP.H, представленный на рис. 17.2.

DDEPOP2.MAK

```
#----------------------
# DDEPOP2.MAK make file
#----------------------

ddepop2.exe : ddepop2.obj ddepop2.res
     $(LINKER) $(GUIFLAGS) -OUT:ddepop2.exe ddepop2.obj ddepop2.res $(GUILIBS)

ddepop2.obj : ddepop2.c ddepop.h
     $(CC) $(CFLAGS) ddepop2.c

ddepop2.res : ddepop2.rc ddepop.ico
     $(RC) $(RCVARS) ddepop2.rc
```

DDEPOP2.C

```
/*-----------------------------------------------
   DDEPOP2.C -- DDEML Server for Population Data
                (c) Charles Petzold, 1996
  ----------------------------------------------*/

#include <windows.h>
#include <ddeml.h>
#include <string.h>
#include "ddepop.h"

#define WM_USER_INITIATE(WM_USER + 1)
#define ID_TIMER         1

LRESULT  CALLBACK WndProc    (HWND, UINT, WPARAM, LPARAM);
HDDEDATA CALLBACK DdeCallback(UINT, UINT, HCONV, HSZ, HSZ,
                                       HDDEDATA, DWORD, DWORD);

char      szAppName[] = "DdePop2";
char      szTopic[]   = "US_Population";
DWORD     idInst;
HINSTANCE hInst;
HWND      hwnd;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
     {
     MSG        msg;
     WNDCLASSEX wndclass;

     wndclass.cbSize        = sizeof(wndclass);
     wndclass.style         = 0;
     wndclass.lpfnWndProc   = WndProc;
     wndclass.cbClsExtra    = 0;
     wndclass.cbWndExtra    = 0;
```

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

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "DDEML Population Server",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);
     ShowWindow(hwnd, SW_SHOWMINNOACTIVE);
     UpdateWindow(hwnd);

               // Initialize for using DDEML

     if(DdeInitialize(&idInst,(PFNCALLBACK) &DdeCallback,
                      CBF_FAIL_EXECUTES | CBF_FAIL_POKES |
                      CBF_SKIP_REGISTRATIONS | CBF_SKIP_UNREGISTRATIONS, 0))
          {
          MessageBox(hwnd, "Could not initialize server!",
                     szAppName, MB_ICONEXCLAMATION | MB_OK);

          DestroyWindow(hwnd);
          return FALSE;
          }

                // Set the timer

     SetTimer(hwnd, ID_TIMER, 5000, NULL);

               // Start things going

     SendMessage(hwnd, WM_USER_INITIATE, 0, 0L);

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

               // Clean up

     DdeUninitialize(idInst);
     KillTimer(hwnd, ID_TIMER);

     return msg.wParam;
     }

int GetStateNumber(UINT iFmt, HSZ hszItem)
     {
     char szItem[32];
     int  i;

     if(iFmt != CF_TEXT)
          return -1;

     DdeQueryString(idInst, hszItem, szItem, sizeof(szItem), 0);

     for(i = 0; i < NUM_STATES; i++)
          if(strcmp(szItem, pop[i].szState) == 0)
```

```
               break;
     if(i >= NUM_STATES)
          return -1;

     return i;
     }

HDDEDATA CALLBACK DdeCallback(UINT iType, UINT iFmt, HCONV hConv,
                              HSZ hsz1, HSZ hsz2, HDDEDATA hData,
                              DWORD dwData1, DWORD dwData2)
     {
     char szBuffer[32];
     int  i;

     switch(iType)
          {
          case XTYP_CONNECT :            // hsz1 = topic
                                         // hsz2 = service

               DdeQueryString(idInst, hsz2, szBuffer, sizeof(szBuffer), 0);

               if(0 != strcmp(szBuffer, szAppName))
                    return FALSE;

               DdeQueryString(idInst, hsz1, szBuffer, sizeof(szBuffer), 0);

               if(0 != strcmp(szBuffer, szTopic))
                    return FALSE;

               return(HDDEDATA) TRUE;

          case XTYP_ADVSTART :           // hsz1 = topic
                                         // hsz2 = item

                    // Check for matching format and data item

               if(-1 ==(i = GetStateNumber(iFmt, hsz2)))
                    return FALSE;

               pop[i].lPopLast = 0;
               PostMessage(hwnd, WM_TIMER, 0, 0L);

               return(HDDEDATA) TRUE;

          case XTYP_REQUEST :
          case XTYP_ADVREQ :             // hsz1 = topic
                                         // hsz2 = item

                    // Check for matching format and data item

               if(-1 ==(i = GetStateNumber(iFmt, hsz2)))
                    return NULL;
               wsprintf(szBuffer, "%ld\r\n", pop[i].lPop);

               return DdeCreateDataHandle(idInst,(unsigned char *) szBuffer,
                                          strlen(szBuffer) + 1,
                                          0, hsz2, CF_TEXT, 0);

          case XTYP_ADVSTOP :            // hsz1 = topic
                                         // hsz2 = item

                    // Check for matching format and data item

               if(-1 ==(i = GetStateNumber(iFmt, hsz2)))
                    return FALSE;
```

```
               return(HDDEDATA) TRUE;
          }

     return NULL;
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HSZ hszService, hszTopic;
     HSZ        hszItem;
     int        i;

     switch(iMsg)
          {
          case WM_USER_INITIATE :
               InitPops();

               hszService = DdeCreateStringHandle(idInst, szAppName, 0);
               hszTopic   = DdeCreateStringHandle(idInst, szTopic,   0);

               DdeNameService(idInst, hszService, NULL, DNS_REGISTER);
               return 0;

          case WM_TIMER :
          case WM_TIMECHANGE :

                    // Calculate new current populations

               CalcPops();

               for(i = 0; i < NUM_STATES; i++)
                    if(pop[i].lPop != pop[i].lPopLast)
                         {
                         hszItem = DdeCreateStringHandle(idInst,
                                                  pop[i].szState, 0);

                         DdePostAdvise(idInst, hszTopic, hszItem);
                         DdeFreeStringHandle(idInst, hszItem);

                         pop[i].lPopLast = pop[i].lPop;
                         }

               return 0;

          case WM_QUERYOPEN :
               return 0;

          case WM_DESTROY :
               DdeNameService(idInst, hszService, NULL, DNS_UNREGISTER);
               DdeFreeStringHandle(idInst, hszService);
               DdeFreeStringHandle(idInst, hszTopic);

               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

DDEPOP2.RC

```
/*---------------------------
   DDEPOP2.RC resource script
  ---------------------------*/

DdePop2 ICON ddepop.ico
```

Рис. 17.3 Программа DDEPOP2

SHOWPOP2.MAK

```
#------------------------
# SHOWPOP2.MAK make file
#------------------------

showpop2.exe : showpop2.obj
     $(LINKER) $(GUIFLAGS) -OUT:showpop2.exe showpop2.obj $(GUILIBS)

showpop2.obj : showpop2.c showpop.h
     $(CC) $(CFLAGS) showpop2.c
```

SHOWPOP2.C

```
/*----------------------------------------
   SHOWPOP2.C -- DDEML Client using DDEPOP2
                 (c) Charles Petzold, 1996
  ----------------------------------------*/

#include <windows.h>
#include <ddeml.h>
#include <stdlib.h>
#include <string.h>
#include "showpop.h"

#define WM_USER_INITIATE(WM_USER + 1)
#define DDE_TIMEOUT      3000

LRESULT  CALLBACK WndProc    (HWND, UINT, WPARAM, LPARAM);
HDDEDATA CALLBACK DdeCallback(UINT, UINT, HCONV, HSZ, HSZ,
                              HDDEDATA, DWORD, DWORD);

char  szAppName[] = "ShowPop2";
DWORD idInst;
HCONV hConv;
HWND  hwnd;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
     {
     MSG        msg;
     WNDCLASSEX wndclass;

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

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "DDEML Client - US Population",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);
```

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

               // Initialize for using DDEML

     if(DdeInitialize(&idInst,(PFNCALLBACK) &DdeCallback,
                         APPCLASS_STANDARD | APPCMD_CLIENTONLY, 0L))
          {
          MessageBox(hwnd, "Could not initialize client!",
                      szAppName, MB_ICONEXCLAMATION | MB_OK);

          DestroyWindow(hwnd);
          return FALSE;
          }

               // Start things going

     SendMessage(hwnd, WM_USER_INITIATE, 0, 0L);

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

               // Uninitialize DDEML

     DdeUninitialize(idInst);

     return msg.wParam;
     }

HDDEDATA CALLBACK DdeCallback(UINT iType, UINT iFmt, HCONV hConv,
                              HSZ hsz1, HSZ hsz2, HDDEDATA hData,
                              DWORD dwData1, DWORD dwData2)
     {
     char szItem[10], szPopulation[16];
     int  i;

     switch(iType)
          {
          case XTYP_ADVDATA :       // hsz1  = topic
                                    // hsz2  = item
                                    // hData = data

                    // Check for matching format and data item

               if(iFmt != CF_TEXT)
                    return DDE_FNOTPROCESSED;

               DdeQueryString(idInst, hsz2, szItem, sizeof(szItem), 0);

               for(i = 0; i < NUM_STATES; i++)
                    if(strcmp(szItem, pop[i].szAbb) == 0)
                         break;

               if(i >= NUM_STATES)
                    return DDE_FNOTPROCESSED;

                    // Store the data and invalidate the window

               DdeGetData(hData,(unsigned char *) szPopulation,
                           sizeof(szPopulation), 0);

               pop[i].lPop = atol(szPopulation);
```

```
               InvalidateRect(hwnd, NULL, FALSE);

               return(HDDEDATA) DDE_FACK;

          case XTYP_DISCONNECT :
               hConv = NULL;

               MessageBox(hwnd, "The server has disconnected.",
                          szAppName, MB_ICONASTERISK | MB_OK);

               return NULL;
          }

     return NULL;
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char  szService[] = "DdePop2",
                  szTopic[]   = "US_Population";
     static long  cxChar, cyChar;
     char         szBuffer[24];
     HDC          hdc;
     HSZ          hszService, hszTopic, hszItem;
     PAINTSTRUCT  ps;
     int          i;
     long         x, y;
     TEXTMETRIC   tm;

     switch(iMsg)
          {
          case WM_CREATE :
               hdc = GetDC(hwnd);
               GetTextMetrics(hdc, &tm);
               cxChar = tm.tmAveCharWidth;
               cyChar = tm.tmHeight + tm.tmExternalLeading;
               ReleaseDC(hwnd, hdc);

               return 0;

          case WM_USER_INITIATE :
                         // Try connecting

               hszService = DdeCreateStringHandle(idInst, szService, 0);
               hszTopic   = DdeCreateStringHandle(idInst, szTopic,   0);

               hConv = DdeConnect(idInst, hszService, hszTopic, NULL);

                         // If that doesn't work, load server

               if(hConv == NULL)
                    {
                    WinExec(szService, SW_SHOWMINNOACTIVE);

                    hConv = DdeConnect(idInst, hszService, hszTopic, NULL);
                    }

                         // Free the string handles

               DdeFreeStringHandle(idInst, hszService);
               DdeFreeStringHandle(idInst, hszTopic);

                         // If still not connected, display message box
```

```
          if(hConv == NULL)
               {
               MessageBox(hwnd, "Cannot connect with DDEPOP2.EXE!",
                          szAppName, MB_ICONEXCLAMATION | MB_OK);

               return 0;
               }

                    // Request notification

          for(i = 0; i < NUM_STATES; i++)
               {
               hszItem = DdeCreateStringHandle(idInst, pop[i].szAbb, 0);

               DdeClientTransaction(NULL, 0, hConv, hszItem, CF_TEXT,
                                    XTYP_ADVSTART | XTYPF_ACKREQ,
                                    DDE_TIMEOUT, NULL);

               DdeFreeStringHandle(idInst, hszItem);
               }

          if(i < NUM_STATES)
               {
               MessageBox(hwnd, "Failure on WM_DDE_ADVISE!",
                          szAppName, MB_ICONEXCLAMATION | MB_OK);
               }

          return 0;
     case WM_PAINT :
          hdc = BeginPaint(hwnd, &ps);

          for(i = 0; i < NUM_STATES; i++)
               {
               if(i <(NUM_STATES + 1) / 2)
                    {
                    x = cxChar;
                    y = i * cyChar;
                    }
               else
                    {
                    x = 44 * cxChar;
                    y =(i -(NUM_STATES + 1) / 2) * cyChar;
                    }

               TextOut(hdc, x, y, szBuffer,
                         wsprintf(szBuffer, "%-20s",
                                  (PSTR) pop[i].szState));

               x += 36 * cxChar;

               SetTextAlign(hdc, TA_RIGHT | TA_TOP);

               TextOut(hdc, x, y, szBuffer,
                         wsprintf(szBuffer, "%10ld", pop[i].lPop));

               SetTextAlign(hdc, TA_LEFT | TA_TOP);
               }

          EndPaint(hwnd, &ps);
          return 0;

     case WM_CLOSE :
          if(hConv == NULL)
               break;
```

```
                    // Stop the advises

          for(i = 0; i < NUM_STATES; i++)
               {
               hszItem = DdeCreateStringHandle(idInst, pop[i].szAbb, 0);

               DdeClientTransaction(NULL, 0, hConv, hszItem, CF_TEXT,
                                    XTYP_ADVSTOP, DDE_TIMEOUT, NULL);

               DdeFreeStringHandle(idInst, hszItem);
               }

                    // Disconnect the conversation
          DdeDisconnect(hConv);

          break;             // for default processing

     case WM_DESTROY :
          PostQuitMessage(0);
          return 0;
     }
return DefWindowProc(hwnd, iMsg, wParam, lParam);
}
```

Рис. 17.4 Программа SHOWPOP2

В *WinMain* программа SHOWPOP2 передает адрес своей функции обратного вызова (которая названа *DdeCallback*) функции *DdeInitialize*. Описатель экземпляра DDEML хранится в глобальной переменной *idInst*, которая должна передаваться в качестве первого параметра всем остальным функциям DDEML.

Затем программа SHOWPOP2 посылает себе сообщение WM_USER_INITIATE; остальные инициализации происходят в оконной процедуре *WndProc*. Программа вызывает функцию *DdeCreateStringHandle* дважды при создании описателей строк для имени сервиса сервера ("*DdePop2* ") и для имени раздела ("US_Population"). Программа SHOWPOP2 передает эти описатели строк в качестве параметров функции *DdeConnect*, пытаясь инициировать диалог с сервером DDEPOP2. Как и в программе SHOWPOP1, если диалог начаться не может, программа вызывает функцию *WinExec*, чтобы загрузить DDEPOP2, и затем повторяет попытку. После этого описатели строк могут быть освобождены с помощью функции *DdeFreeStringHandle*.

Когда сервер DDEPOP2 запускается, он вызывает функцию *DdeInitialize* в своей функции *WinMain*, а также вызывает функцию *SetTimer*, устанавливая 5-секундный интервал для обновления данных о населении. Затем программа DDEPOP2 отправляет своей оконной процедуре асинхронное сообщение WM_USER_INITIATE. При обработке этого сообщения программа DDEPOP2 инициализирует данные о населении и создает два описателя строк для своих имен сервиса и раздела. Затем *WndProc* вызывает функцию *DdeNameService* для регистрации имени своего сервиса. Это не дает функции обратного вызова возможности получить какой бы то ни было запрос о взаимодействии без точного имени сервиса.

Когда программа SHOWPOP2 вызывает функцию *DdeConnect*, библиотека DDEML вызывает функцию обратного вызова в программе DDEPOP2 с типом транзакции XTYP_CONNECT. Параметры *hsz1* и *hsz2* функции обратного вызова являются описателями строк, обозначающими имена раздела и сервиса, которые переданы функции *DdeConnect* в качестве параметров. Функция *DdeCallback* программы DDEPOP2 использует функцию *DdeQueryString* для проверки того, что имена раздела и сервиса соответствуют строкам "US_Population" и "DdePop2". Если соответствует, то возвращаемым значением функции *DdeCallback* является TRUE; в противном случае ее возвращаемым значением будет FALSE. Описатели строк, переданные функции обратного вызова, в ней освобождать не нужно.

Когда, после обработки транзакции XTYP_CONNECT функция сервера *DdeCallback* возвращает TRUE, диалог DDE начинается. Функция *DdeConnect*, вызываемая клиентом (SHOWPOP2), возвращает описатель диалога, который имеет тип HCONV. Этот описатель также передается функции обратного вызова сервера в транзакции XTYP_CONNECT_CONFIRM. В сервере при желании можно было бы сохранить этот описатель диалога, но в программе DDEPOP2 этого не делается.

Когда мы последний раз говорили о программе SHOWPOP2, то она уже установила связь с программой DDEPOP2 с помощью функции *DdeConnect* и освободила описатели строк для имен сервиса и раздела, необходимые для этой функции. После этого программа SHOWPOP2 пытается установить горячую связь для получения данных о населении каждого из 50 штатов, округа Колумбия и Соединенных Штатов в целом. Для этого программа SHOWPOP2 вводит цикл и вызывает функцию *DdeCreateStringHandle* для всех двухсимвольных кодов штатов. Они передаются вместе с описателем диалога, типом формата папки обмена и флагом XTYP_ADVSTART в

функцию *DdeClientTransaction*. Это основная функция, которую клиент использует для получения данных от сервера. Затем каждый описатель строки освобождается с помощью вызова функции *DdeFreeStringHandle*.

Когда клиент, занятый в диалоге, вызывает функцию *DdeClientTransaction* с параметром XTYP_ADVSTART, функция обратного вызова сервера вызывается управляющей библиотекой DDE с типом транзакции XTYP_ADVSTART. Программа DDEPOP2 просто проверяет, является ли формат папки обмена — CF_TEXT и является ли имя элемента правильным двухсимвольным кодом. (Это делается в функции *GetStateNumber* программы DDEPOP2.) Если элемент существует, то тогда программа DDEPOP2 устанавливает значение населения штата в 0 и посылает своей оконной процедуре синхронное сообщение WM_TIMER.

Здесь начинаются настоящие чудеса DDEML. Сервер DDEPOP2 согласился участвовать в диалоге, но не сохранил описатель диалога. Функция обратного вызова DDEPOP2 также получила транзакции XTYP_ADVSTART для конкретных штатов, но не сохранила никакой информации о том, какой клиент в каком диалоге о каком штате затребовал информацию.

Программа DDEPOP2 не сохранила эту информацию, но ее сохранила управляющая библиотека DDE. Теперь посмотрим, что происходит, когда программа DDEPOP2 обновляет свою информацию о населении при обработке в *WndProc* сообщения WM_TIMER (и WM_TIMECHANGE). *WndProc* вызывает функцию *CalcPops* для расчета новой численности населения, а затем проверяет, население какого штата изменилось. Если население какого-либо штата изменилось, программа DDEPOP2 создает описатель строки для двухсимвольного кода штата и затем вызывает функцию *DdePostAdvise* с именами раздела и элемента.

Для каждого диалога, в котором клиент потребовал горячей связи с именами раздела и элемента, переданными функции *DdePostAdvise*, функция *DdePostAdvise* вызывает функцию обратного вызова программы DDEPOP2 с типом транзакции XTYP_ADVREQ. Параметры *hsz1* и *hsz2* функции обратного вызова соответствуют именам раздела и элемента. Все, что функции обратного вызова нужно сделать, это возвратить описатель данных, ссылающийся уже на обновленные данные. Для этого функция *DdeCallback* форматирует численность населения в строку и передает буфер строки функции *DdeCreateDataHandle*. Возвращаемым значением этой функции является описатель данных типа DDEDATA, который просто возвращается функцией обратного вызова. Описатели данных, возвращаемые из функции обратного вызова, не нужно специально освобождать.

Когда функция обратного вызова программы DDEPOP2 возвращает описатель данных в ответ на транзакцию XTYP_ADVREQ, управляющая библиотека DDE вызывает функцию обратного вызова программы SHOWPOP2 с типом транзакции XTYP_ADVDATA. Программа SHOWPOP2 использует функцию *DdeGetData* для получения данных о населении и сохраняет их. Затем SHOWPOP2 делает свое окно недействительным для обновления информации на экране.

Это, в основном, все. Когда программе SHOWPOP2 приходит время завершаться, она вызывает функцию *DdeClientTransaction* для всех штатов, используя тип транзакции XTYP_ADVSTOP. Программа DDEPOP2 получает эту транзакцию в своей функции обратного вызова. Затем программа SHOWPOP2 вызывает функцию *DdeDisconnect* и, после выхода из цикла обработки сообщений, функцию *DdeUninitialize*. Все действия, необходимые для освобождения ресурсов, которые не выполнила программа DDE до вызова этой функции, выполняются внутри функции *DdeUninitialize* автоматически.

Глава 18 Многооконный интерфейс

18

Многооконный интерфейс (Multiple Document Interface, MDI) является спецификацией для приложений, которые обрабатывают документы в Microsoft Windows. Спецификация описывает структуру окон и пользовательский интерфейс, который позволяет пользователю работать с несколькими документами внутри одного приложения (например, с документами в текстовом процессоре или с таблицами в программе электронных таблиц). Точно также, как Windows поддерживает несколько окон приложений на одном экране, приложение MDI поддерживает несколько окон документов в одной рабочей области. Первым приложением MDI для Windows была первая версия Microsoft Excel. И Microsoft Word for Windows, и Microsoft Access являются приложениями MDI.

Хотя спецификация MDI была введена, уже начиная с Windows 2, в то время писать приложения MDI было трудно, и от программиста требовалось большая очень сложная работа. Однако, начиная с Windows 3, большая часть этой работы была сделана. Windows 95 добавляет только одну новую функцию и одно новое сообщение к набору функций, структурам данных и сообщениям, которые существуют специально для упрощения создания приложений MDI.

Элементы MDI

Главное окно приложения программы MDI обычно: в нем имеется строка заголовка, меню, рамка изменения размера, значок системного меню и значки свертывания/развертывания. Рабочая область, однако, не используется непосредственно для вывода выходных данных программы. В этой рабочей области может находиться несколько дочерних окон, в каждом из которых отображается какой-то документ.

Эти дочерние окна выглядят совершенно также, как обычные окна приложений. В них имеется строка заголовка, рамка изменения размера, значок системного меню, значки свертывания/развертывания и, возможно, полосы прокрутки. Однако, ни в одном из окон документов нет меню. Меню главного окна приложения относится и к окнам документов.

В каждый конкретный момент времени только одно окно документа активно (об этом говорит выделенная подсветкой строка заголовка) и находится над всеми остальными окнами документов. Все дочерние окна документов находятся только в рабочей области главного окна приложения и никогда не выходят за ее границы.

Поначалу, MDI для Windows-программиста кажется совершенно понятным. Все, что нужно сделать — это создать для каждого документа окно WS_CHILD, делая главное окно приложения родительским окном для окна документа. Но при более близком знакомстве с приложением MDI, таким как Microsoft Excel, обнаруживаются определенные трудности, требующие сложного программирования. Например:

- Окно документа MDI может быть свернуто. Соответствующий значок выводится в нижней части рабочей области. (Как правило, в приложении MDI для главного окна приложения и для каждого типа окна документа будут использоваться разные значки.)
- Окно документа MDI может быть развернуто. В этом случае строка заголовка окна документа (которая обычно используется для вывода в окне имени файла документа) исчезает, и имя файла оказывается присоединенным к имени приложения в строке заголовка окна приложения. Значок системного меню окна документа становится первым пунктом строки основного меню окна приложения. Значок для восстановления размера окна документа становится последним пунктом строки основного меню и оказывается крайним справа.
- Системные быстрые клавиши для закрытия окна документа те же, что и для закрытия главного окна, за исключением того, что клавиша <Ctrl> используется вместо клавиши <Alt>. Таким образом, комбинация <Alt>+<F4> закрывает окно приложения, а комбинация <Ctrl>+<F4> закрывает окно документа. Вдобавок к

остальным быстрым клавишам, комбинация <Ctrl>+<F6> позволяет переключаться между дочерними окнами документов активного приложения MDI. Комбинация <Alt>+<Spacebar>, как обычно, вызывает системное меню главного окна. Комбинация <Alt>+<-> (минус) вызывает системное меню активного дочернего окна документа.

- При использовании клавиш управления курсором для перемещения по пунктам меню, обычно происходит переход от системного меню к первому пункту строки меню. В приложении MDI порядок перехода изменяется: от системного меню приложения — к системному меню активного документа — и далее к первому пункту строки меню.
- Если приложение имеет возможность поддерживать несколько типов дочерних окон (например, электронные таблицы и диаграммы в Microsoft Excel), то меню должно отражать операции, ассоциированные с каждым типом документа. Для этого требуется, чтобы программа изменяла меню программы, когда становится активным окно документа другого типа. Кроме этого, при отсутствии окна документа, в меню должны быть представлены только операции, связанные с открытием нового документа.
- В строке основного меню имеется пункт Window. По соглашению, он является последним пунктом строки основного меню, исключая Help. В подменю Window обычно имеются опции для упорядочивания окон документов внутри рабочей области. Окна документов можно расположить "каскадно" (cascaded), начиная от верхнего левого угла, или "мозаично" (tiled) так, что окно каждого документа будет полностью видимо. Кроме того, в этом подменю имеется перечень всех окон документов. При выборе одного из окон документов, оно выходит на передний план.

Все эти аспекты MDI поддерживаются в Windows 95. Конечно, кое-какая работа все-таки необходима (как будет показано в примере программы), но эта работа не идет ни в какое сравнение с ситуацией, когда приходится самому реализовывать поддержку всех этих возможностей.

Windows 95 и MDI

При знакомстве с поддержкой MDI в Windows 95 требуется кое-какая новая терминология. Окно приложения в целом называется главным окном (frame window). Также как в традиционной программе для Windows, это окно имеет стиль WS_OVERLAPPEDWINDOW.

Приложение MDI также создает окно-администратор (client window) на основе предопределенного класса окна MDICLIENT. Окно-администратор создается с помощью вызова функции *CreateWindow* с использованием этого класса окна и стиля WS_CHILD. Последним параметром функции *CreateWindow* является указатель на небольшую структуру типа CLIENTCREATESTRUCT. Это окно-администратор охватывает всю рабочую область главного окна и обеспечивает основную поддержку MDI. Цветом окна-администратора является системный цвет COLOR_APPWORKSPACE.

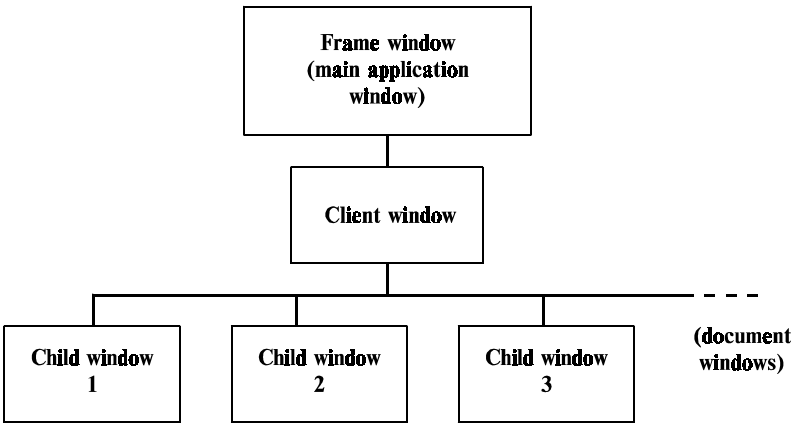


Рис. 18.1 Иерархия родительских и дочерних окон приложения MDI в Windows

Окна документов называются дочерними окнами (child windows). Эти окна создаются путем инициализации структуры типа MDICREATESTRUCT и посылки окну-администратору сообщения WM_MDICREATE с указателем на эту структуру.

Окна документов являются дочерними окнами окна-администратора, которое, в свою очередь, является дочерним окном главного окна. Эта иерархия показана на рис. 18.1.

Для главного окна и для каждого типа дочерних окон, которые поддерживаются в приложении, необходим класс окна (и оконная процедура). Для окна-администратора оконная процедура не нужна, поскольку ее класс окна предварительно зарегистрирован в системе.

Для поддержки MDI в Windows 95 имеется один класс окна, пять функций, две структуры данных и двенадцать сообщений. О классе окна MDICLIENT и структурах данных CLIENTCREATESTRUCT и MDICREATESTRUCT уже упоминалось. Две из пяти функций заменяют в приложениях MDI функцию *DefWindowProc*: вместо вызова функции *DefWindowProc* для всех необрабатываемых сообщений, оконная процедура главного окна вызывает функцию *DefFrameProc*, а оконная процедура дочернего окна вызывает функцию *DefMDIChildProc*. Другая характерная функция MDI *TranslateMDISysAccel* используется также, как функция *TranslateAccelerator*, о которой рассказывалось в главе 10. В Windows 3 добавлена функция *ArrangeIconicWindows*, но одно из специальных сообщений MDI делает ее использование не нужным в программах MDI.

Если в дочерних окнах MDI выполняются какие-то протяженные во времени операции, рассмотрите возможность их запуска в отдельных потоках. Это позволит пользователю покинуть "задумавшееся" дочернее окно и продолжить работу в другом окне, пока первое дочернее окно решает свою задачу в фоновом режиме. В Windows 95 специально для этой цели имеется новая функция *CreateMDIWindow*. Поток вызывает функцию *CreateMDIWindow* для создания дочернего окна MDI; таким образом окно действует исключительно внутри контекста потока. В программе, имеющей один поток, для создания дочернего окна функция *CreateMDIWindow* не требуется, поскольку то же самое выполняет сообщение WM_MDICREATE.

Подошло время для примера однопотоковой программы, в которой будет показано девять из двенадцати сообщений MDI. (Оставшиеся три обычно не требуются.) Эти сообщения имеют префикс WM_MDI. Главное окно посылает одно из этих сообщений окну-администратору для выполнения какой-либо операции над дочерним окном или для получения информации о дочернем окне. (Например, главное окно посылает сообщение WM_MDICREATE окну-администратору для создания дочернего окна.) Исключение составляет сообщение WM_MDIACTIVATE: в то время, как главное окно может послать это сообщение окну-администратору для активизации одного из дочерних окон, окно-администратор также посылает сообщение тем дочерним окнам, которые будут активизированы и тем, которые потеряют активность, чтобы проинформировать их о предстоящем изменении.

Пример программы

Программа MDIDEMO, представленная на рис. 18.2, иллюстрирует основы написания приложения MDI.

MDIDEMO.MAK

```
#------------------------
# MDIDEMO.MAK make file
#------------------------

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

mdidemo.obj : mdidemo.c mdidemo.h
     $(CC) $(CFLAGS) mdidemo.c

mdidemo.res : mdidemo.rc mdidemo.h
     $(RC) $(RCVARS) mdidemo.rc
```

MDIDEMO.C

```
/*--------------------------------------------------------
   MDIDEMO.C -- Multiple Document Interface Demonstration
                (c) Charles Petzold, 1996
  --------------------------------------------------------*/

#include <windows.h>
#include <stdlib.h>
#include "mdidemo.h"

LRESULT CALLBACK FrameWndProc (HWND, UINT, WPARAM, LPARAM);
BOOL    CALLBACK CloseEnumProc(HWND, LPARAM);
LRESULT CALLBACK HelloWndProc (HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK RectWndProc  (HWND, UINT, WPARAM, LPARAM);

          // structure for storing data unique to each Hello child window

typedef struct tagHELLODATA
     {
     UINT     iColor;
```

```
     COLORREF clrText;
     }
     HELLODATA, *LPHELLODATA;

          // structure for storing data unique to each Rect child window

typedef struct tagRECTDATA
     {
     short cxClient;
     short cyClient;
     }
     RECTDATA, *LPRECTDATA;

          // global variables

char      szFrameClass[] = "MdiFrame";
char      szHelloClass[] = "MdiHelloChild";
char      szRectClass[]  = "MdiRectChild";
HINSTANCE hInst;
HMENU     hMenuInit, hMenuHello, hMenuRect;
HMENU     hMenuInitWindow, hMenuHelloWindow, hMenuRectWindow;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
     {
     HACCEL      hAccel;
     HWND        hwndFrame, hwndClient;
     MSG         msg;
     WNDCLASSEX  wndclass;

     hInst = hInstance;

     if(!hPrevInstance)
          {
                    // Register the frame window class

          wndclass.cbSize        = sizeof(wndclass);
          wndclass.style         = CS_HREDRAW | CS_VREDRAW;
          wndclass.lpfnWndProc   = FrameWndProc;
          wndclass.cbClsExtra    = 0;
          wndclass.cbWndExtra    = 0;
          wndclass.hInstance     = hInstance;
          wndclass.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
          wndclass.hCursor       = LoadCursor(NULL, IDC_ARROW);
          wndclass.hbrBackground =(HBRUSH)(COLOR_APPWORKSPACE + 1);
          wndclass.lpszMenuName  = NULL;
          wndclass.lpszClassName = szFrameClass;
          wndclass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

          RegisterClassEx(&wndclass);

                    // Register the Hello child window class

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

```
     RegisterClassEx(&wndclass);

               // Register the Rect child window class

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

     RegisterClassEx(&wndclass);
     }
          // Obtain handles to three possible menus & submenus

hMenuInit  = LoadMenu(hInst, "MdiMenuInit");
hMenuHello = LoadMenu(hInst, "MdiMenuHello");
hMenuRect  = LoadMenu(hInst, "MdiMenuRect");

hMenuInitWindow  = GetSubMenu(hMenuInit,   INIT_MENU_POS);
hMenuHelloWindow = GetSubMenu(hMenuHello, HELLO_MENU_POS);
hMenuRectWindow  = GetSubMenu(hMenuRect,   RECT_MENU_POS);

          // Load accelerator table

hAccel = LoadAccelerators(hInst, "MdiAccel");

          // Create the frame window

hwndFrame = CreateWindow(szFrameClass, "MDI Demonstration",
                         WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, hMenuInit, hInstance, NULL);

hwndClient = GetWindow(hwndFrame, GW_CHILD);
ShowWindow(hwndFrame, iCmdShow);
UpdateWindow(hwndFrame);

          // Enter the modified message loop

while(GetMessage(&msg, NULL, 0, 0))
     {
     if(!TranslateMDISysAccel(hwndClient, &msg) &&
         !TranslateAccelerator(hwndFrame, hAccel, &msg))
          {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
          }
     }
          // Clean up by deleting unattached menus

DestroyMenu(hMenuHello);
DestroyMenu(hMenuRect);

return msg.wParam;
}
```

```
LRESULT CALLBACK FrameWndProc(HWND hwnd, UINT iMsg, WPARAM wParam,
                                                LPARAM lParam)
     {
     static HWND         hwndClient;
     CLIENTCREATESTRUCT  clientcreate;
     HWND                hwndChild;
     MDICREATESTRUCT     mdicreate;

     switch(iMsg)
          {
          case WM_CREATE :          // Create the client window

               clientcreate.hWindowMenu  = hMenuInitWindow;
               clientcreate.idFirstChild = IDM_FIRSTCHILD;

               hwndClient = CreateWindow("MDICLIENT", NULL,
                              WS_CHILD | WS_CLIPCHILDREN | WS_VISIBLE,
                              0, 0, 0, 0, hwnd,(HMENU) 1, hInst,
                             (LPSTR) &clientcreate);
               return 0;

          case WM_COMMAND :
               switch(wParam)
                    {
                    case IDM_NEWHELLO :       // Create a Hello child window

                         mdicreate.szClass = szHelloClass;
                         mdicreate.szTitle = "Hello";
                         mdicreate.hOwner  = hInst;
                         mdicreate.x       = CW_USEDEFAULT;
                         mdicreate.y       = CW_USEDEFAULT;
                         mdicreate.cx      = CW_USEDEFAULT;
                         mdicreate.cy      = CW_USEDEFAULT;
                         mdicreate.style   = 0;
                         mdicreate.lParam  = 0;

                         hwndChild =(HWND) SendMessage(hwndClient,
                                        WM_MDICREATE, 0,
                                       (LPARAM)(LPMDICREATESTRUCT) &mdicreate);
                         return 0;

                    case IDM_NEWRECT :        // Create a Rect child window

                         mdicreate.szClass = szRectClass;
                         mdicreate.szTitle = "Rectangles";
                         mdicreate.hOwner  = hInst;
                         mdicreate.x       = CW_USEDEFAULT;
                         mdicreate.y       = CW_USEDEFAULT;
                         mdicreate.cx      = CW_USEDEFAULT;
                         mdicreate.cy      = CW_USEDEFAULT;
                         mdicreate.style   = 0;
                         mdicreate.lParam  = 0;

                         hwndChild =(HWND) SendMessage(hwndClient,
                                        WM_MDICREATE, 0,
                                       (LPARAM)(LPMDICREATESTRUCT) &mdicreate);
                         return 0;

                    case IDM_CLOSE :          // Close the active window

                         hwndChild =(HWND) SendMessage(hwndClient,
                                                  WM_MDIGETACTIVE, 0, 0);

                         if(SendMessage(hwndChild, WM_QUERYENDSESSION, 0, 0))
                              SendMessage(hwndClient, WM_MDIDESTROY,
```

```
                                        (WPARAM) hwndChild, 0);
                    return 0;

               case IDM_EXIT :           // Exit the program

                    SendMessage(hwnd, WM_CLOSE, 0, 0);
                    return 0;

                              // messages for arranging windows
               case IDM_TILE :
                    SendMessage(hwndClient, WM_MDITILE, 0, 0);
                    return 0;

               case IDM_CASCADE :
                    SendMessage(hwndClient, WM_MDICASCADE, 0, 0);
                    return 0;

               case IDM_ARRANGE :
                    SendMessage(hwndClient, WM_MDIICONARRANGE, 0, 0);
                    return 0;

               case IDM_CLOSEALL :       // Attempt to close all children

                    EnumChildWindows(hwndClient, &CloseEnumProc, 0);
                    return 0;

               default :            // Pass to active child...

                    hwndChild =(HWND) SendMessage(hwndClient,
                                             WM_MDIGETACTIVE, 0, 0);

                   if(IsWindow(hwndChild))
                        SendMessage(hwndChild, WM_COMMAND,
                                     wParam, lParam);

                   break;        // ...and then to DefFrameProc
               }
          break;

     case WM_QUERYENDSESSION :
     case WM_CLOSE :                     // Attempt to close all children

          SendMessage(hwnd, WM_COMMAND, IDM_CLOSEALL, 0);

          if(NULL != GetWindow(hwndClient, GW_CHILD))
               return 0;

          break;   // I.e., call DefFrameProc

     case WM_DESTROY :
          PostQuitMessage(0);
          return 0;
     }
          // Pass unprocessed messages to DefFrameProc(not DefWindowProc)

return DefFrameProc(hwnd, hwndClient, iMsg, wParam, lParam);
}

BOOL CALLBACK CloseEnumProc(HWND hwnd, LPARAM lParam)
{
if(GetWindow(hwnd, GW_OWNER))         // Check for icon title
     return 1;

SendMessage(GetParent(hwnd), WM_MDIRESTORE,(WPARAM) hwnd, 0);
if(!SendMessage(hwnd, WM_QUERYENDSESSION, 0, 0))
```

```
          return 1;

     SendMessage(GetParent(hwnd), WM_MDIDESTROY,(WPARAM) hwnd, 0);
          return 1;
     }

LRESULT CALLBACK HelloWndProc(HWND hwnd, UINT iMsg, WPARAM wParam,
                                                    LPARAM lParam)
     {
     static COLORREF clrTextArray[] = { RGB(0,   0, 0), RGB(255, 0,   0),
                                        RGB(0, 255, 0), RGB(  0, 0, 255),
                                        RGB(255, 255, 255) };
     static HWND     hwndClient, hwndFrame;
     HDC             hdc;
     HMENU           hMenu;
     LPHELLODATA     lpHelloData;
     PAINTSTRUCT     ps;
     RECT            rect;

     switch(iMsg)
          {
          case WM_CREATE :
                         // Allocate memory for window private data

               lpHelloData =(LPHELLODATA) HeapAlloc(GetProcessHeap(),
                                                    HEAP_ZERO_MEMORY,
                                                    sizeof(HELLODATA));
               lpHelloData->iColor  = IDM_BLACK;
               lpHelloData->clrText = RGB(0, 0, 0);
               SetWindowLong(hwnd, 0,(long) lpHelloData);

                         // Save some window handles

               hwndClient = GetParent(hwnd);
               hwndFrame  = GetParent(hwndClient);
               return 0;

          case WM_COMMAND :
               switch(wParam)
                    {
                    case IDM_BLACK :
                    case IDM_RED :
                    case IDM_GREEN :
                    case IDM_BLUE :
                    case IDM_WHITE :
                                   // Change the text color

                         lpHelloData =(LPHELLODATA) GetWindowLong(hwnd, 0);

                         hMenu = GetMenu(hwndFrame);
                         CheckMenuItem(hMenu, lpHelloData->iColor,
                                              MF_UNCHECKED);
                         lpHelloData->iColor = wParam;
                         CheckMenuItem(hMenu, lpHelloData->iColor,
                                              MF_CHECKED);

                         lpHelloData->clrText =
                               clrTextArray[wParam - IDM_BLACK];

                         InvalidateRect(hwnd, NULL, FALSE);
                    }
               return 0;

          case WM_PAINT :
                         // Paint the window
```

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

               lpHelloData =(LPHELLODATA) GetWindowLong(hwnd, 0);
               SetTextColor(hdc, lpHelloData->clrText);

               GetClientRect(hwnd, &rect);

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

               EndPaint(hwnd, &ps);
               return 0;

          case WM_MDIACTIVATE :

                         // Set the Hello menu if gaining focus

               if(lParam ==(LPARAM) hwnd)
                    SendMessage(hwndClient, WM_MDISETMENU,
                              (WPARAM) hMenuHello,(LPARAM) hMenuHelloWindow);

                         // Check or uncheck menu item

               lpHelloData =(LPHELLODATA) GetWindowLong(hwnd, 0);
               CheckMenuItem(hMenuHello, lpHelloData->iColor,
                       (lParam ==(LPARAM) hwnd) ? MF_CHECKED : MF_UNCHECKED);

                         // Set the Init menu if losing focus

               if(lParam !=(LPARAM) hwnd)
                    SendMessage(hwndClient, WM_MDISETMENU,(WPARAM) hMenuInit,
                                (LPARAM) hMenuInitWindow);

               DrawMenuBar(hwndFrame);
               return 0;
          case WM_QUERYENDSESSION :
          case WM_CLOSE :
               if(IDOK != MessageBox(hwnd, "OK to close window?", "Hello",
                                     MB_ICONQUESTION | MB_OKCANCEL))
                    return 0;

               break;   // I.e., call DefMDIChildProc

          case WM_DESTROY :
               lpHelloData =(LPHELLODATA) GetWindowLong(hwnd, 0);
               HeapFree(GetProcessHeap(), 0, lpHelloData);
               return 0;
          }
               // Pass unprocessed message to DefMDIChildProc

     return DefMDIChildProc(hwnd, iMsg, wParam, lParam);
     }

LRESULT CALLBACK RectWndProc(HWND hwnd, UINT iMsg, WPARAM wParam,
                                                        LPARAM lParam)
     {
     static HWND  hwndClient, hwndFrame;
     HBRUSH       hBrush;
     HDC          hdc;
     LPRECTDATA   lpRectData;
     PAINTSTRUCT  ps;
     int          xLeft, xRight, yTop, yBottom;
     short        nRed, nGreen, nBlue;
```

```
switch(iMsg)
     {
     case WM_CREATE :
                    // Allocate memory for window private data

          lpRectData =(LPRECTDATA) HeapAlloc(GetProcessHeap(),
                                             HEAP_ZERO_MEMORY,
                                             sizeof(RECTDATA));

          SetWindowLong(hwnd, 0,(long) lpRectData);

                    // Start the timer going

          SetTimer(hwnd, 1, 250, NULL);

                    // Save some window handles

          hwndClient = GetParent(hwnd);
          hwndFrame  = GetParent(hwndClient);
          return 0;

     case WM_SIZE :            // If not minimized, save the window size
          if(wParam != SIZE_MINIMIZED)
               {
               lpRectData =(LPRECTDATA) GetWindowLong(hwnd, 0);

               lpRectData->cxClient = LOWORD(lParam);
               lpRectData->cyClient = HIWORD(lParam);
               }

          break;        // WM_SIZE must be processed by DefMDIChildProc

     case WM_TIMER :           // Display a random rectangle

          lpRectData =(LPRECTDATA) GetWindowLong(hwnd, 0);

          xLeft   = rand() % lpRectData->cxClient;
          xRight  = rand() % lpRectData->cxClient;
          yTop    = rand() % lpRectData->cyClient;
          yBottom = rand() % lpRectData->cyClient;
          nRed    = rand() & 255;
          nGreen  = rand() & 255;
          nBlue   = rand() & 255;

          hdc = GetDC(hwnd);
          hBrush = CreateSolidBrush(RGB(nRed, nGreen, nBlue));
          SelectObject(hdc, hBrush);

          Rectangle(hdc, min(xLeft, xRight), min(yTop, yBottom),
                         max(xLeft, xRight), max(yTop, yBottom));

          ReleaseDC(hwnd, hdc);
          DeleteObject(hBrush);
          return 0;

     case WM_PAINT :           // Clear the window

          InvalidateRect(hwnd, NULL, TRUE);
          hdc = BeginPaint(hwnd, &ps);
          EndPaint(hwnd, &ps);
          return 0;

     case WM_MDIACTIVATE :     // Set the appropriate menu
          if(lParam ==(LPARAM) hwnd)
               SendMessage(hwndClient, WM_MDISETMENU,(WPARAM) hMenuRect,
```

```
                              (LPARAM) hMenuRectWindow);
               else
                    SendMessage(hwndClient, WM_MDISETMENU,(WPARAM) hMenuInit,
                              (LPARAM) hMenuInitWindow);

               DrawMenuBar(hwndFrame);
               return 0;
          case WM_DESTROY :
               lpRectData =(LPRECTDATA) GetWindowLong(hwnd, 0);
               HeapFree(GetProcessHeap(), 0, lpRectData);
               KillTimer(hwnd, 1);
               return 0;
          }
               // Pass unprocessed message to DefMDIChildProc

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

MDIDEMO.RC

```
/*----------------------------
   MDIDEMO.RC resource script
  ----------------------------*/

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

MdiMenuInit MENU
     {
     POPUP "&File"
          {
          MENUITEM "New &Hello",             IDM_NEWHELLO
          MENUITEM "New &Rectangles",        IDM_NEWRECT
          MENUITEM SEPARATOR
          MENUITEM "E&xit",                  IDM_EXIT
          }
     }

MdiMenuHello MENU
     {
     POPUP "&File"
          {
          MENUITEM "New &Hello",             IDM_NEWHELLO
          MENUITEM "New &Rectangles",        IDM_NEWRECT
          MENUITEM "&Close",                 IDM_CLOSE
          MENUITEM SEPARATOR
          MENUITEM "E&xit",                  IDM_EXIT
          }
     POPUP "&Color"
          {
          MENUITEM "&Black",                 IDM_BLACK
          MENUITEM "&Red",                   IDM_RED
          MENUITEM "&Green",                 IDM_GREEN
          MENUITEM "B&lue",                  IDM_BLUE
          MENUITEM "&White",                 IDM_WHITE
          }
     POPUP "&Window"
          {
          MENUITEM "&Cascade\tShift+F5",     IDM_CASCADE
          MENUITEM "&Tile\tShift+F4",        IDM_TILE
          MENUITEM "Arrange &Icons",         IDM_ARRANGE
          MENUITEM "Close &All",             IDM_CLOSEALL
          }
     }

MdiMenuRect MENU
```

```
     {
     POPUP "&File"
          {
          MENUITEM "New &Hello",             IDM_NEWHELLO
          MENUITEM "New &Rectangles",        IDM_NEWRECT
          MENUITEM "&Close",                 IDM_CLOSE
          MENUITEM SEPARATOR
          MENUITEM "E&xit",                  IDM_EXIT
          }
     POPUP "&Window"
          {
          MENUITEM "&Cascade\tShift+F5",     IDM_CASCADE
          MENUITEM "&Tile\tShift+F4",        IDM_TILE
          MENUITEM "Arrange &Icons",         IDM_ARRANGE
          MENUITEM "Close &All",             IDM_CLOSEALL
          }
     }

MdiAccel ACCELERATORS
     {
     VK_F5, IDM_CASCADE, VIRTKEY, SHIFT
     VK_F4, IDM_TILE,    VIRTKEY, SHIFT
     }
```

MDIDEMO.H

```
/*-----------------------
   MDIDEMO.H header file
  -----------------------*/

#define EXPORT            __declspec(dllexport)

#define INIT_MENU_POS     0
#define HELLO_MENU_POS    2
#define RECT_MENU_POS     1

#define IDM_NEWHELLO     10
#define IDM_NEWRECT      11
#define IDM_CLOSE        12
#define IDM_EXIT         13

#define IDM_BLACK        20
#define IDM_RED          21
#define IDM_GREEN        22
#define IDM_BLUE         23
#define IDM_WHITE        24

#define IDM_TILE         30
#define IDM_CASCADE      31
#define IDM_ARRANGE      32
#define IDM_CLOSEALL     33

#define IDM_FIRSTCHILD  100
```

Рис. 18.2 Текст программы MDIDEMO

Программа MDIDEMO поддерживает два типа предельно простых окон документов: в одном в центре рабочей области выводится фраза "Hello, World!", а во втором отображается последовательность случайных прямоугольников. (В листингах исходных текстов программы и именах идентификаторов они упоминаются как документ Hello и документ Rect.) С каждым из этих двух типов окон документов связано свое меню. Окно документа, в котором выводятся слова "Hello, World!", имеется меню, дающее возможность изменять цвет символов.

Три меню

Начнем анализ программы с файла описания ресурсов MDIDEMO.RC. В нем определяются три шаблона меню, используемые в программе.

Программа выводит на экран меню *MdiMenuInit* при отсутствии окон документов. Это меню просто позволяет создать новый документ или завершить программу.

Меню *MdiMenuHello* связано с окном документа, в котором выводится фраза "Hello, World!". Подменю File позволяет открыть новый документ любого типа, закрыть активный документ и завершить программу. Подменю Color позволяет задать цвет символов. В подменю Window имеются опции для упорядочивания окон документов в каскадном или мозаичном виде, упорядочивания значков документов и закрытия всех окон. В этом подменю также имеется список всех открытых окон документов.

Меню *MdiMenuRect* связано с документом со случайными прямоугольниками. Оно аналогично меню *MdiMenuHello*, за исключением того, что в нем нет подменю Color.

В заголовочном файле MDIDEMO.H все идентификаторы меню определяются как три константы:

```
#define INIT_MENU_POS              0
#define HELLO_MENU_POS             2
#define RECT_MENU_POS              1
```

Эти идентификаторы задают положение подменю Window в каждом из трех шаблонов меню. Эта информация необходима программе, чтобы информировать окно-администратор о том, когда должен появиться список документов. Конечно, в меню *MdiMenuInit* нет подменю Window, поэтому в файле обозначено, что список должен быть присоединен к первому подменю (положение 0). Однако, фактически здесь список никогда не появится. (Зачем он нужен станет ясно при дальнейшем анализе программы.)

Идентификатор IDM_FIRSTCHILD не соответствует ни одному пункту меню. Этот идентификатор будет связан с первым окном документа в списке, который появляется в подменю Window. Значение этого идентификатора должно быть больше, чем значение всех остальных идентификаторов меню.

Инициализация программы

В файле MDIDEMO.C *WinMain* начинается с регистрации классов окна для главного окна и двух дочерних окон. Их оконные процедуры называются *FrameWndProc*, *HelloWndProc* и *RectWndProc*. Как правило, с каждым из этих трех классов связан свой значок. Для простоты в программе и для главного и для дочерних окон использован стандартный значок IDI_APPLICATION.

Обратите внимание, что в поле *hbrBackground* структуры WNDCLASSEX для класса главного окна задан системный цвет COLOR_APPWORKSPACE. Это необязательно, поскольку рабочая область главного окна полностью занята окном-администратором, а окно-администратор имеет тот же самый цвет. Однако, когда главное окно появляется на экране первым, одинаковый цвет смотрится несколько лучше.

В поле *lpszMenuName* заносится значение NULL для каждого из трех классов окна. Для классов окна дочерних окон Hello и Rect это нормально. Для главного окна это сделано для того чтобы указать описатель меню при создании главного окна в функции *CreateWindow*.

Классы окна дочерних окон Hello и Rect для каждого окна резервируют дополнительную область памяти путем указания ненулевого значения в поле *cbWndExtra* структуры WNDCLASSEX. Эта область будет использоваться для хранения указателя на блок памяти (размером со структуру HELLODATA или RECTDATA, которые определены в начале файла MDIDEMO.C), необходимый для хранения информации, уникальной для каждого окна документа.

Далее *WinMain* использует функцию *LoadMenu* для загрузки трех меню и сохраняет их описатели в глобальных переменных. Три вызова функции *GetSubMenu* позволяют получить описатели подменю Window, в которому будет добавлен список окон документов. Они также запоминаются в глобальных переменных. Функция *LoadAccelerators* загружает таблицу быстрых клавиш.

Вызов функции *CreateWindow* в *WinMain* создает главное окно. При обработке в *FrameWndProc* сообщения WM_CREATE главное окно создает окно-администратор. При этом еще раз вызывается функция *CreateWindow*. Класс окна задается как MDICLIENT, который представляет собой уже зарегистрированный в системе класс для окна-администратора MDI. Последний параметр функции *CreateWindow* должен быть указателем на структуру типа CLIENTCREATESTRUCT. В этой структуре имеется два поля:

- *hWindowMenu* является описателем подменю, в котором появится список документов. В программе MDIDEMO это описатель *hMenuInitWindow*, полученный в *WinMain*. Позже мы узнаем, как изменить меню.

- *idFirstChild* является идентификатором меню, относящимся к первому окну документа в списке документов. Он просто равен IDM_FIRSTCHILD.

Вернемся к *WinMain*. Программа MDIDEMO выводит на экран только что созданное главное окно и входит в цикл обработки сообщений. Этот цикл обработки сообщений немного отличается от обычного: после получения сообщения из очереди при помощи функции *GetMessage* программа MDI передает сообщение функции *TranslateMDISysAccel* (и функции *TranslateAccelerator*, если, как и в программе MDIDEMO, в программе также имеются быстрые клавиши меню).

Функция *TranslateMDISysAccel* преобразует любые комбинации клавиш, которые могут соответствовать специальным быстрым клавишам MDI (например, <Ctrl>+<F6>), в сообщение WM_SYSCOMMAND. Если одна из функций *TranslateMDISysAccel* или *TranslateAccelerator* возвращает TRUE (что означает, что сообщение было преобразовано одной из этих функций), то вызова функции *TranslateMessage* и функции *DispatchMessage* не происходит.

Обратите внимание на два описателя окон, передаваемые функциям *TranslateMDISysAccel* и *TranslateAccelerator*: соответственно *hwndClient* и *hwndFrame*. Функция *WinMain* получает описатель окна *hwndClient*, используя вызов функции *GetWindow* с параметром GW_CHILD.

Создание дочерних окон

Часть *FrameWndProc* связана с обработкой сообщений WM_COMMAND, которые информируют о выборе какого-либо пункта меню. Как обычно, параметр *wParam* сообщения в *FrameWndProc* содержит идентификатор меню.

При значениях параметра *wParam* IDM_NEWHELLO и IDM_NEWRECT, *FrameWndProc* должна создать новое окно документа. Это требует инициализации полей структуры MDICREATESTRUCT (большая часть которых соответствует параметрам функции *CreateWindow*) и отправки окну-администратору сообщения WM_MDICREATE с параметром *lParam*, равным указателю на эту структуру. Затем окно-администратор создает дочернее окно документа. *FrameWndProc*, вызывая функцию *CreateMDIWindow*, могла бы сама создать это дочернее окно. Для программы, имеющей один поток, такой как MDIDEMO, можно выбрать любой из этих методов.

Как правило, поле *szTitle* структуры MDICREATESTRUCT является именем файла, соответствующего документу. В поле *style* могут быть заданы стили окна WS_HSCROLL, или WS_VSCROLL, или оба вместе, что позволяет включить в окно документа полосы прокрутки. (Функцию *ShowScrollBar* для вывода полос прокрутки на экран вызывать не обязательно.) В поле *style* могут быть также указаны стили WS_MINIMIZE или WS_MAXIMIZE для первого появления окна документа в свернутом или развернутом состоянии.

Поле *lParam* структуры MDICREATESTRUCT дает возможность главному и дочернему окну использовать некоторые общие переменные. Этому полю можно присвоить значение описателя памяти, соответствующего блоку памяти, содержащему структуру. При обработке сообщения WM_CREATE в дочернем окне документа, параметр *lParam* — это указатель на структуру CREATESTRUCT, а поле *lpCreateParams* этой структуры — это указатель на структуру MDICREATESTRUCT, используемую для создания окна.

При получении сообщения WS_MDICREATE окно-администратор создает дочернее окно документа и добавляет заголовок окна к нижней части подменю, заданного в структуре MDICREATESTRUCT, которая используется для создания окна-администратора. Когда программа MDIDEMO создает свое первое окно документа, этим подменю является подменю File меню *MdiMenuInit*. Позже мы увидим, как этот список документов переносится в подменю Window меню *MdiMenuHello* и *MdiMenuRect*.

В меню может быть перечислено до девяти документов, перед каждым из которых ставится номер от 1 до 9. Номер подчеркивается. Если создается более девяти окон документов, то за этим списком появляется пункт меню "More windows". Выбор этого пункта вызывает появление окна диалога содержащего список, в котором перечислены все окна документов. Поддержка такого списка документов — это одно из самых лучших характеристик поддержки MDI в Windows 95.

Дополнительная информация об обработке сообщений в главном окне

Перед тем, как перейти к рассмотрению дочерних окон документов, разберемся с обработкой сообщений в *FrameWndProc*.

При выборе в меню File опции Close программа MDIDEMO закрывает активное дочернее окно. Описатель активного дочернего окна она получает, посылая окну-администратору сообщение WM_MDIGETACTIVE. Если дочернее окно отвечает утвердительно на сообщение WM_QUERYENDSESSION, то программа MDIDEMO для закрытия дочернего окна посылает окну-администратору сообщение WM_MDIDESTROY.

Для обработки опции Exit меню File необходимо только, чтобы оконная процедура главного окна послала себе сообщение WM_CLOSE.

Обработать опции Tile, Cascade и Arrange Icons из подменю Window проще простого, нужно только послать окну-администратору сообщения WM_MDITILE, WM_MDICASCADE и WM_MDIICONARRANGE.

Обработка опции Close All несколько сложнее. *FrameWndProc* вызывает функцию *EnumChildWindows*, передавая указатель на функцию *CloseEnumProc*. Эта функция посылает сообщение WM_MDIRESTORE каждому дочернему окну, затем сообщение WM_QUERYENDSESSION и (возможно) сообщение WM_MDIDESTROY. Этого не делается для окна заголовка значка, определяемого, если возвращаемое значение функции *GetWindow* с параметром GW_OWNER не равно NULL.

Обратите внимание, что *FrameWndProc* не обрабатывает ни одного сообщения WM_COMMAND, которые сигнализируют о том, что в меню Color выбран один из цветов. Эти сообщения относятся сфере ответственности окна документа. По этой причине *FrameWndProc* посылает все необрабатываемые сообщения WM_COMMAND активному дочернему окну, следовательно, дочернее окно может обработать те сообщения, которые относятся к нему.

Все сообщения, которые оконная процедура главного окна не обрабатывает, должны передаваться в *DefFrameProc*. Эта функция заменяет в оконной процедуре главного окна функцию *DefWindowProc*. Даже если оконная процедура главного окна и перехватывает сообщения WM_MENUCHAR, WM_SETFOCUS или WM_SIZE, все равно они должны передаваться в *DefFrameProc*.

Необрабатываемые сообщения WM_COMMAND также должны передаваться в *DefFrameProc*. В частности, *FrameWndProc* не обрабатывает сообщений WM_COMMAND, появившихся в результате того, что пользователь выбирает один из документов из списка в подменю Window. (Значение параметра *wParam* для этих опций начинается с IDM_FIRSTCHILD.) Эти сообщения передаются в *DefFrameProc* и обрабатываются там.

Обратите внимание, что главному окну не нужно поддерживать список описателей окон всех созданных документов. При необходимости (например, при обработке опции Close All из меню), эти описатели можно получить, вызывая функцию *EnumChildWindows*.

Дочерние окна документов

Теперь рассмотрим *HelloWndProc* — оконную процедуру тех дочерних окон документов, которые выводят на экран фразу "Hello, World!".

Как и для любого класса окна, который используется более, чем для одного окна, статические переменные, определенные в оконной процедуре (как и в любой функции, вызываемой из оконной процедуры), совместно используются всеми окнами, созданными на основе этого класса окна.

Данные, уникальные для каждого окна, должны храниться в форме, отличной от статических переменных. Один из таких приемов подразумевает применение свойств окна. При другом подходе (реализованном в программе) используется область памяти, зарезервированная путем определения отличного от нуля значения в поле *cbWndExtra* структуры WNDCLASSEX, используемой при регистрации класса окна.

В программе MDIDEMO эта область предназначена для хранения указателя на блок памяти размером со структуру HELLODATA. *HelloWndProc* выделяет эту память при обработке сообщения WM_CREATE, инициализирует оба поля (обозначающие помеченный в данный момент контрольной меткой пункт меню и цвет символов) и с помощью функции *SetWindowLong* запоминает указатель на блок памяти.

При обработке сообщения WM_COMMAND, связанного с изменением цвета символов (вспомните, что это сообщение сначала появляется в оконной процедуре главного окна), *HelloWndProc* задействует функцию *GetWindowLong* для получения указателя на блок памяти, содержащий структуру HELLODATA. Используя эту структуру, *HelloWndProc* снимает пометку с помеченного ранее пункта меню, ставит ее к выбранному пункту, и запоминает новый цвет.

Оконная процедура окна документа получает сообщение WM_MDIACTIVATE всегда, когда окно становится активным или перестает быть активным (в зависимости от того, содержится ли описатель окна в параметре *lParam* этого сообщения). Вспомните, что в программе MDIDEMO имеется три различных меню: *MdiMenuInit* выводится, если нет ни одного окна документа, *MdiMenuHello* выводится, если активно окно документа Hello, и *MdiMenuRect* выводится, если активно окно документа с прямоугольниками.

Сообщение WM_MDIACTIVATE дает возможность окну документа изменить меню. Если в параметре *lParam* этого сообщения содержится описатель окна (означающий, что окно становится активным), *HelloWndProc* изменяет меню на *MdiMenuHello*. Если в параметре *lParam* этого сообщения содержится описатель другого окна, *HelloWndProc* преобразует меню в *MdiMenuInit*.

HelloWndProc изменяет меню путем отправки сообщения WM_MDISETMENU окну-администратору. Окно-администратор обрабатывает это сообщение, удаляя список документов из текущего меню и присоединяя его к новому меню. Таким образом список документов попадает из меню *MdiMenuInit* (которое является результатом

создания первого документа) в меню *MdiMenuHello*. Не используйте для изменения меню в приложении MDI функцию *SetMenu*.

Другая небольшая проблема связана с пометкой в подменю Color. Такие опции программы как эта, должны быть уникальны для каждого документа. Например, необходимо иметь возможность задать для одного окна документа черный цвет символов, а для другого — красный. Пометка в меню должна отражать выбранную в активном окне опцию. По этой причине *HelloWndProc* удаляет пометку выбранного пункта меню, когда окно перестает быть активным и ставит ее у соответствующего пункта, когда окно становится активным.

Значения параметров *wParam* и *lParam* сообщения WM_MDIACTIVATE являются, соответственно, описателями окна, которое перестает быть активным, и окна, становящегося активным. Оконная процедура получает первое сообщение WM_MDIACTIVATE с параметром *lParam* равным описателю текущего окна, когда это окно впервые создается, а когда окно закрывается, она получает последнее сообщение WM_MDIACTIVATE с параметром *lParam* равным другому значению. Когда пользователь переключается с одного документа на другой, первое окно документа получает сообщение WM_MDIACTIVATE с параметром *lParam* равным описателю первого окна (в это время оконная процедура устанавливает меню в *MdiMenuInit*). Второе окно документа получает сообщение WM_MDIACTIVATE с параметром *wParam* равным описателю второго окна (в это время оконная процедура устанавливает меню либо в *MdiMenuHello*, либо в *MdiMenuRect*, в зависимости от описателя). При закрытии всех окон документов остается только меню *MdiMenuInit*.

Вспомните, что *FrameWndProc* посылает дочернему окну асинхронное сообщение WM_QUERYENDSESSION, когда пользователь выбирает в меню опцию Close или Close All. *HelloWndProc* обрабатывает сообщения WM_QUERYENDSESSION и WM_CLOSE, выводя на экран окно сообщений с запросом пользователю о том, можно ли закрывать окно. (В реальной программе в этом окне сообщений может появляться запрос о том, нужно ли сохранять файл.) Если пользователь выбирает опцию, соответствующую тому, что окно закрывать не следует, оконная процедура возвращает 0.

При обработке сообщения WM_DESTROY, *HelloWndProc* освобождает блок памяти, выделенный при обработке сообщения WM_CREATE.

Все необрабатываемые сообщения должны передаваться в *DefMDIChildProc* (а не в *DefWindowProc*) для их обработки по умолчанию. Некоторые сообщения должны быть переданы в *DefMDIChildProc* независимо от того, обрабатываются ли они как-нибудь в оконной процедуре или нет. Такими сообщениями являются: WM_CHILDACTIVATE, WM_GETMINMAXINFO, WM_MOVE, WM_SETFOCUS, WM_SIZE, WM_MENUCHAR и WM_SYSCOMMAND.

RectWndProc почти полностью аналогична *HelloWndProc*, но она несколько проще (нет опции меню для выбора цвета, и окно может быть закрыто без запроса разрешения на закрытие у пользователя), поэтому нет смысла ее специально рассматривать. Но, тем не менее, обратите внимание, что *RectWndProc* прерывает обработку сообщения WM_SIZE, и оно передается в *DefMDIChildProc*.

Освобождение захваченных ресурсов

Программа MDIDEMO в функции *WinMain* использует функцию *LoadMenu* для загрузки трех меню, определенных в файле описания ресурсов. Обычно Windows удаляет меню, когда закрывается окно, к которому меню относится. Это касается и меню *MdiMenuInit*. Однако, меню, не относящиеся к какому бы то ни было окну (в программе MDIDEMO это меню Hello и Rect), будут продолжать занимать некоторую область памяти, даже после завершения программы. Поэтому, для освобождения памяти, занимаемой меню Hello и Rect, в программе MDIDEMO функция *DestroyMenu* в *WinMain* вызывается дважды.

Сила оконной процедуры

Большая часть того, что в Windows 95 создано для поддержки многооконного интерфейса, заключено в классе окна MDICLIENT. В этом совершенно отчетливо проявляется сила объектно-ориентированной архитектуры Windows. Оконная процедура окна-администратора служит, как бы, связующим звеном между главным окном и различными окнами документов.

Теперь рассмотрим другое мощное средство Windows — динамически подключаемые библиотеки.

Глава 19 Динамически подключаемые библиотеки

19

Динамически подключаемые библиотеки (DLL, или динамические библиотеки, или библиотеки динамической компоновки, или модули библиотек) являются одним из наиболее важных структурных элементов Windows. Большинство файлов, из которых состоит Windows, представляют из себя либо программные модули, либо модули динамически подключаемых библиотек. До сих пор мы писали программы для Windows; теперь настала пора заняться написанием динамически подключаемых библиотек. Большая часть принципов, относящихся к написанию программ, вполне подходит и для написания этих библиотек, но есть несколько важных отличий.

Основы библиотек

Как известно, Windows-программа представляет собой исполняемый файл, который обычно создает одно или более окон, а для получения данных от пользователя использует цикл обработки сообщений. Динамически подключаемые библиотеки, как правило, непосредственно не выполняются и обычно не получают сообщений. Они представляют из себя отдельные файлы с функциями, которые вызываются программами и другими динамическими библиотеками для выполнения определенных задач. Динамически подключаемая библиотека активизируется только тогда, когда другой модуль вызывает одну из функций, находящихся в библиотеке.

Термин *динамическое связывание* (dinamic linking) относится к процессам, которые Windows использует для того, чтобы связать вызов функции в одном из модулей с реальной функцией из модуля библиотеки. *Статическое связывание* (static linking) имеет место в процессе создания программы, когда для создания исполняемого (.EXE) файла связываются воедино разные объектные (.OBJ) модули, файлы библиотек (.LIB) и, как правило, скомпилированные файлы описания ресурсов (.RES). В отличие от этого, динамическое связывание имеет место во время выполнения программы.

Файлы KERNEL32.DLL, USER32.DLL и GDI32.DLL, файлы различных драйверов, например, KEYBOARD.DRV, SYSTEM.DRV и MOUSE.DRV, драйверы мониторов и принтеров — все это динамически подключаемые библиотеки. Эти библиотеки можно использовать во всех программах Windows.

Некоторые динамически подключаемые библиотеки (например, файлы шрифтов) содержат только ресурсы (resource only). В них содержатся только данные (обычно в виде ресурсов), и нет текстов программ. Таким образом, одной из целью существования динамически подключаемых библиотек должно быть обеспечение функциями и ресурсами, которые можно использовать во многих, совершенно разных программах. В традиционной операционной системе, только сама операционная система содержит программы, которые для решения каких-то задач могут вызывать другие программы. В Windows принцип вызова одним модулем функции из другого модуля, распространен на всю операционную систему. Это приводит к тому, что когда вы пишете динамически подключаемую библиотеку — вы пишете расширение Windows. Можете считать динамически подключаемые библиотеки (включая те, которые составляют Windows) дополнением вашей программы.

Хотя модуль динамически подключаемой библиотеки может иметь любое расширение (например, .EXE или .FON), стандартным расширением, принятым в Windows 95, является .DLL. Только те динамически подключаемые библиотеки, которые имеют расширение .DLL, Windows загрузит автоматически. Если файл имеет другое расширение, то программа должна загрузить модуль библиотеки явно. Для этого используется функция *LoadLibrary* или *LoadLibraryEx*.

Как правило, наибольший смысл динамически подключаемые библиотеки приобретают в контексте большого приложения. Например, предположим вы написали большой пакет приложений для расчетов под Windows, содержащий несколько разных программ. Обычно в таких программах используются некоторые общие подпрограммы. Эти подпрограммы можно поместить в обычную объектную библиотеку (с расширением .LIB) и

добавить ее к каждому программному модулю при компоновке программы с помощью компоновщика LINK. Но такой подход избыточен, поскольку в этом случае в каждой программе пакета будет находиться одинаковый код подпрограмм, выполняющих одни и те же задачи. Более того, при изменении одной из подпрограмм в библиотеке, необходимо перекомпоновать все программы, в состав которых входит измененная подпрограмма. Однако, если поместить эти общеиспользуемые подпрограммы в динамически подключаемую библиотеку (назовем ее к примеру ACCOUNT.DLL), то обе проблемы будут решены. Только в одном библиотечном модуле нужно содержать необходимые всем программам подпрограммы (таким образом, для файлов требуется меньше дискового пространства, а, при одновременном запуске двух или более приложений, меньше оперативной памяти), и можно изменить библиотечный модуль без перекомпоновки программ.

Динамически подключаемые библиотеки сами могут стать ценным продуктом. Например, предположим вы написали набор программ трехмерного рисования и поместили их в динамически подключаемую библиотеку GDI3.DLL. Если своей библиотекой вам удастся заинтересовать производителей программного обеспечения, то можно получить лицензию на нее для включения в состав других графических программ. Пользователю, у которого имеется несколько таких программ понадобится только один файл GDI3.DLL.

Библиотека: одно слово, множество значений

Путаница, связанная с использованием динамически подключаемых библиотек, является результатом появления слова "библиотека" в нескольких разных контекстах. Помимо динамически подключаемых библиотек, оно упоминается в сочетаниях "объектная библиотека" и "библиотека импорта".

Объектная библиотека — это файл с расширением .LIB, в котором содержится код, добавляемый к коду программы, находящемуся в файле с расширением .EXE, когда идет процесс компоновки программы. Например, в Microsoft Visual C++ обычная объектная библиотека времени выполнения, которая при компоновке присоединяется к вашей программе, называется LIBC.LIB.

Библиотека импорта представляет собой особую форму файла объектной библиотеки. Также как объектные библиотеки, библиотеки импорта имеют расширение .LIB и используются компоновщиком для разрешения ссылок на функции в исходном коде программы. Однако, в библиотеках импорта программного кода нет. Вместо них в библиотеках импорта находится информация, необходимая компоновщику для установки таблицы ссылок внутри файла .EXE, предназначенной для динамического связывания. Файлы KERNEL32.LIB, USER32.LIB и GDI32.LIB, поставляемые с компилятором Microsoft, являются баблиотеками импорта функций Windows. Если в программе вызывается функция *Rectangle*, GDI32.LIB сообщает компоновщику, что данная функция находится в динамически подключаемой библиотеке GDI32.DLL. Эта информация попадает в файл .EXE, следовательно Windows может выполнить динамическое связывание с библиотекой GDI32.DLL во время выполнения программы.

Объектные библиотеки и библиотеки импорта используются только при разработке программы. Динамически подключаемые библиотеки используются во время выполнения программы. Динамически подключаемая библиотека должна находиться на диске, когда выполняется программа, использующая эту библиотеку. Если операционной системе Windows 95 требуется загрузить модуль динамически подключаемой библиотеки перед запуском программы, для которой этот модуль необходим, то файл библиотеки должен храниться в том же каталоге, что и файл .EXE программы, или в текущем каталоге, или в системном каталоге Windows, или в каталоге, доступном из переменной PATH окружения MS-DOS. (Именно в таком порядке в каталогах происходит поиск.)

Пример простой DLL

Как обычно, начнем с простого примера. На рис. 19.1 представлен исходный текст программы динамически подключаемой библиотеки, которая называется EDRLIB, и содержит одну функцию. Символы "EDR" в этом названии означают "простая подпрограмма рисования" (easy drawing routine). Вы можете легко добавлять в эту библиотеку другие функции, упрощающие процесс рисования в приложениях. Единственной функцией библиотеки EDRLIB является функция *EdrCenterText*, которая просто помещает в центр прямоугольника оканчивающуюся нулем строку текста, также, как функция *DrawText*, но без такого количества параметров.

EDRLIB.MAK

```
#----------------------
# EDRLIB.MAK make file
#----------------------

edrlib.dll : edrlib.obj
     $(LINKER) $(DLLFLAGS) -OUT:edrlib.dll edrlib.obj $(GUILIBS)

edrlib.obj : edrlib.c edrlib.h
     $(CC) $(CFLAGS) edrlib.c
```

EDRLIB.H

```
/*---------------------
   EDRLIB.H header file
  ---------------------*/

#define EXPORT extern "C" __declspec(dllexport)

EXPORT BOOL CALLBACK EdrCenterText(HDC, PRECT, PSTR);
```

EDRLIB.C

```
/*--------------------------------------------
   EDRLIB.C -- Easy Drawing Routine Library module
               (c) Charles Petzold,  1996
  --------------------------------------------*/

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

int WINAPI DllMain(HINSTANCE hInstance, DWORD fdwReason, PVOID pvReserved)
     {
     return TRUE;
     }
EXPORT BOOL CALLBACK EdrCenterText(HDC hdc, PRECT prc, PSTR pString)
     {
     int  iLength;
     SIZE size;

     iLength = strlen(pString);

     GetTextExtentPoint32(hdc, pString, iLength, &size);

     return TextOut(hdc,(prc->right - prc->left - size.cx) / 2,
                        (prc->bottom - prc->top - size.cy) / 2,
                    pString, iLength);
     }
```

Рис. 19.1 Библиотека EDRLIB

Как можно заметить, в make-файле имеется два отличия от make-файлов, которые мы уже использовали для других приложений. Во-первых, в командной строке компоновщика переменная среды DLLFLAGS заменила переменную среды GUIFLAGS. Она включена в файл MSC.BAT, с которым мы познакомились в главе 1, и просто добавляет к командной строке параметр DLL. Этот параметр сообщает компоновщику, что результирующий файл должен быть динамически подключаемой библиотекой. Кроме того, обратите внимание, что вместо создания файла с расширением .EXE, make-файл создает файл EDRLIB.DLL.

Вначале файла EDRLIB.H определяется макроконстанта EXPORT:

```
#define EXPORT extern "C" __declspec(dllexport)
```

Использование ключевого слова EXPORT при определении функции в вашей динамически подключаемой библиотеке сообщит компоновщику, что функции доступны для использования другими программами. Функция *EdrCenterText* определяется в заголовочном файле с помощью ключевого слова EXPORT. Кроме этого, точно также, как и оконная процедура, эта функция определяется с помощью константы CALLBACK.

Файл EDRLIB.C также несколько отличается от обычных файлов на языке C для Windows. В нем вместо функции *WinMain* имеется функция *DllMain*. Эта функция используется для выполнения инициализации и деинициализации, о чем будет рассказано позже в этой главе. В настоящий момент все, что нам необходимо — это чтобы ее возвращаемым значением было TRUE. И наконец, в файле EDRLIB.C имеется функция *EdrCenterText*.

Когда в командной строке системы вводится команда:

```
NMAKE EDRLIB.MAK
```

строится динамически подключаемая библиотека EDRLIB.DLL.

Кроме этого файла появляются еще два новых файла. Библиотека импорта EDRLIB.LIB вскоре нам понадобится. Библиотека экспорта EDRLIB.EXP — это побочный эффект процесса компоновки. Ее можно удалить.

Работает ли библиотека EDRLIB.DLL? Попытаемся это проверить. Программа EDRTEST, представленная на рис. 19.2, использует библиотеку EDRLIB.DLL для вывода в центр своей рабочей области строки текста.

EDRTEST.MAK

```
#----------------------
# EDRTEST.MAK make file
#----------------------

edrtest.exe : edrtest.obj edrlib.lib
     $(LINKER) $(GUIFLAGS) -OUT:edrtest.exe edrtest.obj edrlib.lib $(GUILIBS)

edrtest.obj : edrtest.c edrlib.h
     $(CC) $(CFLAGS) edrtest.c
```

EDRTEST.C

```
/*--------------------------------------------------------
   EDRTEST.C -- Program using EDRLIB dynamic link library
              (c) Charles Petzold, 1996
  --------------------------------------------------------*/

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

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

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

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

     RegisterClassEx(&wndclass);
     hwnd = CreateWindow(szAppName, "DLL Demonstration Program",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

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

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

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

```
{
HDC         hdc;
PAINTSTRUCT ps;
RECT        rect;

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

          GetClientRect(hwnd, &rect);

          EdrCenterText(hdc, &rect,
                        "This string was displayed by a DLL");

          EndPaint(hwnd, &ps);
          return 0;

     case WM_DESTROY :
          PostQuitMessage(0);
          return 0;
     }
return DefWindowProc(hwnd, iMsg, wParam, lParam);
}
```

Рис. 19.2 Программа EDRTEST

Эта программа выглядит как обычная программа для Windows, чем она в сущности и является. Тем не менее обратите внимание, что в файл EDRTEST.C с помощью инструкции *include* включен заголовочный файл EDRLIB.H, в котором определяется функция *EdrCenterText*, вызываемая программой при обработке сообщения WM_PAINT. Также обратите внимание, что в make-файл EDRTEST.MAK в инструкцию для компоновки включен файл EDRLIB.LIB. Эта библиотека импорта обеспечивает компоновщик информацией, необходимой для разрешения ссылки на файл EDRLIB.DLL в файле EDRTEST.EXE.

Чрезвычайно важно понимать, что сам текст функции *EdrCenterText* не включается в файл EDRTEST.EXE. Вместо этого там просто имеется ссылка на файл EDRLIB.DLL и функцию *EdrCenterText*, которая находится в этом файле. Файл EDRTEST.EXE требует запуска файла EDRLIB.DLL.

Заголовочный файл EDRLIB.H включен в файл с исходным текстом программы EDRTEST.C так же, как туда включен файл WINDOWS.H. Включение библиотеки импорта EDRLIB.LIB в командную строку для компоновки аналогично включению туда всех библиотек импорта Windows (например, USER32.LIB), перечисленных в переменной среды GUILIBS, на которую ссылается make-файл. Когда программа работает, она подключается к библиотеке EDRLIB.DLL точно также, как к библиотеке USER32.DLL. Грандиозно! Мы создали дополнение для Windows 95!

Перед тем, как продолжить, необходимо сказать несколько слов о динамически подключаемых библиотеках. Несмотря на категоричность утверждения, что динамически подключаемые библиотеки — это дополнение Windows 95, это еще и дополнение вашего собственного приложения. Все, что делает динамически подключаемая библиотека, делается от имени приложения. Например, владельцем всей выделяемой библиотекой памяти является приложение. Владельцем всех создаваемых библиотекой окон является приложение. Владельцем всех открываемых библиотекой файлов является приложение. Несколько приложений одновременно могут использовать одну и ту же динамически подключаемую библиотеку, но под Windows 95 они не мешают друг другу. Однако, если написать динамически подключаемую библиотеку с функциями, которые могут вызываться из нескольких потоков одной программы, то компилировать библиотеку следует не с переменной среды GFLAGS, а с переменной среды GFLAGSMT.

Поскольку код защищен от записи, один и тот же код динамически подключаемой библиотеки может использоваться разными процессами. Однако данные, с которыми работает DLL — для каждого процесса свои. Каждый процесс имеет собственное адресное пространство для всех данных, которые использует DLL. Разделение памяти между процессами требует (как дальше будет видно) дополнительных усилий.

Разделяемая память в DLL

Прекрасно, что Windows 95 изолирует друг от друга приложения, которые в одно и то же время используют одни и те же динамически подключаемые библиотеки. Однако, иногда это нежелательно. Может понадобиться написать DLL, содержащую некоторую область памяти, которая могла бы быть разделена между различными

приложениями, или, может быть, различными экземплярами одного приложения. Это подразумевает использование разделяемой памяти (фактически, файла, проецируемого в память), о которой упоминалось в главе 13, и было обещано позже объяснить ее использование.

Давайте рассмотрим, как это работает на примере программы STRPROG ("string program") и динамически подключаемой библиотеки STRLIB ("string library"). В STRLIB имеется три экспортируемые функции, которые вызываются из STRPROG. Только для того, чтобы сделать пример интереснее, одна из функций в STRLIB использует функцию обратного вызова, определенную в STRPROG.

STRLIB является модулем динамически подключаемой библиотеки, в котором хранятся и сортируются до 256 символьных строк. Строки переводятся в верхний регистр и запоминаются в разделяемой памяти STRLIB. Программа STRPROG может использовать функции из библиотеки STRLIB для добавления строк, удаления строк и получения из STRLIB всех имеющихся там строк. В программе имеется два пункта меню (Enter и Delete), которые вызывают окна диалога для добавления и удаления таких строк. Список всех находящихся в данный момент в STRLIB строк программа STRPROG выводит в своей рабочей области.

Приведенная ниже функция, определенная в STRLIB, добавляет строку в разделяемую память этой библиотеки:

```
EXPORT BOOL CALLBACK AddString(PSTR pStringIn)
```

Параметром *pStringIn* этой функции является указатель на строку. Строка преобразуется к верхнему регистру внутри функции *AddString*. Если такая строка в списке строк библиотеки STRLIB уже существует, функция добавляет копию строки. Функция *AddString* возвращает TRUE (ненулевое значение), если ее вызов проходит удачно, и FALSE (0) — в противном случае. Значение FALSE может оказаться результатом того, что длина строки равна 0, или результатом того, что для хранения строки невозможно выделить память, или того, что 256 строк уже хранятся.

Следующая функция библиотеки STRLIB удаляет строку из разделяемой памяти этой библиотеки:

```
EXPORT BOOL CALLBACK DeleteString(PSTR pStringIn)
```

И снова, параметром *pStringIn* этой функции является указатель на строку. Если более чем одна строка совпадает, удаляется только первая. Функция *DeleteString* возвращает TRUE (ненулевое значение), если ее вызов проходит удачно, и FALSE (0) — в противном случае. Значение FALSE означает, что длина строки равна 0, или что совпадающей строки найти не удалось.

Следующая функция библиотеки STRLIB использует для упорядочения строк, которые в данный момент хранятся в разделяемой памяти этой библиотеки, функцию обратного вызова, находящуюся в вызывающей программе:

```
EXPORT int CALLBACK GetStrings(PSTRCB pfnGetStrCallBack, PVOID pParam)
```

Функция обратного вызова должна быть определена следующим образом:

```
EXPORT BOOL CALLBACK GetStrCallBack(PSTR pString, PVOID pParam)
```

Параметр *pfnGetStrCallBack* функции *GetStrings* является указателем на функцию обратного вызова. Функция *GetStrings* вызывает функцию *GetStrCallBack* по разу для каждой строки или до тех пор, пока возвращаемым значением функции обратного вызова не станет FALSE (0). Возвращаемым значением функции *GetStrings* является число строк, переданных функции обратного вызова. Параметр *pParam* — это указатель на определенные программистом данные.

Библиотека STRLIB

На рис. 19.3 представлено три файла, необходимых для создания модуля динамически подключаемой библиотеки STRLIB.DLL.

STRLIB.MAK

```
#----------------------
# STRLIB.MAK make file
#----------------------

strlib.dll : strlib.obj
     $(LINKER) $(DLLFLAGS) -SECTION:shared,rws -OUT:strlib.dll \
               strlib.obj $(GUILIBS)

strlib.obj : strlib.c strlib.h
     $(CC) $(CFLAGS) strlib.c
```

STRLIB.H

```
/*----------------------
   STRLIB.H header file
```

```
  ----------------------*/

typedef BOOL(CALLBACK *PSTRCB)(PSTR, PVOID);
#define MAX_STRINGS 256

#define EXPORT  extern "C" __declspec(dllexport)

EXPORT BOOL CALLBACK AddString   (PSTR);
EXPORT BOOL CALLBACK DeleteString(PSTR);
EXPORT int  CALLBACK GetStrings  (PSTRCB, PVOID);
STRLIB.C
/*-----------------------------------------------
   STRLIB.C -- Library module for STRPROG program
               (c) Charles Petzold,  1996
  ----------------------------------------------*/
#include <windows.h>
#include "strlib.h"

#pragma data_seg("shared")

PSTR pszStrings[MAX_STRINGS] = { NULL };
int  iTotal = 0;

#pragma data_seg()

int WINAPI DllMain(HINSTANCE hInstance, DWORD fdwReason, PVOID pvReserved)
     {
     int i;

     switch(fdwReason)
          {
          // Nothing to do when process(or thread) begins

          case DLL_PROCESS_ATTACH :
          case DLL_THREAD_ATTACH :
          case DLL_THREAD_DETACH :
               break;

          // When process terminates, free any remaining blocks

          case DLL_PROCESS_DETACH :
               for(i = 0; i < iTotal; i++)
                    UnmapViewOfFile(pszStrings[i]);
               break;
          }

     return TRUE;
     }

EXPORT BOOL CALLBACK AddString(PSTR pStringIn)
     {
     HANDLE hString;
     PSTR   pString;
     int    i, iLength, iCompare;

     if(iTotal == MAX_STRINGS - 1)
          return FALSE;

     iLength = strlen(pStringIn);
     if(iLength == 0)
          return FALSE;

     hString = CreateFileMapping((HANDLE) -1, NULL, PAGE_READWRITE,
                                 0, 1 + iLength, NULL);
     if(hString == NULL)
```

```
          return FALSE;
     pString =(PSTR) MapViewOfFile(hString, FILE_MAP_WRITE, 0, 0, 0);
     strcpy(pString, pStringIn);
     AnsiUpper(pString);

     for(i = iTotal; i > 0; i--)
          {
          iCompare = strcmpi(pStringIn, pszStrings[i - 1]);

          if(iCompare >= 0)
               break;

          pszStrings[i] = pszStrings[i - 1];
          }

     pszStrings[i] = pString;

     iTotal++;
     return TRUE;
     }

EXPORT BOOL CALLBACK DeleteString(PSTR pStringIn)
     {
     int i, j, iCompare;

     if(0 == strlen(pStringIn))
          return FALSE;

     for(i = 0; i < iTotal; i++)
          {
          iCompare = lstrcmpi(pszStrings[i], pStringIn);

          if(iCompare == 0)
               break;
          }

     // If given string not in list, return without taking action

     if(i == iTotal)
          return FALSE;

     // Else free memory occupied by the string and adjust list downward

     UnmapViewOfFile(pszStrings[i]);

     for(j = i; j < iTotal; j++)
          pszStrings[j] = pszStrings[j + 1];

     pszStrings[iTotal--] = NULL;    // Destroy unused pointer
     return TRUE;
     }
EXPORT int CALLBACK GetStrings(PSTRCB pfnGetStrCallBack, PVOID pParam)
     {
     BOOL bReturn;
     int  i;

     for(i = 0; i < iTotal; i++)
          {
          bReturn = pfnGetStrCallBack(pszStrings[i], pParam);

          if(bReturn == FALSE)
               return i + 1;
          }
     return iTotal;
     }
```

Рис. 19.3 Библиотека STRLIB

Точка входа/выхода библиотеки

Как можно заметить, в файле STRLIB.C мы некоторым образом задействовали функцию *DllMain*. Эта функция вызывается при первом запуске библиотеки и при завершении ее работы. Хотя функция *DllMain* включена еще и в EDRLIB.C, в действительности это не обязательно; функция, выполняющая такие действия, была бы включена туда компоновщиком по умолчанию.

Первым параметром функции *DllMain* является описатель экземпляра библиотеки. Если в библиотеке используются ресурсы, для которых требуется описатель экземпляра (например, *DialogBox*), необходимо сохранить *hInstance* в глобальной переменной. Последний параметр функции *DllMain* резервируется системой.

Параметр *fdwReason* может принимать одно из четырех значений, которое идентифицирует причину вызова функции *DllMain* системой Windows 95. Перед тем, как продолжить, запомните, что одна и та же программа может быть загружена несколько раз, и ее экземпляры могут вместе работать под Windows. Каждая загруженная программа рассматривается как отдельный процесс.

Значение DLL_PROCESS_ATTACH параметра *fdwReason* означает, что динамически подключаемая библиотека отображена в адресном пространстве процесса. Это сигнал библиотеке выполнить какие-то задачи по инициализации, которые требуются для обслуживания последующих запросов от процесса. Такая инициализация может включать в себя, например, выделение памяти. Во время выполнения программы функция *DllMain* вызывается с параметром DLL_PROCESS_ATTACH только однажды. Любой другой процесс, использующий ту же динамически подключаемую библиотеку, приводит к новому вызову функции *DllMain* с параметром DLL_PROCESS_ATTACH, но это происходит уже от имени нового процесса.

Если инициализация проходит удачно, возвращаемым значением функции *DllMain* должно быть ненулевое значение. Нулевое возвращаемое значение приведет к тому, что Windows не запустит программу.

Если параметр *fdwReason* имеет значение DLL_PROCESS_DETACH, то это означает, что динамически подключаемая библиотека больше процессу не нужна. Это дает возможность библиотеке освободить занимаемые ресурсы системы. В Windows 95 это не является абсолютно необходимым, но это хороший стиль программирования.

Аналогично, если функция *DllMain* вызывается с параметром DLL_THREAD_ATTACH, это означает, что связанный процесс создал новый поток. Когда поток завершается, Windows вызывает функцию *DllMain* с параметром DLL_THREAD_DETACH. Запомните, что существует вероятность того, что вызов функции *DllMain* с параметром DLL_THREAD_DETACH произойдет без предварительного вызова ее с параметром DLL_THREAD_ATTACH. Это возможно в том случае, если библиотека была загружена после создания потока.

Когда функция *DllMain* вызывается с параметром DLL_THREAD_DETACH, поток еще существует. Динамически подключаемая библиотека может даже посылать сообщения потоку в этот момент. Но при этом нельзя использовать функцию *PostMessage*, поскольку поток, вероятно, закончится еще до того, как такое сообщение будет обработано.

Кроме функции *DllMain*, в STRLIB имеется только три функции, которые будут экспортированы для использования другими программами. Все эти функции определяются как EXPORT. Это приводит к тому, что компоновщик заносит их в библиотеку импорта STRLIB.LIB.

Программа STRPROG

Программа STRPROG, представленная на рис. 19.4, совершенно проста и понятна. Две опции меню (Enter и Delete) вызывают появление окон диалога для ввода строки. Затем в программе STRPROG вызываются функции *AddString* и *DeleteString*. Если программе нужно обновить свою рабочую область, вызывается функция *GetStrings*, а для получения списка перенумерованных строк вызывается функция *GetStrCallBack*.

STRPROG.MAK

```
#----------------------
# STRPROG.MAK make file
#----------------------

strprog.exe : strprog.obj strprog.res strlib.lib
     $(LINKER) $(GUIFLAGS) -OUT:strprog.exe strprog.obj strprog.res \
               strlib.lib $(GUILIBS)

strprog.obj : strprog.c strprog.h strlib.h
     $(CC) $(CFLAGS) strprog.c
```

```
strprog.res : strprog.rc strprog.h
     $(RC) $(RCVARS) strprog.rc
```

STRPROG.C

```
/*--------------------------------------------------------
   STRPROG.C -- Program using STRLIB dynamic link library
              (c) Charles Petzold, 1996
  --------------------------------------------------------*/

#include <windows.h>
#include <string.h>
#include "strprog.h"
#include "strlib.h"

#define MAXLEN 32
#define WM_DATACHANGE WM_USER

typedef struct
     {
     HDC  hdc;
     int  xText;
     int  yText;
     int  xStart;
     int  yStart;
     int  xIncr;
     int  yIncr;
     int  xMax;
     int  yMax;
     }
     CBPARAM;

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

char szAppName[] = "StrProg";
char szString[MAXLEN];

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
     {
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wndclass;

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

     RegisterClassEx(&wndclass);

     hwnd = CreateWindow(szAppName, "DLL Demonstration Program",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, iCmdShow);
```

```
     UpdateWindow(hwnd);

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

BOOL CALLBACK DlgProc(HWND hDlg, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     switch(iMsg)
          {
          case WM_INITDIALOG :
               SendDlgItemMessage(hDlg, IDD_STRING, EM_LIMITTEXT,
                                  MAXLEN - 1, 0);
               return TRUE;

          case WM_COMMAND :
               switch(wParam)
                    {
                    case IDOK :
                         GetDlgItemText(hDlg, IDD_STRING, szString, MAXLEN);
                         EndDialog(hDlg, TRUE);
                         return TRUE;

                    case IDCANCEL :
                         EndDialog(hDlg, FALSE);
                         return TRUE;
                    }
          }
     return FALSE;
     }

BOOL CALLBACK EnumCallBack(HWND hwnd, LPARAM lParam)
     {
     char szClassName[16];

     GetClassName(hwnd, szClassName, sizeof(szClassName));

     if(0 == strcmp(szClassName, szAppName))
          SendMessage(hwnd, WM_DATACHANGE, 0, 0);

     return TRUE;
     }

BOOL CALLBACK GetStrCallBack(PSTR pString, CBPARAM *pcbp)
     {
     TextOut(pcbp->hdc, pcbp->xText, pcbp->yText,
               pString, strlen(pString));

     if((pcbp->yText += pcbp->yIncr) > pcbp->yMax)
          {
          pcbp->yText = pcbp->yStart;
          if((pcbp->xText += pcbp->xIncr) > pcbp->xMax)
               return FALSE;
          }
     return TRUE;
     }

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

```
CBPARAM          cbparam;
HDC              hdc;
PAINTSTRUCT      ps;
TEXTMETRIC       tm;

switch(iMsg)
     {
     case WM_CREATE :
          hInst =((LPCREATESTRUCT) lParam)->hInstance;
          hdc   = GetDC(hwnd);
          GetTextMetrics(hdc, &tm);
          cxChar =(int) tm.tmAveCharWidth;
          cyChar =(int)(tm.tmHeight + tm.tmExternalLeading);
          ReleaseDC(hwnd, hdc);

          return 0;

     case WM_COMMAND :
          switch(wParam)
               {
               case IDM_ENTER :
                    if(DialogBox(hInst, "EnterDlg", hwnd, &DlgProc))
                         {
                         if(AddString(szString))
                              EnumWindows(&EnumCallBack, 0);
                         else
                              MessageBeep(0);
                         }
                    break;

               case IDM_DELETE :
                    if(DialogBox(hInst, "DeleteDlg", hwnd, &DlgProc))
                         {
                         if(DeleteString(szString))
                              EnumWindows(&EnumCallBack, 0);
                         else
                              MessageBeep(0);
                         }
                    break;
               }
          return 0;

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

     case WM_DATACHANGE :
          InvalidateRect(hwnd, NULL, TRUE);
          return 0;

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

          cbparam.hdc   = hdc;
          cbparam.xText = cbparam.xStart = cxChar;
          cbparam.yText = cbparam.yStart = cyChar;
          cbparam.xIncr = cxChar * MAXLEN;
          cbparam.yIncr = cyChar;
          cbparam.xMax  = cbparam.xIncr *(1 + cxClient / cbparam.xIncr);
          cbparam.yMax  = cyChar *(cyClient / cyChar - 1);

          GetStrings((PSTRCB) GetStrCallBack,(PVOID) &cbparam);

          EndPaint(hwnd, &ps);
```

```
               return 0;

          case WM_DESTROY :
               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

STRPROG.RC

```
/*---------------------------
   STRPROG.RC resource script
  ---------------------------*/

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

StrProg MENU
     {
     MENUITEM  "&Enter!",  IDM_ENTER
     MENUITEM  "&Delete!", IDM_DELETE
     }

EnterDlg DIALOG 24, 24, 190, 44
     STYLE WS_POPUP | WS_DLGFRAME
     {
     LTEXT          "&Enter:",  0,            4,  8,  24,  8
     EDITTEXT                   IDD_STRING,  32,  6, 154, 12
     DEFPUSHBUTTON  "Ok",       IDOK,         44, 24,  32, 14
     PUSHBUTTON     "Cancel",   IDCANCEL,    114, 24,  32, 14
     }

DeleteDlg DIALOG 24, 24, 190, 44
     STYLE WS_POPUP | WS_DLGFRAME
     {
     LTEXT          "&Delete:", 0,            4,  8,  28,  8
     EDITTEXT                   IDD_STRING,  36,  6, 150, 12
     DEFPUSHBUTTON  "Ok",       IDOK,         44, 24,  32, 14
     PUSHBUTTON     "Cancel",   IDCANCEL,    114, 24,  32, 14
     }
```

STRPROG.H

```
/*----------------------
   STRPROG.H header file
  ----------------------*/

#define IDM_ENTER      1
#define IDM_DELETE     2
#define IDD_STRING  0x10
```

Рис. 19.4 Программа STRPROG

В файл STRPROG.C включен заголовочный файл STRPROG.H, в котором просто определены константы, используемые в файле описания ресурсов STRPROG.RC. Туда также включен заголовочный файл STRLIB.H, в котором определены три функции из библиотеки STRLIB, которые будут использоваться в программе STRPROG.

Работа программы STRPROG

После того, как созданы файлы STRLIB.DLL и STRPROG.EXE, можно запускать программу STRPROG. Перед тем, как это сделать, удостоверьтесь, что файл STRLIB.DLL находится в текущем каталоге или каталоге, который доступен Windows (об этом ранее говорилось). Windows должна иметь возможность загрузить файл STRLIB.DLL при выполнении программы STRPROG. Если Windows не сможет найти файл STRLIB.DLL, на экран будет выведено окно сообщения, информирующее об этой ошибке.

При выполнении файла STRPROG.EXE, Windows реализует связывание с функциями во внешних модулях библиотек. Многие из этих функций находятся в обычных динамически подключаемых библиотеках Windows. Но Windows также видит, что программа вызывает три функции из STRLIB, и поэтому, Windows загружает файл STRLIB.DLL в память и вызывает функцию инициализации STRLIB. Внутри STRPROG вызовы этих трех функций динамически связываются с функциями библиотеки STRLIB. После этого программу STRPROG можно использовать для добавления или удаления строк из внутренней таблицы библиотеки STRLIB. В рабочей области программы STRPROG выводятся строки, находящиеся в данный момент в таблице.

Вызов из программы STRPROG функций *AddString*, *DeleteString* и *GetStrings*, находящихся в библиотеке STRLIB, очень эффективен и не требует затрат больших, чем вызов любого другого внешнего модуля. Фактически, связь между STRPROG и STRLIB также эффективна, как если бы три функции библиотеки STRLIB были обычными функциями программы STRPROG. Вы спросите, почему? Зачем нужно делать эту динамически подключаемую библиотеку? Нельзя ли включить коды этих трех функций в файл STRPROG.EXE?

Можно конечно. В определенном смысле библиотека STRLIB — это не больше, чем дополнение программы STRPROG. Однако, вам может быть интересно узнать, что случится, если запустить второй экземпляр программы STRPROG. Библиотека STRLIB запоминает строки символов и их указатели в разделяемой памяти, что позволяет всем экземплярам программы STRPROG совместно использовать эти данные. Рассмотрим, как это делается.

Разделение данных между экземплярами программы STRPROG

Windows воздвигает как бы стену вокруг адресного пространства процесса Win32. Обычно, адресное пространство является частным, недоступным для других процессов. Но работа нескольких экземпляров программы STRPROG показывает, что библиотека STRLIB не имеет проблем при разделении своих данных между всеми экземплярами программы. При добавлении или удалении строки в окне программы STRPROG, изменение немедленно отражается в других окнах.

Библиотека STRLIB предлагает для совместного использования два типа данных: строки и указатели на строки. Для иллюстрации, в библиотеке для каждого типа данных используются разные методы разделения данных. В главе 13 один из методов мы уже встречали. Библиотека STRLIB запоминает каждую строку в файле, проецируемом в память, делая строку видимой для всех процессов.

Библиотека STRLIB хранит указатели на строки в специальной области памяти, которая обозначается как разделяемая (shared):

```
#pragma data_seg("shared")
PSTR pszStrings[MAX_STRINGS] = { NULL };
int iTotal = 0;

#pragma data_seg( )
```

Первая директива *#pragma* создает область данных, которая называется *shared*. Эту область можно назвать как угодно, хотя компоновщик распознает только первые восемь символов. Все инициализированные переменные, расположенные после директивы *#pragma* попадают в область памяти *shared*. Второй директивой *#pragma* отмечен конец области данных. Важно специально инициализировать эти переменные, в противном случае компилятор помещает их вместо области памяти *shared*, в обычную неинициализируемую область.

Компоновщику необходимо сообщить об области памяти *shared*. В командной строке компоновщика задайте параметр -SECTION следующим образом:

```
-SECTION: shared, rws
```

Буквы "rws" обозначают, что область памяти имеет атрибуты для чтения (read), записи (write) и разделения (shared) данных.

Теперь все экземпляры программы STRPROG видят один экземпляр строк и указателей. Функция *EnumCallBack* программы STRPROG служит для уведомления всех экземпляров программы STRPROG об изменении содержимого области данных библиотеки STRLIB. Вызов функции *EnumWindows* приводит к тому, что Windows вызывает функцию *EnumCallBack* с описателями всех родительских окон. Затем функция *EnumCallBack* проверяет соответствует ли имя класса каждого окна "StrProg"; если да, то функция посылает окну сообщение WM_DATACHANGE, определяемое в программе. Теперь можно легко представить себе модернизированную версию библиотеки STRLIB, управляющей базой данных, которая совместно используется несколькими экземплярами одной и той же программы или экземплярами нескольких программ.

Некоторые ограничения библиотек

Как уже говорилось, сам модуль динамически подключаемой библиотеки сообщений не получает. Однако, модуль библиотеки может вызывать функции *GetMessage* и *PeekMessage*. Сообщения, которые с помощью этих функций библиотека извлекает из очереди, фактически предназначены программе, вызывающей функции из библиотеки. В

общем, библиотека работает от имени вызвавшей ее программы, это правило остается верным для большинства функций Windows, вызываемых из библиотеки.

Динамически подключаемая библиотека может загружать ресурсы (такие, как значки, строки и битовые образы) либо из файла библиотеки, либо из файла программы, которая вызывает библиотеку. Функциям, которые загружают ресурсы требуется описатель экземпляра. Если библиотека использует собственный описатель экземпляра (который передается библиотеке при инициализации), то библиотека может получить ресурсы из собственного файла. Для загрузки ресурсов из вызывающего библиотеку файла программы с расширением .EXE, функции библиотеки требуется описатель экземпляра программы, вызвавшей функцию.

Регистрация классов окна и создание окон в библиотеке может оказаться несколько сложнее. И для структуры класса окна и для вызова функции *CreateWindow* требуется описатель экземпляра. Хотя при создании класса окна и самого окна можно использовать описатель экземпляра модуля библиотеки, сообщения окна по-прежнему будут проходить через очередь сообщений программы, вызвавшей библиотеку. Если необходимо создавать классы окна и сами окна внутри библиотеки, вероятно, лучше пользоваться описателем экземпляра вызывающей программы.

Поскольку сообщения для модальных окон диалога минуют очередь сообщений программы, с помощью вызова функции *DialogBox* можно создать в библиотеке модальное окно диалога. Описатель экземпляра можно взять из библиотеки, а параметр *hwndParent* функции *DialogBox* может быть задан равным NULL.

Динамическое связывание без импорта

Вместо того, чтобы Windows выполняла динамическое связывание при первой загрузке вашей программы в оперативную память, можно связать программу с модулем библиотеки во время выполнении программы. Например, можно было бы просто вызвать функцию *Rectangle*:

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

Это работает, поскольку программа была скомпонована с библиотекой импорта GDI32.LIB, в которой имеется адрес функции *Rectangle*.

Можно также вызвать функцию *Rectangle* и совершенно необычным образом. Сначала воспользуемся функцией *typedef* для определения типа функции *Rectangle*:

```
typedef BOOL(WINAPI *PFNRECT)(HDC, int, int, int, int);
```

Затем определяем две переменные:

```
HANDLE hLibrary;
PFNRECT pfnRectangle;
```

Теперь устанавливаем значение переменной *hLibrary* равным описателю библиотеки, а значение переменной *pfnRectangle* — равным адресу функции *Rectangle*:

```
hLibrary = LoadLibrary("GDI32.DLL");

pfnRectangle =(PFNRECT) GetProcAddress(hLibrary, "Rectangle"");
```

Функции *LoadLibrary* возвращает NULL, если не удается найти файл библиотеки или случается какая-то другая ошибка. Теперь можно вызывать функцию и затем освободить библиотеку:

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

FreeLibrary(hLibrary);
```

Если этот прием динамического связывания во время выполнения не имеет особого смысла для функции *Rectangle*, то смысл определенно появляется, если до начала выполнения программы неизвестно имя модуля библиотеки.

В приведенном выше коде используются функции *LoadLibrary* и *FreeLibrary*. Windows поддерживает для всех модулей библиотек счетчик ссылок. Вызов функции *LoadLibrary* приводит к увеличению на 1 значения счетчика ссылок. Счетчик ссылок также увеличивается на 1, когда Windows загружает любую программу, в которой используется библиотека. Вызов функции *FreeLibrary* приводит к уменьшению на 1 значения счетчика ссылок; то же происходит при завершении экземпляра программы, в которой используется библиотека. Если значение счетчика ссылок становится равным 0, Windows может удалить библиотеку из памяти, поскольку библиотека больше не нужна.

Библиотеки, содержащие только ресурсы

Любая функция в динамически подключаемой библиотеке, которую программы Windows или другие библиотеки могут использовать, должны экспортироваться. Однако, динамически подключаемая библиотека может не содержать никаких экспортируемых функций. Что же тогда содержится в DLL? Ответ — ресурсы.

Давайте поговорим о том, как работает приложение Windows, для которого требуются битовые образы. Обычно, они перечисляются в файле описания ресурсов программы и загружаются в оперативную память с помощью

функции *LoadBitmap*. Но может быть нужно создать несколько наборов битовых образов, причем каждый задается для одного из основных видеоадаптеров, используемых Windows. Имело бы прямой смысл хранить эти разные наборы битовых образов в разных файлах, поскольку пользователю мог бы понадобиться только один набор битовых образов на жестком диске. Такие файлы являются библиотеками, содержащими только ресурсы.

На рис. 19.5 показано, как создать файл библиотеки, названный BITLIB.DLL, содержащий только ресурсы — девять битовых образов. В файле BITLIB.RC перечислены все файлы отдельных битовых образов и каждому присвоен номер. Для создания файла BITLIB.DLL, необходимо девять битовых образов BITLIB1.BMP, BITLIB2.BMP и т. д. Можно использовать готовые битовые образы или создать их в программе PAINT, поставляемой с Windows.

BITLIB.MAK

```
#---------------------
# BITLIB.MAK make file
#---------------------

bitlib.dll : bitlib.obj bitlib.res
     $(LINKER) $(DLLFLAGS) -OUT:bitlib.dll bitlib.obj bitlib.res $(GUILIBS)

bitlib.obj : bitlib.c
     $(CC) $(CFLAGS) bitlib.c

bitlib.res : bitlib.rc
     $(RC) $(RCVARS) bitlib.rc
```

BITLIB.C

```
/*------------------------------------------------------------
   BITLIB.C -- Code entry point for BITLIB dynamic link library
               (c) Charles Petzold,  1996
  ------------------------------------------------------------*/

#include <windows.h>

int WINAPI DllMain(HINSTANCE hInstance, DWORD fdwReason, PVOID pvReserved)
     {
     return TRUE;
     }
```

BITLIB.RC

```
/*--------------------------
   BITLIB.RC resource script
  --------------------------*/

1 BITMAP bitmap1.bmp
2 BITMAP bitmap2.bmp
3 BITMAP bitmap3.bmp
4 BITMAP bitmap4.bmp
5 BITMAP bitmap5.bmp
6 BITMAP bitmap6.bmp
7 BITMAP bitmap7.bmp
8 BITMAP bitmap8.bmp
9 BITMAP bitmap9.bmp
```

Рис. 19.5 Библиотека BITLIB

Программа SHOWBIT, приведенная на рис. 19.6, считывает ресурсы битовых образов из библиотеки BITLIB и отображает их в левом верхнем углу рабочей области. Нажимая клавишу на клавиатуре, можно вывести последовательно все битовые образы.

SHOWBIT.MAK

```
#----------------------
# SHOWBIT.MAK make file
#----------------------

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

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

SHOWBIT.C

```
/*------------------------------------------------------------
   SHOWBIT.C -- Shows bitmaps in BITLIB dynamic link library
                (c) Charles Petzold, 1996
  ------------------------------------------------------------*/

#include <windows.h>

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

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

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

     RegisterClassEx(&wndclass);
     hwnd = CreateWindow(szAppName, "Show Bitmaps from BITLIB(Press Key)",
                         WS_OVERLAPPEDWINDOW,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         CW_USEDEFAULT, CW_USEDEFAULT,
                         NULL, NULL, hInstance, NULL);

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

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

void DrawBitmap(HDC hdc, int xStart, int yStart, HBITMAP hBitmap)
     {
     BITMAP bm;
     HDC    hMemDC;
     POINT  pt;

     hMemDC = CreateCompatibleDC(hdc);
     SelectObject(hMemDC, hBitmap);
     GetObject(hBitmap, sizeof(BITMAP),(PSTR) &bm);
     pt.x = bm.bmWidth;
     pt.y = bm.bmHeight;
```

```
     BitBlt(hdc, xStart, yStart, pt.x, pt.y, hMemDC, 0, 0, SRCCOPY);

     DeleteDC(hMemDC);
     }

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HINSTANCE  hLibrary;
     static int        iCurrent = 1;
     HBITMAP           hBitmap;
     HDC               hdc;
     PAINTSTRUCT       ps;

     switch(iMsg)
          {
          case WM_CREATE :
               if((hLibrary = LoadLibrary("BITLIB.DLL")) == NULL)
                    DestroyWindow(hwnd);

               return 0;

          case WM_CHAR :
               if(hLibrary)
                    {
                    iCurrent ++;
                    InvalidateRect(hwnd, NULL, TRUE);
                    }
               return 0;

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

               if(hLibrary)
                    {
                    if(NULL ==(hBitmap = LoadBitmap(hLibrary,
                                        MAKEINTRESOURCE(iCurrent))))
                         {
                         iCurrent = 1;
                         hBitmap = LoadBitmap(hLibrary,
                                        MAKEINTRESOURCE(iCurrent));
                         }

                    if(hBitmap)
                         {
                         DrawBitmap(hdc, 0, 0, hBitmap);
                         DeleteObject(hBitmap);
                         }
                    }

               EndPaint(hwnd, &ps);
               return 0;

          case WM_DESTROY :
               if(hLibrary)
                    FreeLibrary(hLibrary);

               PostQuitMessage(0);
               return 0;
          }
     return DefWindowProc(hwnd, iMsg, wParam, lParam);
     }
```

Рис. 19.6 Программа SHOWBIT

При обработке сообщения WM_CREATE, программа SHOWBIT получает описатель библиотеки BITLIB.DLL:

```
if((hLibrary = LoadLibrary("BITLIB.DLL")) == NULL) DestroyWindow(hwnd);
```

Если библиотеки BITLIB.DLL нет в текущем каталоге, программа SHOWBIT будет искать ее так, как уже говорилось ранее в этой главе. Если найти библиотеку не удается, Windows выводит на экран окно сообщения, извещающее об этой ошибке. Когда пользователь нажимает клавишу <OK>, функция *LoadLibrary* возвращает значение NULL, и программа SHOWBIT завершается.

Программа SHOWBIT, с помощью вызова функции *LoadBitmap* с описателем библиотеки и номером битового образа, может получить его описатель:

```
hBitmap = LoadBitmap(hLibrary, MAKEINTRESOURCE(iCurrent));
```

Эта функция вернет ошибку, если битовый образ, соответствующий числу *iCurrent*, содержит ошибку, или, если недостаточно памяти для загрузки битового образа.

При обработке сообщения WM_DESTROY, программа SHOWBIT освобождает библиотеку:

```
FreeLibrary(hLibrary);
```

Когда последний экземпляр программы SHOWBIT завершается, счетчик ссылок библиотеки BITLIB.DLL становится равным 0, и занимаемая библиотекой память освобождается.

Глава 20 Что такое OLE?

20

OLE (Object Linking and Embedding) — это набор стандартов для связи и внедрения объектов при создании компонентов программного обеспечения. Одним из стандартов OLE является спецификация модели составного объекта (Component Object Model, COM), основа для бинарных соединений между компонентами. Другим определяющим элементом OLE является набор динамически подключаемых библиотек, являющихся частью Windows 95 и Windows NT. Фирма Microsoft совместно с Digital, Software AG и Bristol Technologies приняла решение о переносе некоторых технологий из этих библиотек на платформы других операционных систем. Набор услуг, которые предлагает OLE, не постоянен. Microsoft постоянно модернизирует и расширяет операционную систему Windows, и точно также она готова развивать OLE, чтобы адаптировать к широкому диапазону требований по интеграции приложений. В соответствие с этим положением Network (сетевой) OLE, например, призван стать частью Windows NT версии 4.0, что даст возможность связывать компоненты в рамках сети.

OLE позволяет увеличить степень интеграции между программными модулями, для чего ранее требовались специальные условия, и следовательно является первым шагом в создании множества взаимозаменяемых компонентов программного обеспечения. Из такого множества пользователь мог бы в окне текстового процессора, купленного у одного поставщика, запускать программу проверки орфографии другого, а для предварительного просмотра выводимого на печать текста — программу третьего поставщика программного продукта.

К настоящему времени существует несколько категорий стандартных соединений OLE. Причем архитектура OLE не только допускает увеличение количества таких категорий, но делает неизбежным появление новых семейств соединений. До появления Windows 95 четырьмя основными категориями соединений OLE были: поддержка составных документов (compound document support), автоматизация OLE (OLE Automation), дочерние элементы управления OLE (OLE Controls) и расширенный интерфейс программирования приложений для обмена сообщениями (Extended Messaging Applications Programming Interface, Extended MAPI). Windows 95 добавила новое семейство компонентов OLE — расширения оболочки (shell extensions) — для создания тесной связи между программами-приложениями и рабочим столом (desktop) Windows 95. Другим OLE-компонентом, связанным с появлением Windows 95, является дочерний элемент управления диалогового окна — усовершенствованный редактор (Rich Edit Control), который представляет собой простой контейнер составных документов.

Контейнер составных документов (compound document container) создает составные документы (compound documents), в которых могут содержаться данные из разных, несвязанных между собой приложений. Контейнеры связываются с приложением-сервером объектов (object server) для обеспечения двунаправленного перемещения данных между сервером и составным документом. Примером составного документа является документ, образованный внедрением части электронной таблицы Excel в документ Microsoft Word for Windows. В этом примере данные из электронной таблицы Excel являются внедренным объектом (embedded object), а Word for Windows является контейнером составных документов. Однако из этого примера не видны все достоинства OLE, поскольку легко предположить, что между двумя программами одной фирмы существует некое соглашение. Поддержка составных документов OLE является действительно универсальной, что становится очевидным, когда программы нескольких независимых производителей — например, Adobe PageMaker или Micrografx Designer — обеспечивают такую же поддержку составных документов, как и Microsoft Word. В каждом из приложений-контейнеров могут находиться объекты электронных таблиц Excel, рисованные объекты Corel Draw, диаграммы Visio или объекты данных из любого OLE-совместимого сервера объектов.

Автоматизация обеспечивает механизм определения набора макросов. Макросы состоят из методов (methods) (другое название вызовов функций) и свойств (properties) (то есть, элементов данных, которые можно читать и записывать, или только читать, или только записывать). Термин "объект автоматизации" (automation object) относится к такому компоненту OLE, который обеспечивает поддержку макропримитивов. "Контроллер автоматизации" (automation controller) манипулирует методами и характеристиками объекта автоматизации. С помощью определенных OLE-стандартов для автоматизации, среды программирования, например, Visual Basic фирмы Microsoft, Delphi фирмы Borland или Power Builder фирмы PowerSoft, можно создавать контроллеры

автоматизации для обеспечения централизованного управления работой, распределенной по отдельным приложениям.

Элементы управления OLE (OLE Controls) являются третьим типом стандартных компонентов OLE; они напоминают элементы управления диалогового окна тем, что по существу являются специализированными дочерними окнами. Тем не менее, дочерние элементы управления OLE относятся не к окнам диалога, а к контейнерам элементов управления OLE (OLE control containers). Элементы управления OLE являются объектами автоматизации и экспортируют собственные макросы. Элементы управления OLE напоминают объекты составных документов тем, что могут сохранять информацию о своем состоянии в файле, созданном и управляемом приложением-контейнером. Элементы управления OLE определяют свойства окружения (ambient properties), атрибуты типа цвета фона и текущего шрифта, которые позволяют им визуально входить в свой контейнер. Элементы управления OLE могут посылать уведомления о событиях контейнеру элементов управления, и в этом смысле, они напоминают элементы управления Visual Basic. Фактически фирма Microsoft публично заявила, что элементы управления Visual Basic (VBXs) не будут поддерживаться в 32-разрядной среде и призвала всех производителей VBX модернизировать 16-разрядные элементы VBXs, превратив их в 32-разрядные элементы управления OLE.

Общим во всех стандартных соединениях OLE является то, что все они входят в модель составного объекта OLE (OLE Component Object Model, COM). Модель составного объекта является объектно-ориентированной спецификацией для определения *интерфейсов*, то есть точек соприкосновения компонентов. COM обеспечивает фундамент, на котором строятся все возможности OLE.

Следует учитывать, что перечень услуг рассматриваемой части OLE очень велик и продолжает расти! Поскольку в этой главе нет возможности охватить все аспекты OLE, внимание будет сосредоточено на основах модели составного объекта. В соответствие с этим, мы рассмотрим две программы-сервера OLE (один закрытый (private) компонент и один открытый (public) компонент) и две программы-клиента OLE. Более подробное рассмотрение многочисленных возможностей OLE можно найти в книге "*Inside OLE*", написанной Kraig Brockschmidt (Microsoft Press, 1995).

Основы OLE

Для знакомства с компонентами OLE требуется усвоить следующие основные правила.

Связь с библиотеками OLE

Перед тем, как выполнять любые операции OLE, процесс Win32 должен связаться с библиотеками OLE. Любой модуль внутри процесса — независимо от того, программа это или DLL (Dynamic Link Library) — может установить это соединение. Поскольку вполне допускается несколько попыток соединения, и поскольку один модуль не информируется о том, соединились другие модули или нет, каждый модуль, который нуждается в библиотеках OLE, должен пытаться установить связь от имени каждого процесса, в котором он запускается.

Самым обычным способом связаться с библиотеками OLE является вызов функции *OleInitialize*:

```
HRESULT hr = OleInitialize(NULL);

if(FAILED(hr))
{
       // Обработка ошибки
}
```

Единственный параметр функции *OleInitialize* должен быть равен NULL (в 16-разрядном OLE были допустимы и другие значения, но единственным правильным значением в 32-разрядном OLE является NULL). Эта функция устанавливает услуги OLE для текущего процесса, а затем вызывает дополнительную функцию установки *CoInitialize* для инициализации библиотеки модели составного объекта (Component Object Model library). Эта библиотека обеспечивает услуги нижнего уровня для создания компонента и его удаления.

В некоторых случаях, может понадобиться обойти вызов функции *OleInitialize* и непосредственно вызвать функцию *CoInitialize*. Например, если при создании пользовательского компонента OLE, требуются только базовые возможности модели составного объекта, то соединяться с OLE нужно с помощью функции *CoInitialize*. Выгода состоит в том, что требуется меньшая область памяти, поскольку фактически в нее загружается только часть динамически подключаемой библиотеки OLE. Вызов функции *CoInitialize* идентичен вызову функции *OleInitialize*:

```
HRESULT hr = CoInitialize(NULL);

if(FAILED(hr))
{
       // Обработка ошибки
}
```

Каждый компонент OLE, который пытается элементизовать соединение с библиотеками OLE, должен перед окончанием работы отсоединиться, если при попытке соединения в качестве возвращаемого значения не было получено кода ошибки. Для функции *OleInitialize* функцией отсоединения является *OleUninitialize*, а для функции *CoInitialize — CoUninitialize*. Можно догадаться, что функция *OleUninitialize* вызывает функцию *CoUninitialize*, поэтому компоненту OLE для отсоединения требуется вызов только той функции, которая соответствует вызываемой для создания соединения.

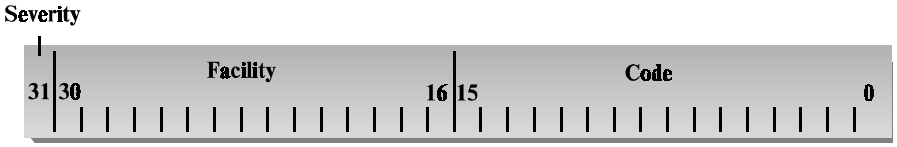
Возвращаемое значение обеих функций инициализации имеет тип HRESULT, сообщающий вызывающей функции об успешном или ошибочном результате. В предыдущих примерах демонстрировалось использование макрокоманды FAILED для отслеживания неудачных вызовов. Обратная макрокоманда SUCCEEDED в случае удачного вызова возвращает TRUE. Почти у всех функций OLE возвращаемое значение имеет тип HRESULT. Несколько советов помогут правильно интерпретировать его смысл.

Расшифровка кода результата

Общим для всего OLE в целом является использование стандартного типа возвращаемого значения HRESULT почти для всех функций библиотек OLE и почти для всех функций интерфейса OLE. В OLE имеются файлы, в которых этот тип определен следующим образом:

```
typedef LONG HRESULT;
```

Хотя он известен как описатель результата (result handle), это необычный описатель, который описывает объект. Это 32-разрядное значение, составленное из битовых полей, как показано на рис. 20.1.



Severity:	**(1 bit)**	**Severity field**
	0	**Success. The function was successful.**
	1	**Error. The function failed due to an error condition.**
Facility:	**(15 bits)**	**Indicates to which group of status codes this belongs. Microsoft reserves the exclusive right to define facility codes.**
Code:	**(16 bits)**	**Describes what actually took place, error or otherwise.**

Рис. 20.1 Битовые поля HRESULT

Наиболее важным полем в HRESULT является поле Severity (критическое), которое указывает на успех или ошибку. Макросы SUCCEEDED и FAILED определяют успех или ошибку исключительно на основе значения этого поля.

Поле Facility (компонент) идентифицирует компонент, который вызвал ошибку. (Во избежание конфликтов с кодами компонентов компании Microsoft, не создавайте собственных кодов компонентов.) В настоящее время, как показано в таблице, определено только несколько компонентов:

Имя компонента	Значение	Описание
FACILITY_NULL	0	Используется для широко применяемых кодов общего статуса, которые не относятся к какой-либо конкретной группе (например, S_OK).
FACILITY_RPC	1	Результат вызова удаленной процедуры.
FACILITY_DISPATCH	2	Результат относится к интерфейсу *IDispath*.
FACILITY_STORAGE	3	Результат относится к постоянной памяти. Коды результата, которые меньше чем 256, имеют тот же смысл, что и коды ошибок MS-DOS.
FACILITY_ITF	4	Наиболее часто используемый компонент. Значение зависит от интерфейса.
FACILITY_WIN32	7	Результат вызова функции Win32 *GetLastError*.
FACILITY_WINDOWS	8	Результат от интерфейса, определенного Microsoft.
FACILITY_CONTROL	10	Результат относящийся к элементам управления OLE.

Подробности находятся в поле Code, в разрядах от 0 до 15. В случае ошибки здесь находится код ошибки. И даже при успехе, как будет показано далее, оно заполняется кодом, позволяющим оценить успех.

Назначение разрядов в кодах HRESULT может зависеть от реализации. Определение успеха или ошибки возлагается только на макросы SUCCEEDED и FAILED. Чтобы выделить код компонента используйте макрокоманду HRESULT_FACILITY. А для кода ошибки используйте макрокоманду HRESULT_CODE. Это

гарантирует, что ваш код OLE будет легче перенести с Windows 95, например, на Apple Macintosh, Unix, VMS и любую другую платформу, которая поддерживает OLE.

В качестве примера значений HRESULT, в этой таблице объединены возвращаемые значения функций *CoInitialize* и *OleInitialize*:

Возвращаемое значение	Двоичное значение	Описание
S_OK	0x00000000L	Означает, что инициализация библиотеки прошла успешно.
S_FALSE	0x00000001L	Означает, что библиотека уже инициализирована.
E_OUTOFMEMORY	0x8007000E	Ошибка из-за нехватки памяти.
E_INVALIDARG	0x80070057	Ошибка из-за неправильного аргумента.
E_UNEXPECTED	0x8000FFFF	Неожиданная ошибка.

Когда модуль пытается инициализировать библиотеки OLE, возвращаемые значения S_OK и S_FALSE означают, что инициализация прошла успешно. Первый успешный вызов функции инициализации в процессе возвращает S_OK. Последующие вызовы функций инициализации не влияют на уже инициализированные библиотеки. Эти вызовы возвращают значение S_FALSE. Префикс "S_" означает "success" (успех) — точное название для обоих кодов результата, поскольку оба говорят о том, что OLE готов к использованию. Конечно, модуль всегда может проверить указанные значение для случаев, когда, например, отладчик является первым модулем, инициализирующим библиотеки OLE.

В документации OLE перечислены некоторые коды результата для большинства функций, но функция может (а часто так и происходит) возвращать ошибку с кодом результата, отличающимся от приводимых в документации значений. Трудный способ понять значение итогового кода HRESULT состоит в поиске внутри файлов OLE. Легкий способ — это вызов функции *FormatMessage*, которая представляет собой функцию Win32 для преобразования кода сообщения (или в нашем случае описателя результата OLE) в строку, пригодную для чтения. Далее представлена функция-надстройка над функцией *FormatMessage*, один из параметров которой является кодом ошибки OLE или Win32, а возвращаемым значением является текстовое описание, соответствующее коду ошибки:

```
// Функция DisplayErrorString формирует текстовую строку, соответствующую коду
// ошибки Win32 или OLE


void DisplayErrorString(DWORD dwError, BOOL bDebugger)
{
       CHAR achBuffer [120];

       FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, NULL,
                            dwError,
                            LANG_SYSTEM_DEFAULT,
                            achBuffer, 120,
                            NULL);

       if(bDebugger)
       {
              OutputDebugString(achBuffer);
       }
       else
       {
              MessageBox(NULL, achBuffer, "DescribeError", MB_OK);
       }
}
```

В этой функции имеется два параметра: *dwError* — значение типа HRESULT, возвращаемое функцией OLE, и *bDebugger* — значение типа Boolean, задающее, куда направить выходные данные. Если *bDebugger* равно TRUE, выходные данные попадают в окно отладчика (если отладчик не запущен, выходные данные пропадут) или в окно сообщения.

К сожалению, значения OLE типа HRESULT включены в коды ошибок Win32, что становится очевидным при просмотре файла Win32, который называется WINERROR.H. Одно из отличий между способом, которым обрабатываются коды ошибок в OLE и в Win32, состоит в том, что функции OLE возвращают значение типа HRESULT непосредственно, в то время как функции Win32 возвращают код ошибки косвенно. Для функций Win32 такие коды ошибок можно получить с помощью функции *GetLastError*. Возвращаемым значением этой функции является код ошибки последней функции, потерпевшей неудачу в текущем потоке. Некоторые процедуры Win32 стирают код ошибки — чтобы установить его в известное состояние перед вызовом других процедур Win32, которые могут потерпеть неудачу — поэтому перед вызовом других функций Win32, необходимо сразу проверять код ошибки.

Итак, мы коснулись двух основных концепций программирования OLE: во-первых, чтобы использовать OLE, необходимо в первую очередь связаться с библиотеками OLE. Во-вторых, перед расшифровкой возвращаемого значения функций типа HRESULT, необходимо определить успешной или неудачной была попытка установки соединения. После соединения с библиотеками OLE, у модуля есть выбор: соединиться с интерфейсом OLE (то есть, стать клиентом интерфейса OLE), или сделать интерфейс OLE доступным для других (то есть, стать сервером интерфейса OLE). Большинство программ, использующих OLE, делают и то, и другое. Чтобы вы не мучились вопросом, что следует рассматривать первым, следующий раздел посвящен основам интерфейсов OLE. Интерфейсы OLE (OLE interfaces) создаются на базе спецификации модели составного объекта (Component Object Model), и поэтому иногда их называют интерфейсами *модели составного объекта* или *COM-интерфейсами*.

Интерфейсы модели составного объекта (COM-интерфейсы)

Модель составного объекта — это спецификация для создания компонентов OLE. COM обеспечивает описание фундаментальных связей, которые необходимы между поставщиком интерфейса (interface provider) и пользователем интерфейса (interface user). Все интерфейсы OLE — для поддержки составных документов, для автоматизации OLE, для элементов управления OLE, для расширений оболочки Windows 95, для создания расширенного интерфейса программирования приложений обмена сообщениями — строятся в соответствие со спецификацией модели составного объекта.

Интерфейс модели составного объекта предлагает одно единственное решение проблемы, которая ранее требовала множества решений, а именно, соединение компонентов программного обеспечения, действующих в разных процессах или на разных компьютерах. Короче говоря, основная идея состоит в том, чтобы эффективно устанавливать связь между компонентами, разделенными границами адресного пространства, границами сетей или не имеющих границ вообще. Интерфейсы OLE обеспечивают как соединение между процессами, так и между машинами, и в то же время, эффективно управляют соединениями между компонентами, действующими в одном и том же адресном пространстве. Такое решение для всех типов соединений означает, что создатели программ могут игнорировать отличия, основанные только на местоположении компонентов, и сосредоточиться на задаче, как таковой. Это напоминает то, как используются телефоны: люди связываются друг с другом с помощью единственного механизма — самого телефона — независимо от того, находятся ли они в разных комнатах одного и того же здания, на противоположных сторонах одного и того же континента, или в разных полушариях Земли.

Как и во всех хороших проектах, в интерфейсах вместе сочетаются несколько различных вещей. Интерфейсы одновременно являются контрактами на услуги, бинарными соединениями между поставщиками услуг и пользователями этих услуг, и механизмом, поддерживающим соединения между процессами и между машинами. То, что связывает эти аспекты вместе, представляет собой набор основополагающих функций ядра, которые разделяются всеми интерфейсами. Каждая из них подлежит внимательному изучению.

Интерфейс — это контракт на услуги

Интерфейс представляет собой контракт на услуги, поставляемые одним компонентом — "сервером" и используемые другим — "клиентом". Единственный код, соответствующий определению интерфейса, состоит из списка прототипов функций — возвращаемых значений функций, имен функций и их параметров. Контракт задает то, *какие* услуги предоставляются, но не *как* они предоставляются. Детали реализации скрыты — то есть инкапсулированы (encapsulated) — внутри поставщика услуги.

Например, далее представлены прототипы функций для *Imalloc* — интерфейса выделения памяти OLE:

Прототип функции	Описание
void *Alloc (ULONG cb) ;	Выделяет *cb* байтов памяти.
void *Realloc (void *pv, ULONG cb) ;	Изменяет размер области *pv* памяти до *cb* байтов.
void Free (void *pv) ;	Освобождает область памяти, на которую указывает *pv*.
ULONG GetSize (void *pv) ;	Запрос размера памяти.
int DidAlloc (void *pv) ;	Выделена ли память данной программой?
void HeapMinimize (void) ;	Дефрагментирование памяти.

Услуги, поставляемые такой функцией выделения памяти, знакомы любому программисту, имеющему опыт работы с динамической памятью. Но ее *реализация*, то есть механизм удовлетворения этого запроса, не задается. При выполнении интерфейса *IMalloc* в Windows 95 мог бы использоваться любой доступный в Win32 механизм, включая функцию постраничного выделения памяти (*VirtualAlloc*) для больших объектов, функцию выделения памяти из "кучи" (*HeapAlloc*) для маленьких объектов или Win16 API-совместимые функции выделения памяти (*GlobalAlloc* или *LocalAlloc*), совместимые с Win16 и Win32. При реализации могли бы использоваться даже функции динамического выделения памяти языка C для создания переносимого интерфейса *IMalloc*, независимого от Win32 API.

Заслуживает внимания имя интерфейса *IMalloc*. Следуя венгерской системе обозначения, имена интерфейсов начинаются с префикса "I", например, *IStorage*, *IOleObject* и *IDataObject*. Также используется префикс "C" для имен классов библиотеки MFC фирмы Microsoft (*CWinApp, CFrameWnd*) или префикс "T" с именами классов библиотеки OWL фирмы Borland (*TApplication, Twindow*).

Хотя детали того, как реализован интерфейс скрыты, место бинарного соединения между клиентом и сервером вполне доступно и легко определяется.

Интерфейс определяет место бинарного соединения

По своей сути бинарная реализация интерфейса является массивом указателей на функции. Сервер для интерфейса типа *IMalloc* реализует конкретные функции и строит массив указателей на функции. Клиент ожидает доступа к функциям сервера через такой массив.

Специальным соединением, которым сервер обеспечивает клиентов, является указатель на массив указателей на функции (pointer to an array of function pointers). В языке C, синтаксис массива предоставляет один способ для создания массива указателей на функции. Сервер *IMalloc* мог бы следующим образом выделить память для массива указателей на функции:

```
// Определение базового типа указателя на функцию
typedef(* PFNINTERFACE)();

// Выделение памяти для массива указателей на функции
PFNINTERFACE allocator [6] = { Alloc, Realloc, Free, GetSize, DidAlloc, HeapMinimize };
```

Затем, в качестве возвращаемого значения, сервер мог бы вернуть искомое соединение — указатель на массив указателей на функции — любому клиенту, вызвавшему такую функцию:

```
PFNINTERFACE *FetchAllocator()
{
       return &allocator;
}
```

Проблема с синтаксисом массива в том, что вызовы функций *IMalloc* неуклюжи и могут вызвать ошибку. Возвращаемое значение должно быть преобразовано, нет возможности проверить параметры, а имя функции заменяется индексом массива. Например, вызов функции *Alloc* для выделения 20 байтов памяти выглядит следующим образом:

```
PFNINTERFACE *pAlloc = FetchAllocator( );

void *pData =(void *) pAlloc[0](20);
```

С запрещенной проверкой типов параметров, вызовы функций типа следующего не генерируют каких бы то ни было предупреждающих сообщений или сообщений об ошибках:

```
// Ошибка: неправильный тип параметра!!
pData =(void *) pMalloc[0]("20 bytes");

// Ошибка: неправильное число параметров!!
pData =(void *) pMalloc[0](20, 30, 40, 50, 60);
```

Более удачный подход состоит в использовании структуры с указателями на функции. При реализации этого подхода нужно сделать следующие объявления:

```
// Задание типов указателей на функции
typedef void *       (* PFNALLOC)         (ULONG);
typedef void *       (* PFNREALLOC)       (void *, ULONG);
typedef void         (* PFNFREE)                (void *);
typedef ULONG (* PFNGETSIZE)              (void *);
typedef int          (* PFNDIDALLOC)            (void *);
typedef void         (* PFNHEAPMINIMIZE)  (void);

// Определение структуры с указателями на функции
typedef struct tagALLOCATOR
{
       PFNALLOC            Alloc;
       PFNREALLOC          Realloc;
       PFNFREE             Free;
       PFNGETSIZE          GetSize;
       PFNDIDALLOC         DidAlloc;
       PFNHEAPMINIMIZE     HeapMinimize;
} ALLOCATOR;
```

```
ALLOCATOR imalloc = { Alloc, Realloc, Free, GetSize, DidAlloc, HeapMinimize };

ALLOCATOR *FetchAllocator( )
{
       return &imalloc;
}
```

Использование структуры — достаточно сложной для определения — делает вызовы простыми и естественными. В частности, преобразование типа возвращаемого значения больше не требуется, для вызова функций можно использовать их имена, а проверка типа параметров полностью реализуется компилятором. Ниже представлен вызов функции для выделения 20 байтов памяти:

```
ALLOCATOR *pAlloc = FetchAllocator( );

void *pData = pAlloc -> Alloc(20);
```

Лучшим подходом, который и выбран для интерфейсов OLE, является использование массива указателей на функции, создаваемого автоматически при объявлении класса C++. Массив указателей на функции создается для всех "виртуальных" функций — таких функций, которым предшествует ключевое слово *virtual*. Наличие прототипов функций в объявлении класса C++ обеспечивает проверку параметров без многократного использования *typedef*:

```
class IMalloc
{
       virtual void         *Alloc        (ULONG cb);
       virtual void         *Realloc      (void *pv, ULONG cb);
       virtual void         Free          (void *pv);
       virtual ULONG        GetSize       (void *pv);
       virtual int          DidAlloc      (void *pv);
       virtual void         HeapMinimize  (void);
};
```

Одно небольшое затруднение возникает при использовании классов C++. А именно, при вызове таких функций из C объект C++ требует дополнительный уровень в таблице функций. Это вынуждает создавать, вместо простого указателя *IMalloc**, указатель *IMalloc*** . Другая сложность заключается в том, что указатель на составной объект должен передаваться в качестве первого параметра для эмуляции неявной передачи указателя *this*, что необходимо для всех функций объектов C++. Далее показано, как в C вызвать функцию интерфейса *IMalloc*:

```
IMalloc *pMalloc = FetchMalloc();

void *pData = pMalloc -> lpVtbl -> Alloc(pMalloc, 20);
```

За исключением появления этих дополнительных сложностей, суть бинарного определения интерфейса остается той же — массив указателей на функции.

Важным аспектом интерфейсов OLE является поддержка связи между процессами и между машинами. Об этом в нашем введении в OLE рассказывается как просто об "удаленных соединениях" (remotable connections).

Интерфейс — это удаленное соединение

Для клиента бинарный интерфейс — массив функций — остается неизменным, независимо от того, действует ли соединение клиент/сервер в одном процессе (in-process), или в пределах нескольких процессов, или в рамках сети. Когда сервер работает со своим клиентом в одном процессе, один массив функций связывает воедино вызовы от клиента и функции в сервере. При вызове сервера извне (out-of-process), клиент по-прежнему видит только массив функций. Есть два типа серверов, обслуживающих запросы из других процессов: локальные серверы (local servers), которые действуют с клиентом в разных адресных пространствах одной машины, и удаленные серверы (remote servers), которые действуют на разных машинах. Для вызывающей программы отличие между локальными и удаленными услугами незаметно. Сервер обеспечивает необходимый механизм поддержки, так что его интерфейсы могут обеспечить бесперебойное соединение — через границу процессов или через границу сетей — при этом клиент не знает и не заботится о том, где "действительно" происходит действие.

Большую часть поддержки, необходимой для удаленного соединения, выполняют библиотеки OLE. В адресном пространстве процесса-клиента создается массив функций и соединяется с набором передающих функций-посредников (proxy functions). Эти функции, являющиеся логическим эквивалентом системы ретрансляции телефонного вызова, упаковывают параметры в некую оболочку для пересылки их "настоящему" интерфейсу в адресном пространстве сервера. В адресном пространстве сервера набор получающих функций-заглушек (stub functions) распаковывает оболочку и вызывает настоящие функции сервера, которые и используют переданные параметры.

Работа по упаковке, пересылке и распаковке, выполненная функциями-посредниками и заглушками называется формированием (marshaling). OLE по умолчанию обеспечивает поддержку формирования для всех определенных в

OLE интерфейсов, использующих механизм встроенных вызовов удаленных процедур (remote procedure call, RPC) для вызовов между процессами и между компьютерами. В случае определения пользовательского интерфейса — то есть нового интерфейса, не являющегося частью стандарта OLE — у него отсутствует встроенная поддержка удаленного вызова.

Для того, чтобы реализовать удаленный пользовательский интерфейс, в OLE предлагается два варианта. Первый предполагает разработку описания интерфейса на языке определения интерфейса (Interface Definition Language, IDL) и компиляцию его компилятором Microsoft IDL. Этот компилятор создает коды proxy и stub функций, которые встраиваются в DLL. Второй вариант предполагает самостоятельное управление всеми аспектами формирования, что позволяет использовать любой механизм передачи, например, разделяемую память, сообщения Windows, именованные каналы связи, RPC, азбуку Морзе и т. д. Эта возможность называется "пользовательским формированием" (custom marshaling) и означает, что сам OLE не участвует в вызовах между процессами и между компьютерами.

Таблица функций интерфейса OLE содержит два ряда функций: базовые функции интерфейса и специализированные функции интерфейса. В приведенном ранее интерфейсе *IMalloc* показаны только примеры функций, специализированных для задач выделения оперативной памяти. Пропущенными оказались три базовые функции, о которых рассказывается в следующем разделе, и которые всегда занимают первые три позиции любого массива функций интерфейса OLE.

В любом интерфейсе OLE имеются эти три функции

В каждом интерфейсе OLE имеется ряд общих функций интерфейса, предназначенных для поддержки базовой внутренней структуры объекта: *QueryInterface, AddRef* и *Release*. Вместе эти три функции создают свой собственный интерфейс, который называется *IUnknown*:

```
class IUnknown
{
       virtual HRESULT      QueryInterface(REFIID iid, void **ppv);
       virtual ULONG AddRef();
       virtual ULONG Release();
};
```

Целесообразность имени интерфейса "Unknown" (неизвестный) отражается в том факте, что доступ к интерфейсам осуществляется через указатели, и это оказывается полезным для создания родового (настраиваемого) указателя. Аналогично указателю *void* * в языках C и C++, родовой указатель обеспечивает минимальный набор известных действий, которые можно реализовать в любом интерфейсе. В частности, можно встретить функции OLE, возвращаемым значением которых являются разного рода указатели интерфейса. Прототипы таких функций часто составляются таким образом, чтобы их возвращаемое значение имело тип *IUnknown* *.

Для упрощения в приведенный ранее интерфейс *IMalloc* не были включены эти три функции интерфейса. В более точном объявлении интерфейса *IMalloc* имеется ссылка на *IUnknown*, как на базовый класс, все члены которого наследуются:

```
class IMalloc : public IUnknown
{
       virtual void         *Alloc(ULONG cb);
       virtual void         *Realloc(void *pv, ULONG cb);
       virtual void         Free(void *pv);
       virtual ULONG        GetSize(void *pv);
       virtual int          DidAlloc(void *pv);
       virtual void         HeapMinimize(void);
};
```

Замечание для программистов на языке C

В соответствие с механизмом наследования классов C++, добавление фразы: "*public IUnknown*" к определению интерфейса *Imalloc* эквивалентно добавлению ко всем элементам класса *IMalloc* элементов класса *IUnknown*. Девять функций-членов создают таблицу виртуальных функций класса *IMalloc* и определяют бинарное соединение с этим интерфейсом.

Кроме шести функций управления памятью, в интерфейсе *IMalloc* имеется также три функции, не связанные с управлением памятью. Как станет ясно в дальнейшем, эти три функции обеспечивают важное дополнение компонентов OLE. Как и любой другой интерфейс, *IUnknown* предоставляет контракт на услуги. Но, поскольку этот интерфейс всегда присутствует во всех других интерфейсах (в терминах C++ *IUnknown* является базовым

интерфейсом для любого другого интерфейса OLE), интерфейс *IUnknown* определяет те услуги, которые должен поставлять любой интерфейс OLE. Понимание значения этого контракта является важным независимо от того, какая программа, сервер или клиент, реализуется.

Услуги интерфейса *IUnknown*

До тех пор, пока они представлены посредством индивидуальных интерфейсов, три функции интерфейса *IUnknown* в действительности отражают не услуги, характерные для самого интерфейса, а услуги компонента OLE. Двумя базовыми услугами являются: управление временем жизни компонента и доступ к нескольким интерфейсам в одном компоненте. (Обратите внимание, что термин "компонент OLE" эквивалентен термину "объект OLE", который используется в спецификациях OLE.)

Управление "временем жизни компонента"

Говоря о времени жизни компонента, имеется в виду, что компонент должен прекратить работу и освободить ресурсы, когда в нем пропадает необходимость. Без этого решающего действия, даже системы с большим объемом оперативной памяти в конце концов столкнутся с ее нехваткой. Основные приемы очевидны: любой компонент OLE (объект OLE) поддерживает счетчик ссылок, который устанавливается в 1 при создании компонента. Вызовы функций *QueryInterface* и *AddRef* увеличивают на 1 значение счетчика ссылок, а вызов функции *Release* уменьшает это значение на 1. Когда счетчик доходит до нуля, компонент завершается, освобождая все захваченные системные ресурсы.

В простейшем случае, вызова функции *Release* достаточно для отсоединения клиента от интерфейса. Эту функцию поддерживает любой интерфейс, что означает отсутствие необходимости в задании специфичных для интерфейса или специфичных для компонента функций завершения; одна функция подходит всем интерфейсам. В следующем фрагменте программы показаны соединение, использование и отсоединение от одной из реализаций интерфейса *Imalloc* — процедуры выделения памяти для задач OLE. Компоненты OLE используют эту процедуру при работе с областью оперативной памяти, которая выделяется одним компонентом, а освобождается другим:

```
char          *pData;
HRESULT       hr;
IMalloc       *pMalloc;
// Соединение с процедурой выделения памяти

hr = CoGetMalloc(1, &pMalloc);

if(SUCCEEDED(hr))
{
       // Выделение памяти

       pData = pMalloc -> Alloc(cbBytes);

       if(pData)
       {

              lstrcpy(pData, pSource);
       }
       // Отсоединение от процедуры выделения памяти

       pMalloc -> Release();
}
```

Вызов *pMalloc -> Release* не освобождает выделенную память, а уменьшает на 1 значение счетчика ссылок компонента. Для освобождения памяти вызовите функцию *Free*, члена класса *IMalloc*:

```
hr = CoGetMalloc(1, &pMalloc);

if(SUCCEEDED(hr))
{

       pMalloc -> Free(pData);

       pMalloc -> Release();
}
```

Для упрощения доступа к процедуре выделения памяти, в библиотеки OLE были добавлены две простейшие функции упаковки: *CoTaskMemAlloc* и *CoTaskMemFree*. Выделяйте память с помощью вызова функции *CoTaskMemAlloc*:

```
pData = CoTaskMemAlloc(cbBytes);
```

Освобождайте память с помощью вызова функции *CoTaskMemFree*:

```
CoTaskMemFree(pData);
```

Более сложное управление временем жизни компонента имеет место, когда несколько различных частей клиента полагаются на услуги интерфейса. Частично это происходит потому, что программисты обычно не пользуются счетчиком ссылок для управления временем жизни объекта, и это невзирая на то, что имеется две схожих ситуации, с которыми сталкиваются программисты: в мире программирования — это управление объектами GDI, а в мире людей — это связь типа телеконференции.

Объекты для рисования в GDI — перо, кисть, шрифт и т. д. — должны явно создаваться и явно удаляться. (Даже несмотря на то, что Windows 95 и Windows NT сами удаляют такие объекты при завершении процесса, хорошим тоном программирования по-прежнему считается освобождение за собой ресурсов.) То есть, любой вызов функции GDI для создания зеленого пера, требует соответствующего вызова функции для удаления этого пера:

```
hpenGreen = CreatePen(PS_SOLID, 1, RGB(0, 255, 0));
…
DeleteObject(hpenGreen);
```

У объектов GDI значение счетчика ссылок никогда не превышает 1.

Недостатком такого подхода является то, что две программы — скажем *PageMaker* и *Visio* — могли бы создать совершенно одинаковые зеленые перья. Или разные подсистемы каждой программы — интерфейс пользователя, поддержка печати, поддержка битовых образов и поддержка метафайлов — могли бы создать собственные зеленые перья. Всего бы было создано десять таких перьев, что ведет к бесполезной трате системных ресурсов.

Однако, если бы в GDI использовался основанный на счетчике ссылок механизм для отслеживания создания общесистемного пера, второй и последующие запросы на создание зеленых перьев привели бы просто к увеличению значения счетчика ссылок на единицу при каждом таком запросе. Выгода состоит в уменьшении необходимых системных ресурсов, поскольку, в конечном итоге, одно зеленое перо может выполнить работу десяти подобных перьев. Если бы перья в GDI были бы реализованы как компоненты OLE, создание всех перьев, кроме первого, свелось бы к следующему вызову компонента зеленое перо:

```
pPenGreen-> AddRef();
```

Каждый такой вызов увеличивал бы на 1 значение счетчика ссылок на перо. Каждый вызов функции *DeleteObject* приводил бы к вызову функции *Release* и, соответственно, к уменьшению на 1 значения счетчика ссылок:

```
pPenGreen -> Release();
```

Это бы продолжалось до тех пор, пока счетчик ссылок не обнулился, и следовательно, зеленое перо могло бы быть удалено. Управление временем жизни компонента, основанное на счетчике ссылок, позволяет каждому клиенту компонента иметь определенные начало и окончание своего контакта с этим компонентом. (Заметим, очень хорошо, что GDI в Windows 95 для уменьшения количества лишних объектов использует механизм счетчика ссылок, хотя и основанный не на интерфейсах OLE.)

Счетчик ссылок отражает независимое соединение с компонентом. Это не слишком отличается от ситуации, когда вы звоните через систему телеконференции вашей компании президенту той компании, которая является главным покупателем ваших программ. Может быть вы (в качестве ответственного за технологию в вашей компании) связываетесь с ним для ответа на какой-то вопрос. С этой точки зрения, такой вызов делает возможным установить в 1 значение счетчика ссылок.

Если разговор заходит о методах испытаний, вы могли бы обратиться к менеджеру по контролю — вашему коллеге из соседней комнаты — с просьбой подключиться к разговору. Когда он поднимает трубку и говорит "Hello", что напоминает вызов функции *AddRef*, значение счетчика ссылок телеконференции увеличивается до 2. Когда разговор переходит на тему выпуска документации, вы могли бы попросить менеджера по документации поднять трубку и присоединиться к телеконференции. Когда он говорит "Hello", значение счетчика ссылок увеличивается до 3. Ваш партнер знает, что трое служащих компании — поставщика программного обеспечения, находятся на связи. Только совершенно невероятная ситуация могла бы привести к тому, что президент вдруг бросит трубку, это было бы аналогично фатальной ошибке сервера OLE. Иногда, ваш покупатель удовлетворяется тем, что сообщили ему ваши коллеги, и просит соединить его с вашим менеджером по продажам, чтобы сделать свой заказ. Когда менеджер по продажам подключается к разговору, его "Hello" поднимает значения счетчика ссылок телеконференции до 4.

Когда вы, менеджер по контролю, и менеджер по документации будете заканчивать разговор, то каждый из вас скажет "Good-bye" и повесит трубку. Каждое прощание — это аналог вызова функции *Release*. Однако, телеконференция продолжается, что эквивалентно компоненту, продолжающему оставаться в памяти. В конце концов, когда менеджер по реализации добьется получения выгодного заказа, он говорит "Good-bye" покупателю и вешает трубку. Теперь значение счетчика ссылок телеконференции равно нулю, и ваш покупатель может повесить трубку.

Будучи концептуально простым, механизм подсчета ссылок требует времени и усилий. Прекращение работы и освобождение ресурсов в нужное время требует, чтобы клиенты интерфейса правильно учитывали все соединения. Когда клиент интерфейса оказывается не в состоянии правильно управлять временем жизни компонента, могут

возникнуть неожиданные результаты: неудача при отсоединении оставляет компонент в памяти; слишком раннее отсоединение оставляет одну из сторон в неопределенном состоянии.

Управление временем жизни компонента — это только одна из задач, выполняемая функциями — членами класса *IUnknown*. Другая важная задача, а именно поддержка нескольких интерфейсов в одном компоненте, является темой следующего рассмотрения.

Поддержка нескольких интерфейсов

Архитектура модели составного объекта OLE позволяет отдельным компонентам обеспечивать существование более одного интерфейса. На бинарном уровне это означает более одного массива указателей функций. Главным преимуществом нескольких интерфейсов является возможность предлагать более широкий диапазон услуг, чем это было бы возможно в случае одного интерфейса. В конечном итоге, интерфейсы представляют из себя контракты на услуги, следовательно, возможность иметь более одного интерфейса — это просто возможность обеспечить более одного типа услуг. Основным механизмом получения указателей на интерфейсы компонента является функция *QueryInterface*.

Поскольку функция *QueryInterface* сама по себе является функцией интерфейса, ее нельзя вызвать для выборки из компонента указателя первого интерфейса. Однако, она может использоваться для получения из компонента указателей на второй и последующие интерфейсы. Указатель на первый интерфейс из компонента почти всегда является возвращаемым значением функций C (не функций интерфейса). Для начала, пример того, как функция *CoGetMalloc* выбирает указатель на интерфейс *IMalloc* для компонента библиотеки OLE — процедуры выделения памяти для задачи:

```
HRESULT       hr;
IMalloc       *pMalloc;

// Соединение с процедурой выделения памяти
hr = CoGetMalloc(1, &pMalloc);
```

После того, как получен первый указатель интерфейса, последующие указатели интерфейсов получают с помощью вызова функции *QueryInterface*, члена любого из существующих интерфейсов. Функция *QueryInterface* объявляется следующим образом:

```
HRESULT QueryInterface(REFIID iid, void **ppv);
```

- *iid* является указателем на идентификатор интерфейса (IID).
- *ppv* указывает на положение в памяти возвращаемого значения функции. Возвращаемым значением для поддерживаемого интерфейса является указатель интерфейса запрашиваемого типа. Запрос к неподдерживаемому интерфейсу приводит к тому, что возвращаемым значением становится NULL.

При просмотре включаемых файлов Win32, обнаруживается, что идентификатор интерфейса (IID) определяется, как GUID (уникальный глобальный идентификатор, globally unique identifier), который сам определяется в WTYPES.H, включаемом файле Win32, следующим образом:

```
typedef struct _GUID
{
       DWORD  Data1;
       WORD   Data2;
       WORD   Data3;
       BYTE   Data4 [8];
} GUID;
```

Это основное 16-байтовое значение уникально определяющее интерфейс.

Если предположить, что компонент OLE — это учреждение, то тогда про идентификатор интерфейса можно было бы сказать, что это добавочный телефонный номер того подразделения, с которым желательно поговорить. В таком случае, вызов функции *QueryInterface* — это запрос на связь с новым подразделением внутри одной и той же элементизации (компонента), таким же подразделением, с каким вы уже говорите. Имена интерфейсов похожи на названия подразделений — расчетный, кадровый, юридический, развития, контроля и т. д. Далее представлена маленькая часть телефонного справочника, в котором перечисляются стандартные добавочные номера для подразделений, упомянутых в компонентах OLE:

Имя интерфейса	Идентификатор интерфейса
IUnknown	{00000000-0000-0000-C000-000000000046}
IClassFactory	{00000001-0000-0000-C000-000000000046}
IMalloc	{00000002-0000-0000-C000-000000000046}
IMarshal	{00000003-0000-0000-C000-000000000046}

Важно запомнить, что имя, которое легко прочитать человеку (IUnknown), дано исключительно для удобства, но клиенты и серверы OLE для того, чтобы отличить один интерфейс от другого пользуются двоичными цифрами. К счастью, вам нечасто придется иметь дело с такими длинными последовательностями чисел, поскольку для всех идентификаторов интерфейсов имеются символьные константы. В символьных именах к имени интерфейса добавляется префикс "IID_", таким образом IID_IUnknown относится к бинарному интерфейсу для интерфейса *IUnknown*. Например, клиент, имеющий указатель на компонент интерфейса *IMalloc*, для получения указателя на интерфейс *IMarshal* сделал бы такой вызов:

```
LPMARSHAL pMarshal;

HRESULT hr = pMalloc -> QueryInterface(IID_IMarshal, &pMarshal);
```

Запомните, что этот вызов является первым запросом: "Поддерживаете ли вы интерфейс *Imarshal*?". Ответ либо "Да" (HRESULT равен S_OK), либо "Нет" (HRESULT равен E_NOINTERFACE). Смысл здесь в том, что имеются тысячи интерфейсов, и нет компонента, который бы все их поддерживал.

Если нужный интерфейс поддерживается, возвращаемое значение типа HRESULT равно S_OK. При удачном запросе имеют место две вещи: возвращаемый указатель заносится в память по адресу, на который ссылается второй параметр (в данном примере *pMarshal*), и значение счетчика ссылок компонента увеличивается на 1. Увеличение счетчика ссылок отражает тот факт, что создано новое соединение с компонентом. Для закрытия соединения требуется последующий вызов функции *Release* ():

```
pMarshal -> Release();
```

Если, с другой стороны, запрашиваемый интерфейс не поддерживается, то возвращаемым значением функции *QueryInterface* становится E_NOINTERFACE. В этом случае, указатель не возвращается, и счетчик ссылок не увеличивается. Однако, это будет означать, что функция *QueryInterface* перезаписывает возвращаемое значение указателя — в нашем случае *pMarshal* — значением NULL.

Одним из важных достоинств такой возможности запроса является то, что клиенты компонента OLE могут выполнить запрос о возможностях компонента во время выполнения программы. Наличие интерфейса означает существование соответствующих свойств, а его отсутствие означает, что таких свойств нет.

Благодаря этому достоинству относительно просто модернизировать свойства компонента OLE. Однако, новые свойства никогда не добавляются путем расширения функций интерфейса; после того, как интерфейс определен, число его свойств изменить нельзя! Вместо этого новые свойства добавляются путем добавления новых интерфейсов. Существующие интерфейсы остаются неизменными, продолжая поддерживать тех клиентов, которые ожидают их присутствия.

Является ли OLE спецификацией клиент/сервер?

Одним из вопросов, которые задают программисты, начинающие работать с OLE, это вопрос о том, является ли OLE спецификацией клиент/сервер. Вспоминая главу 17, можно сказать, что спецификацией клиент/сервер является DDE. OLE версии 1.0 использовал DDE в качестве транспортного механизма. В этой версии сообщения DDE передавали запросы от клиента OLE к серверу OLE, а сервер OLE отвечал новыми сообщениями DDE, которые ссылались на определенную DDE глобальную (*GlobalAlloc*) память.

Хотя термины "клиент" и "сервер" часто использовались в контексте OLE версии 1.0, эволюция OLE привела к использованию их только в общем смысле. Когда два компонента взаимодействуют, часто оба одновременно являются как клиентами (пользователями интерфейса), так и серверами (поставщиками интерфейса). В частности, в контексте составных документов, термин "клиент" был заменен на термин "приложение-контейнер". В контексте автоматизации OLE, вместо термина "клиент автоматизации" используется термин "контроллер автоматизации". И, хотя термин "сервер" все еще используется, контекст поставляемой услуги определяет уточняющее дополнение — "сервер объекта" или "сервер автоматизации".

Термины "клиент" и "сервер" полезны при попытке понять конкретный интерфейс OLE. В частности, эти термины обеспечивают удобную базу для ответа на общий вопрос, который часто возникает при написании программ, использующих OLE: какие интерфейсы должны быть реализованы, а какие интерфейсы реализуются кем-либо еще?

Сервер закрытого компонента

Настало время рассмотреть программу IMALLOC, динамически подключаемую библиотеку, которая демонстрирует то, как реализовать интерфейс. Чтобы сосредоточиться на интерфейсе как таковом, в этом примере оставлено только несколько простых моментов. Во-первых, интерфейс управления памятью OLE *IMalloc* был выбран потому, что большинство программистов уже хорошо понимает услуги по выделению памяти. Во-вторых, библиотека IMALLOC очень мало реально работает с памятью, поскольку она передает заботы по управлению памятью закрытым функциям работы с "кучей" Win32 (*HeapCreate, HeapAlloc* и т. д.). И наконец, мы начинаем с

закрытого компонента (private component) — доступного только через соответствующий механизм защиты — поскольку, как станет ясно при изучении программы PUBMEM, создание открытого компонента (public component) приводит к дополнительной сложности, без которой сейчас вполне можно обойтись.

Хотя IMALLOC является примером простого компонента, это не означает его тривиальность. В частности, он показывает элементы, которые необходимы при определении интерфейса OLE. Определение интерфейса включает в себя реализацию трех функций интерфейса *IUnknown*: *QueryInterface, AddRef* и *Release*. В программе используется специальный макрос, который создает переносимые объявления интерфейса для поддержки C и C++ в операционных системах различных платформ, включая Microsoft Windows 95, Microsoft Windows NT, Apple Macintosh и другие системы, в которых будет включена поддержка OLE.

Закрытые компоненты — это такие компоненты OLE, которые видимы только закрытыми средствами. Например, IMALLOC обеспечивает доступ к своему интерфейсу через закрытую, экспортируемую точку входа. Этот пример является динамически подключаемой библиотекой, но при другом использовании закрытых компонентов он мог бы быть целым приложением со множеством закрытых COM-компонентов. Ключевое преимущество здесь состоит в инкапсуляции, которая достигается благодаря четкому разграничению компонента-клиента и компонента-сервера. Таким образом, компонент OLE определяется созданием недоступных открыто интерфейсов, поддерживающих спецификации OLE.

После внимательного изучения исходных файлов программы IMALLOC, вы могли бы сказать, что в конечном итоге IMALLOC не является компонентом OLE, поскольку он не делает ни одного вызова функций библиотек OLE. Удивительно, почему IMALLOC является компонентом OLE, если нет ни одного вызова функции *OleInitialize* (или *CoInitialize*)? Все просто. Интерфейс IMALLOC создавался в соответствии со спецификацией модели составного объекта, являющейся стандартом OLE. Поскольку услуги библиотек OLE не требуются, IMALLOC обходится без инициализации библиотек OLE, и без загрузки их в память. Но IMALLOC обеспечивает стандартный интерфейс OLE, что станет ясно, когда мы добавим этот интерфейс к образцу открытого компонента PUBMEM, представленного далее в этой главе. Для программы интерфейса IMALLOC почти не потребуются изменения для нормальной работы в открытом компоненте OLE.

Интерфейс IMALLOC выполнен на C++, что имеет смысл для реализации интерфейса, поскольку бинарная модель интерфейса OLE точно соответствует модели объекта C++. Хотя можно реализовать интерфейс и на C, для этого нужно делать кое-какие утомительные вещи, которые автоматически делает компилятор C++. Чтобы понять, как будет выглядеть программа OLE на C, посмотрите на программу CALLER, которая представлена далее в этой главе. Это программа-клиент, которая загружает и вызывает библиотеку IMALLOC.DLL. На рис. 20.2 представлены файлы программы IMALLOC.

IMALLOC.MAK

```
#-----------------------
# IMALLOC.MAK make file
#-----------------------

imalloc.dll : imalloc.obj
     $(LINKER) $(DLLFLAGS) -OUT:imalloc.dll imalloc.obj $(GUILIBS) uuid.lib

imalloc.obj : imalloc.cpp
     $(CC) $(CFLAGS) imalloc.cpp
```

IMALLOC.CPP

```
/*-----------------------------------------
   IMALLOC.CPP -- Define an imalloc interface
                (c) Paul Yao, 1996
  -----------------------------------------*/
#include <windows.h>
#include "imalloc.h"

//-------------------------------------------------------
// CreateAllocator -- Exported function to create allocator
//-------------------------------------------------------
EXPORT LPMALLOC CreateAllocator()
     {
     DAlloc *pAllocator = new DAlloc();
     if(pAllocator != NULL && pAllocator->Initialize())
          {
          pAllocator->AddRef();
          }
     else
```

```
          {
          delete pAllocator;
          }

     return(LPMALLOC) pAllocator;
     }
//------------------------------------------------------------------
DAlloc::DAlloc()
     {
     RefCount = 0;
     hHeap = NULL;
     }

//------------------------------------------------------------------
DAlloc::~DAlloc()
     {
     if(hHeap)
          HeapDestroy(hHeap);
     }

//------------------------------------------------------------------
BOOL DAlloc::Initialize()
     {
     hHeap = HeapCreate(0, 4096, 65535);

     return(BOOL)hHeap;
     }

//------------------------------------------------------------------
STDMETHODIMP
DAlloc::QueryInterface(REFIID riid, LPVOID FAR *ppvObject)
     {
     // Always initialize "out" parameters to NULL
     *ppvObject = NULL;

     // Everyone supports IUnknown
     if(riid == IID_IUnknown)
          *ppvObject =(LPUNKNOWN) this;

     // We support IMalloc
     if(riid == IID_IMalloc)
          *ppvObject =(LPMALLOC) this;

     if(*ppvObject == NULL)
          {
          // Interface not supported
          return E_NOINTERFACE;
          }
     else
          {
          // Interface supported, so increment reference count
         ((LPUNKNOWN) *ppvObject)->AddRef();
          return S_OK;
          }
     }
//------------------------------------------------------------------
STDMETHODIMP_(ULONG)
DAlloc::AddRef(void)
     {
     return ++RefCount;
     }

//------------------------------------------------------------------
STDMETHODIMP_(ULONG)
DAlloc::Release(void)
```

```
     {
     if(0L != --RefCount)
          return RefCount;

     delete this;
     return 0L;
     }

//------------------------------------------------------------------
STDMETHODIMP_(void *)
DAlloc::Alloc(ULONG cb)
     {
     return HeapAlloc(hHeap, HEAP_ZERO_MEMORY, cb);
     }

//------------------------------------------------------------------
STDMETHODIMP_(void *)
DAlloc::Realloc(void *pv, ULONG cb)
     {
     return HeapReAlloc(hHeap, HEAP_ZERO_MEMORY, pv, cb);
     }

//------------------------------------------------------------------
STDMETHODIMP_(void)
DAlloc::Free(void *pv)
     {
     HeapFree(hHeap, 0, pv);
     }

//------------------------------------------------------------------
STDMETHODIMP_(ULONG)
DAlloc::GetSize(void *pv)
     {
     return HeapSize(hHeap, 0, pv);
     }

//------------------------------------------------------------------
STDMETHODIMP_(int)
DAlloc::DidAlloc(void *pv)
     {
     PROCESS_HEAP_ENTRY phe;
     ZeroMemory(&phe, sizeof(PROCESS_HEAP_ENTRY));

     while(HeapWalk(hHeap, &phe))
          {
          if(phe.lpData == pv)
               return 1;
          }

     return 0;
     }

//------------------------------------------------------------------
STDMETHODIMP_(void)
DAlloc::HeapMinimize(void)
     {
     HeapCompact(hHeap, 0);
     }
```

IMALLOC.H

```
//------------------------------------------------------------------
// C Interface to private allocator
//------------------------------------------------------------------
#define EXPORT extern "C" __declspec(dllexport)
```

```
EXPORT LPMALLOC CreateAllocator();

//--------------------------------------------------------------------
// Implementation of allocator interface
//--------------------------------------------------------------------
#undef  INTERFACE
#define INTERFACE DAlloc

DECLARE_INTERFACE_(DAlloc, IMalloc)
     {
     // *** IUnknown methods ***
     STDMETHOD (QueryInterface)(THIS_ REFIID riid, LPVOID FAR *ppv);
     STDMETHOD_(ULONG, AddRef) (THIS);
     STDMETHOD_(ULONG, Release)(THIS);

     // *** IMalloc methods ***
     STDMETHOD_(void *, Alloc)       (THIS_ ULONG cb);
     STDMETHOD_(void *, Realloc)     (THIS_ void *pv, ULONG cb);
     STDMETHOD_(void,   Free)        (THIS_ void *pv);
     STDMETHOD_(ULONG,  GetSize)     (THIS_ void *pv);
     STDMETHOD_(int,    DidAlloc)    (THIS_ void *pv);
     STDMETHOD_(void,   HeapMinimize)(THIS);

#ifndef CINTERFACE
public :
     DAlloc();
     ~DAlloc();
     BOOL Initialize();

private :
     ULONG  RefCount;
     HANDLE hHeap;
#endif
     };
```

Рис. 20.2 Библиотека IMALLOC

Для того, чтобы увидеть эту процедуру выделения памяти в действии, запустите файл CALLER.EXE, который загружает и использует библиотеку IMALLOC.DLL.

IMALLOC.DLL

Как закрытый компонент OLE библиотека IMALLOC демонстрирует свой интерфейс *IMalloc* посредством указателя интерфейса, являющегося возвращаемым значением закрытой экспортируемой функции *CreateAllocator*. Сами функции интерфейса экпортировать не нужно, поскольку возвращаемый указатель интерфейса полностью обеспечивает доступ ко всем его функциям. Функция *CreateAllocator* выделяет память для экземпляра *DAlloc*, объекта C++, который помещается в защищенную "кучу" Win32 , и чья таблица виртуальных функций соответствует двоичному стандарту, определенному для интерфейса *IMalloc*. Указатель на объект *DAlloc* — это указатель на интерфейс *IMalloc*. Такое преобразование типа осуществляется в функции *CreateAllocator*:

```
return(LPMALLOC) pAllocator;
```

Поскольку функция *CreateAllocator* определяется в файле с исходным текстом на C++, требуется специальная обработка, чтобы запретить искажение имен, вносимое C++, и обеспечить возможность вызова этой функции из C. Эту задачу выполняют следующие строки из файла IMALLOC.H:

```
#define EXPORT extern "C" __declspec(dllexport)


EXPORT LPMALLOC CreateAllocator();
```

Ключевое слово EXPORT мы определили в главе 19.

Замечание для программистов на языке C

Библиотека IMALLOC написана на C++, поэтому некоторые термины и соглашения могут быть вам незнакомы. Внутри функции *CreateAllocator* оператор *new* выделяет область памяти для объекта *DAlloc* (путем вызова функции *malloc*), а затем вызывается его конструктор *DAlloc::DAlloc*, который выполняет инициализацию данных

— членов объекта. Оператор *delete* вызывается из функции *CreateAllocator*, если инициализация не прошла, и освобождает память объекта в случае ошибки. Это функция аналогична вызову функции *free*, за которой следует вызов функции-деструктора для удаления (или очистки) объекта *DAlloc::~DAlloc*.

Замечание для программистов на языке C

Оператор разрешения области видимости (::) указывает на принадлежность функции конкретному классу C++. В IMALLOC такое обозначение показывает, что функция *GetSize* является членом класса *DAlloc*:

DAlloc::GetSize (...)

Хотя все функции, перед именами которых стоит *DAlloc*::, являются членами класса *DAlloc*, не все функции-члены входят в таблицу функций класса *DAlloc*, а только те, которые объявлены с ключевым словом *virtual*. В файле IMALLOC.H макросы STDMETHOD и STDMETHOD_ (о чем подробно будет рассказано позже) содержат ключевое слово *virtual*.

Отличие между *IMalloc* и *DAlloc* в порядке их реализации. Как контракт на услуги, интерфейс не может иметь реализацию, непосредственно ассоциированную с ним. В терминах C++ это требование удовлетворяется тем, что все интерфейсы делаются абстрактными классами, то есть чистыми интерфейсами безо всякой реализации. Следовательно *DAlloc* (префикс "D" означает "производный класс, derived class") использует *IMalloc* в качестве базового класса, от которого он наследует свои виртуальные функции-члены.

Но, учитывая наличие трех дополнительных функций-членов — *DAlloc, ~DAlloc* и *Initialize* — и двух дополнительных данных-членов — *RefCount* и *hHeap* — является ли *DAlloc* правильной реализацией *IMalloc*? Ответ — да, является. Три дополнительные функции не являются виртуальными (они не объявлены с ключевым словом *virtual*), следовательно, они не влияют на таблицу виртуальных функций, которая в терминах OLE представляет собой таблицу функций интерфейса. Поскольку в *DAlloc* имеется таблица функций интерфейса, которая соответствует таблице, необходимой для выполнения того, что определяет *IMalloc*, то эти три дополнительные функции не играют роли. Как и данные-члены, они игнорируются, поскольку интерфейсы OLE определяются в C++ как классы без данных (data-free). *DAlloc* может использовать эти данные-члены для собственных целей, поскольку они не участвуют в выполнении контракта, что необходимо для интерфейса OLE в общем и для интерфейса *IMalloc* в частности.

Теперь о макросах

Одним из аспектов определения интерфейса, который поначалу может вызвать путаницу, является использование ряда макросов типа DECLARE_INTERFACE, STDMETHOD и THIS. Смысл этих макросов состоит в том, чтобы сделать коды OLE переносимыми, как для клиента, так и для сервера. Эти макросы способствуют улучшению переносимости интерфейсов OLE двумя путями: во-первых, макросы определены и на C, и на C++, что обеспечивает доступ к интерфейсу на обоих языках. Во-вторых, макросы помогают поддерживать переносимость для операционных систем разных платформ.

Первые компиляторы C++ появились как раз в то самое время, когда зарождался OLE. Тогда шире, чем C++ использовался язык C, поэтому создатели OLE пытались обеспечить возможность работы с любым из этих языков. Идею того, как эти макросы скрывают отличия C и C++ поможет понять пример. При использовании макросов интерфейс *IUnknown* объявляется следующим образом:

```
#undef INTERFACE
#define INTERFACE IUnknown

DECLARE_INTERFACE(IUnknown)
{
       STDMETHOD(QueryInterface)(THIS_ REFIID riid, LPVOID FAR *ppvObj) PURE;

       STDMETHOD_(ULONG, AddRef)(THIS) PURE;

       STDMETHOD_(ULONG, Release)(THIS) PURE;
};
```

Многие макросы имеют две версии определения интерфейса: обычная версия и расширенная версия, в именах макросов которой имеется суффикс _. В объявлении интерфейса *IUnknown* имеются примеры обеих версий: STDMETHOD и STDMETHOD_. В обеих версиях функции интерфейса объявляются *виртуальными*, и обе используют специальное соглашения для вызова функций *(__stdcall* в Win32). Но STDMETHOD означает, что типом возвращаемого значения является HRESULT. Другие же типы возвращаемого значения должны быть заданы в качестве параметров макроса STDMETHOD_.

В следующей таблице суммируется роль объявления интерфейса и определения макросов:

Имя макроса	Описание

Имя макроса	Описание
DECLARE_INTERFACE	Объявляет интерфейс без базового класса (как например в объявлении интерфейса *IUnknown*)
DECLARE_INTERFACE_	Объявляет интерфейс с базовым классом (например, *DAlloc* в IMALLOC.CPP)
PURE	В C++ добавляет "= 0" в конце объявления виртуальной функции для чистой (без реализации) виртуальной функции. В C нет эквивалента, поэтому является пустой командой.
STDMETHOD	Объявляет об использовании соглашения о вызове функций *(__stdcall* в Win32) и о том, что тип возвращаемого значения — HRESULT. При компиляции эти функции, так же как и виртуальные, включаются в таблицу виртуальных функций классов C++ и функций OLE интерфейса.
STDMETHOD_	То же что и STDMETHOD, за исключением того, что тип возвращаемого значения не HRESULT.
STDMETHODIMP	Определяет функцию-член, тип возвращаемого значения которой HRESULT.
STDMETHODIMP_	Определяет функцию-член, тип возвращаемого значения которой не HRESULT.
THIS	Метка-заполнитель для функций интерфейса без параметров. В C++ THIS заменяется на *void*. В C THIS заменяется указателем на объект, что обеспечивает правильный связующий вызов функции-члена класса C++ из C.
THIS_	Метка-заполнитель для функций интерфейса с одним и более параметров. В C++ THIS_ является пустым. В C заменяется указателем, о чем говорилось выше для THIS, за которым следует запятая.

Когда предыдущее объявление интерфейса *IUnknown* обрабатывается препроцессором в качестве объявления C++, формируется следующее объявление:

```
struct IUnknown
{
	virtual HRESULT __stdcall QueryInterface(const IID *riid, void **ppv) = 0;
	virtual ULONG __stdcall AddRef(void) = 0;
	virtual ULONG __stdcall Release(void) = 0;
};
```

Можно было бы ожидать, что в начале этого объявления окажется *class*, а не *struct*. Отличие структуры от класса состоит в том, что по умолчанию члены структуры являются открытыми. Ключевое слово *__stdcall* определяет заполнение и очистку фрейма стека. Ключевое слово *virtual* определяет виртуальную функцию, что необходимо для включения функции в таблицу виртуальных функций C++, и тождественно включено в таблицу функций интерфейса OLE. И наконец "= 0" заменяет ключевое слово PURE, что делает эту структуру абстрактным классом, то есть классом, имеющим только интерфейс. Объекты этого класса существовать не могут.

Итоговое объявление интерфейса *IUnknown* на C несколько сложнее, поскольку создаются две структуры. Несмотря на это отличие от программы на C++ в двоичный код обе программы компилируются одинаково. Так происходит, поскольку модель объекта компонента OLE определяет бинарный стандарт для интерфейсов. Далее приводится соответствующие коды интерфейса *IUnknown* на C:

```
typedef struct IUnknown
{
	const struct IUnknownVtbl *lpVtbl;
} IUnknown;
typedef const struct IUnknownVtbl IUnknownVtbl;
const struct IUnknownVtbl
{
	HRESULT(__stdcall * QueryInterface)(IUnknown *This, const IID *riid, void **ppv);
	ULONG(__stdcall * AddRef)(IUnknown *This);
	ULONG(__stdcall * Release)(IUnknown *This);
};
```

При реализации на C в начало списка параметров каждой функции добавляется параметр *IUnknown *This*. Это соответствует указателю *this*, который в C++ всегда неявно передается функциям — членам класса. Хотя это может казаться странным, но такая добавка необходима для программы на C, чтобы соответствовать реализации интерфейса на C++. В файлах с исходным текстом программы CALLER.EXE, нашего следующего примера программы, которая вызывает библиотеку IMALLOC.DLL, показано, как должен задаваться этот дополнительный параметр, когда программа на C вызывает функцию класса C++.

Переносимость между платформами означает наличие запаса специфических для платформы ключевых слов. Например, для интерфейсов OLE Win16 при компоновке требуются ключевые слова *__far* и *__pascal*. В Win32 эти два слова заменены на *__stdcall*, как соглашение для вызова всех функций интерфейсов. Макросы обеспечивают метку-заполнитель для ключевых слов, которые нельзя использовать при создании, поскольку, фактически, компоненты OLE могут создаваться для работы в системе Apple Macintosh, реализациях OLE в UNIX или в OS/400, VMS или MSV. Специфический для платформы набор включаемых файлов будет определять специфические для среды ключевые слова там, где они необходимы.

Услуги, предоставляемые интерфейсом *IUnknown*

Хотя в библиотеке IMALLOC.DLL показана реализация интерфейса *IMalloc*, более интересной ее чертой является реализация трех функций — членов *IUnknown* и двух услуг, обеспечиваемых этими функциями: управлением временем жизни компонента и поддержкой многоинтерфейсного компонента.

Управление временем жизни компонента

Жизнь компонента OLE определяется одним закрытым членом данных и двумя функциями интерфейса. Членом данных является счетчик ссылок — значение типа ULONG, названное в *DAlloc RefCount*, а двумя функциями интерфейса — *AddRef* и *Release*.

Библиотека IMALLOC создает компонент — процедуру выделения памяти, когда вызывается функция *CreateAllocator*, функция создания закрытого компонента. Ниже приведены соответствующие строки программы внутри функции *CreateAllocator*:

```
DAlloc *pAllocator = new DAlloc();

if(pAllocator != NULL && pAllocator -> Initialize( ))
{
       pAllocator -> AddRef();
}
else
{
       delete pAllocator;
}
```

Когда создается объект, в конструкторе *DAlloc* его счетчик ссылок инициализируется нулем. В то же самое время описатель "кучи" Win32 устанавливается в NULL:

```
DAlloc::DAlloc( )
{
       RefCount = 0;
       hHeap = NULL;
}
```

В C++ программирование упрощается благодаря тому, что работа конструкторов класса, как только что показано, ограничивается только инициализацией данных-членов. Смысл в том, что конструкторам непросто известить функцию-создателя объекта о неблагоприятных условиях, следовательно они выполняют только то, что не может не получиться. Те задачи, которые могут потерпеть неудачу, часто передаются специальным функциям инициализации, таким как функция *Initialize* в *DAlloc*, которая создает "кучу" Win32, необходимую для удовлетворения запросов на выделение памяти интерфейса *IMalloc*.

После того, как функция *CreateAllocator* должным образом создала и инициализировала объект *DAlloc*, она с помощью функции *AddRef* увеличивает на 1 значение счетчика ссылок. Эта функция определяется следующим образом:

```
STDMETHODIMP_(ULONG)

DAlloc::AddRef(void)
{
       return ++RefCount;
}
```

Теперь становится понятно, почему типом возвращаемого значения этой функции не является тип HRESULT: оно позволяет вызывающей функции узнать текущее значение счетчика ссылок (но им можно пользоваться только в целях отладки).

Другой вызов функции *AddRef* в программе IMALLOC имеет место только в функции *QueryInterface*. Когда функция *QueryInterface* возвращает указатель на интерфейс компонента, она вызывает функцию *AddRef* и таким образом поддерживается правильное значение счетчика соединений компонента с внешним миром.

Хотя в программе IMALLOC функция *AddRef* вызывается в двух местах, функция *Release* не вызывается ни разу. Только клиент компонента может вызвать эту функцию, что имеет смысл, поскольку он может заканчивать жизнь компонента. Далее представлены основные строки этой функции:

```
STDMETHODIMP_(ULONG)

DAlloc::Release( )
{
       if(0L != --RefCount) return RefCount;

       delete this;
       return 0L;
}
```

Когда значение счетчика ссылок становится равным нулю, вызов оператора *delete* заставляет компонент освободить самого себя и вызвать деструктор *DAlloc::~DAlloc*. Внутри этой функции удаляется "куча" Win32 :

```
if(hHeap) HeapDestroy(hHeap);
```

Очевидно, что уменьшение счетчика ссылок до нуля — т. е. освобождение объекта — может иметь необратимые последствия.

Хотя члены интерфейса *IUnknown* компонента OLE обеспечивают способы для управления временем жизни компонента, забота клиента-компонента состоит в том, чтобы правильно и в нужное время вызывать эти функции-члены. Слишком малое число вызовов функции *Release* приведет к тому, что компонент понапрасну будет занимать оперативную память. Слишком много вызовов ведут к тому, что объект исчезнет раньше положенного срока. Понятно, что получение правильного значения счетчика ссылок — это важная задача, которую призвана решать программа клиента.

С точки зрения использования функций — членов интерфейса *IUnknown*, следует предложить несколько советов. Во-первых, каждому вызову функции *AddRef* должен соответствовать вызов функции *Release*. А поскольку функция *AddRef* вызывается функцией *QueryInterface*, то каждому вызову функции *QueryInterface* должен соответствовать вызов функции *Release*.

Поддержка многоинтерфейсного компонента

С точки зрения сервера интерфейса OLE, функция *QueryInterface* — это "центральный пульт" компонента для распределения запросов на соединения с интерфейсами. Далее представлена функция *QueryInterface* из файла IMALLOC.CPP:

```
STDMETHODIMP DAlloc::QueryInterface(REFIID riid, LPVOID FAR *ppvObject)
{
       // Параметры "вывода" всегда инициализируются значением  NULL
       *ppvObject = NULL;

       // Все поддерживают IUnknown
       if(riid == IID_IUnknown) *ppvObject =(LPUNKNOWN) this;

       // Мы поддерживаем IMalloc
       if(riid == IID_IMalloc) *ppvObject =(LPMALLOC) this;


       if(*ppvObject == NULL)
       {
              // Интерфейс не поддерживается
              return E_NOINTERFACE;
       }
       else
       {
              // Интерфейс поддерживается, поэтому значение счетчика
              // ссылок увеличивается на 1
              ((LPUNKNOWN) *ppvObject) -> AddRef();
              return S_OK;
       }
}
```

Как и у большинства функций интерфейса OLE, возвращаемое значение функции *QueryInterface* имеет тип HRESULT. В функции определяется два возвращаемых значения, и оба показывают отсутствие неожиданных ошибок: S_OK означает, что компонент поддерживает данный интерфейс, и возвращаемым значением является

указатель на этот интерфейс. E_NOINTERFACE означает, что компонент не поддерживает данный интерфейс, и возвращаемым значением является NULL.

Из-за возможности межпроцессной поддержки, параметры функций OLE и параметры интерфейсов подразделяют на параметры ввода (in), вывода (out) или ввода/вывода (in/out). По отношению к вызываемой функции, параметр "ввода" предназначен для поступающих данных, параметр "вывода" предназначен для выводимых данных, а параметр "ввода/вывода" обеспечивает как ввод, так и вывод данных. Это простое подразделение помогает сократить объем данных, которые необходимо передавать через границы процесса или по сети. Односторонние параметры — "ввод" и "вывод" — нужно передавать только половину заданного времени, тогда как двусторонние параметры — ввода/вывода — должны копироваться и до, и после вызова функции.

В функции *QueryInterface* имеется один параметр "ввода" (*riid*) и один параметр "вывода" (*ppvObject*). Для поддержки целостности спецификация модели составного объекта гласит, что любой параметр вывода, который является указателем, всегда должен быть установлен в известное значение. По этой причине в первой строке функции *QueryInterface* параметр *ppvObject* устанавливается в NULL:

```
// Параметры "вывода" всегда инициализируются значением  NULL
*ppvObject = NULL;
```

Если требуемый интерфейс не поддерживается, то NULL становится правильным значением указателя. Если, с другой стороны, интерфейс поддерживается, то значение NULL заменяется указателем на требуемый интерфейс.

Если интерфейс поддерживается, то функция возвращает указатель на этот интерфейс. Далее, чтобы гарантировать, что счетчик ссылок на этот интерфейс поддерживается правильно, делается вызов функции *AddRef*, которая является функцией-членом интерфейса:

```
((LPUNKNOWN) *ppvObject) -> AddRef( );
```

Это особенно важно реализовать для клиентов компонента OLE, поскольку безошибочный вызов функции *QueryInterface* всегда требует соответствующего вызова функции *Release*, что позволяет компоненту узнать, когда можно безопасно удалить самого себя.

Несмотря на то, что библиотека IMALLOC.DLL поддерживает только один интерфейс, фактически функция *QueryInterface* реагирует так, как будто поддерживаются два интерфейса: *IUnknown* и *IMalloc*. С точки зрения клиента, действительно поддерживаются два интерфейса: один, который управляет основным компонентом и другой, который реализует услуги по выделению памяти. Факт, что оба используют одну и ту же таблицу виртуальных функций определяется проектом, который соответствует архитектуре модели составного объекта.

Не слишком трудно представить, как функция *QueryInterface* могла бы обеспечивать соединения с дополнительными интерфейсами. Простые приемы того, как функция *QueryInterface* могла бы это делать, лежат на поверхности. Добавление инструкции *if* для проверки конкретных значений идентификаторов интерфейсов позволяет поддерживать любое число новых интерфейсов, например, такого как интерфейс с идентификатором IID_IMarshal:

```
// Мы поддерживаем IMarshal
if(riid == IID_IMarshal) *ppvObject =(LPMARSHAL) pMarshal;
```

Не вполне очевидной является связь интерфейса, на который ссылается указатель *pMarshal*, с заданной функцией *DAlloc::QueryInterface*. На это имеется несколько объяснений, которые к сожалению выходят за пределы нашей дискуссии. А вкратце, ответ таков: объект C++, который поддерживает интерфейс *IMarshal* — назовем его *DMarshal* — мог бы содержаться в объекте *DAlloc*. Или так: благодаря множественному наследованию, несколько поддерживаемых интерфейсов могли бы разделять общий набор функций интерфейса *IUnknown*. Какая бы конкретная реализация ни была выбрана, клиент компонента видит одно и то же: функция *QueryInterface* обеспечивает услуги как бы центрального пульта для создания соединений с поддерживаемыми интерфейсами компонента.

Чтобы узнать, как клиент мог бы использовать закрытый компонент интерфейса IMALLOC, рассмотрим далее программу CALLER. Она также показывает, как можно использовать программу на языке C для доступа к услугам OLE.

Клиент закрытого компонента

Программа CALLER.EXE демонстрирует, как клиент закрытого компонента создает соединение с интерфейсом, вызывает функции интерфейса и отсоединяется от него. Хотя в программе IMALLOC были показаны детали создания интерфейсов с точки зрения сервера, в программе CALLER рассматриваются детали использования интерфейсов с позиции клиента. Из этих двух вариантов проще понять позицию клиента. За исключением действий, связанных с правильным управлением временем жизни компонента, что обеспечивается функциями интерфейса *IUnknown*, позиция клиента близка любому программисту, который когда-либо пользовался библиотекой функций: определение требуемой функции, выбор нужных параметров и проверка возвращаемого значение на успех или неудачу.

В некотором отношении вид интерфейса модели составного объекта отличается от вида других программных интерфейсов, таких как API Win32 Windows или библиотеки времени выполнения C: в то время как другие API поддерживают множество функций, которые связаны друг с другом именами (*strcpy, strlen, strcat* и т. д.) или общим первым параметром (*MoveWindow, FindWindow, ShowWindow* и т. д.), функции интерфейса модели составного объекта определяют четкие границы, отделяющие их от других наборов функций. В сервере объекта этот аспект иллюстрируется тем, что все функции интерфейса являются виртуальными функциями одного вида. В клиенте интерфейса эта близость проявляется благодаря тому факту, что доступ ко всем функциям интерфейса осуществляется через общий указатель на функции интерфейса.

Программа CALLER.EXE также показывает, как получить доступ к интерфейсу модели составного объекта из C. Для этого требуется немного больше усилий, что означает несколько больше символов в программе. Например, ниже представлена программа на C++ для доступа к *Alloc* — члену интерфейса *IMalloc*:

```
LPSTR pData =(LPSTR) pMalloc -> Alloc(cbdata);
```

А аналогичный вызов функции на C выглядит следующим образом:

```
LPSTR pData =(LPSTR) pMalloc -> lpVtbl -> Alloc(pMalloc, cbdata);
```

Это, конечно, не непреодолимое количество дополнительных символов, но оно ведет к дополнительным усилиям и, следовательно, увеличивает риск ошибки.

Очевидным решением было бы спрятать такие вызовы в специальные макросы интерфейса. Это могло бы быть полезно для группы разработки, которая бы планировала дополнить старые приложения на C вызовами функций интерфейсов OLE. Например, для предыдущего вызова функции *IMalloc::Alloc* макрос для его сокрытия мог бы определяться следующим образом:

```
#ifdef __cplusplus

#define ALLOC(pInt, cbSize)(pInt## -> Alloc(cbSize##))
#else

#define ALLOC(pInt, cbSize) \(pInt## -> lpVtbl -> Alloc(pInt##, cbSize##))
#endif
```

Тогда можно было бы использовать один и тот же синтаксис как для вызова функции *IMalloc::Alloc* из C или C++:

```
LPSTR pData =(LPSTR) ALLOC(pMalloc, cbdata);
```

Такой вызов не уменьшает усилий программиста C++ (и возможно больше запутывает, поскольку отражает нестандартный способ доступа к классу C++). Однако для программиста C набор таких макросов исключил бы необходимость дополнительной работы по вводу символов.

В программе CALLER демонстрируются вызовы интерфейсов библиотеки IMALLOC путем выделения памяти и сохранения значений в одной из десяти позиций. Как показано на рис. 20.3, программа CALLER выводит на экран в своем главном окне содержимое каждой позиции. Только для того, чтобы сделать это интереснее в программе CALLER используются две процедуры выделения памяти: доступ к одной осуществляется через интерфейс *IMalloc* из библиотеки IMALLOC.DLL, а к другой из обычной библиотеки функций C. На рис. 20.4 представлены файлы с исходным текстом программы CALLER.

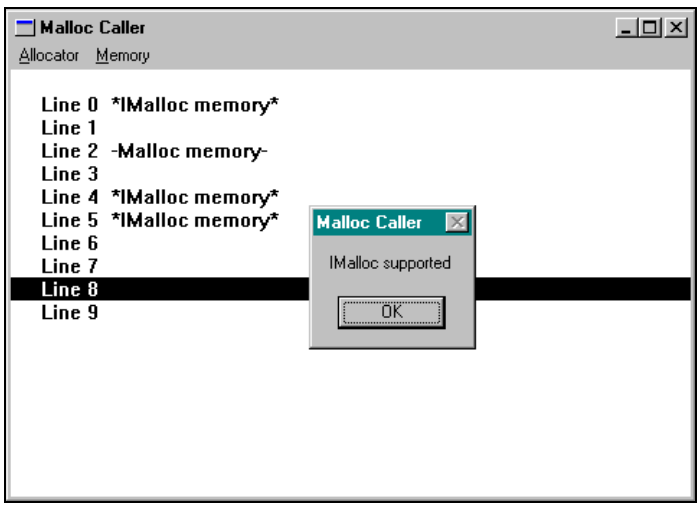


Рис. 20.3 Вид экрана программы CALLER

CALLER.MAK

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

```
# CALLER.MAK make file
#-----------------------

caller.exe : caller.obj caller.res imalloc.lib
     $(LINKER) $(GUIFLAGS) -OUT:caller.exe caller.obj caller.res \
               imalloc.lib $(GUILIBS) uuid.lib

caller.obj : caller.c caller.h imalloc.h
     $(CC) -DCINTERFACE $(CFLAGS) caller.c

caller.res : caller.rc caller.h
     $(RC) $(RCVARS) caller.rc
```

CALLER.C

```
/*-------------------------------------------
   CALLER.C -- Call into private OLE component
               (c) Paul Yao, 1996
  -------------------------------------------*/
#include <windows.h>
#include "caller.h"
#include "imalloc.h"

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

char szWndClass[] = "CallerWindow";
char szAppName[]  = "Malloc Caller";
//---------------------------------------------------------------------
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR lpszCmdLine, int cmdShow)
     {
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wc;

     wc.cbSize        = sizeof(wc);
     wc.lpszClassName = szWndClass;
     wc.hInstance     = hInstance;
     wc.lpfnWndProc   = WndProc;
     wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wc.lpszMenuName  = "MAIN";
     wc.hbrBackground =(HBRUSH)(COLOR_WINDOW + 1);
     wc.style         = 0;
     wc.cbClsExtra    = 0;
     wc.cbWndExtra    = 0;
     wc.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

     RegisterClassEx(&wc);

     hwnd = CreateWindowEx(0L, szWndClass, szAppName,
                           WS_OVERLAPPEDWINDOW,
                           CW_USEDEFAULT, CW_USEDEFAULT,
                           CW_USEDEFAULT, CW_USEDEFAULT,
                           NULL, NULL, hInstance, NULL);

     ShowWindow(hwnd, cmdShow);
     UpdateWindow(hwnd);

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

```
//---------------------------------------------------------------------
LRESULT CALLBACK
WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int       iCurLine = 0;
     static LPMALLOC  pMalloc = NULL;
     static LPSTR     szLine[10];
     static RECT      rHit[10];

     switch(iMsg)
          {
          case WM_CREATE :
               // Initialize data pointer array
               ZeroMemory(szLine, sizeof(szLine));
               return 0;

          case WM_COMMAND :
               switch(LOWORD(wParam))
                    {
                    case IDM_CREATE :
                         pMalloc = CreateAllocator();
                         if(pMalloc == NULL)
                              {
                              MessageBox(hwnd, "Error: No allocator",
                                        szAppName, MB_OK);
                              return 0;
                              }

                         InvalidateRect(hwnd, NULL, TRUE);
                         return 0;

                    case IDM_DESTROY :
                         {
                         int i;

                         // Mark allocated blocks as invalid
                         for(i = 0; i < 10; i++)
                              {
                              if((szLine[i] != NULL) &&
                                 (pMalloc->lpVtbl->DidAlloc(pMalloc,
                                                            szLine[i])))
                                {
                                szLine[i] = NULL;
                                }
                              }

                         // Disconnect from & free allocator
                         pMalloc->lpVtbl->Release(pMalloc);
                         pMalloc = NULL;

                         InvalidateRect(hwnd, NULL, TRUE);
                         return 0;
                         }

                    case IDM_IUNKNOWN :
                         {
                         LPUNKNOWN pUnk;
                         HRESULT hr =
                              pMalloc->lpVtbl->QueryInterface(pMalloc,
                                                              IID_IUnknown,
                                                             (void **) &pUnk);
                         if(SUCCEEDED(hr))
                              {
                              pUnk->lpVtbl->Release(pUnk);
```

```
                MessageBox(hwnd, "IUnknown supported",
                            szAppName, MB_OK);
                }
           else
                {
                MessageBox(hwnd, "IUnknown not supported",
                            szAppName, MB_OK);
                }
           return 0;
           }

      case IDM_IMALLOC :
           {
           LPUNKNOWN pUnk;
           HRESULT hr =
                pMalloc->lpVtbl->QueryInterface(pMalloc,
                                                 IID_IMalloc,
                                                (void **) &pUnk);
           if(SUCCEEDED(hr))
                {
                pUnk->lpVtbl->Release(pUnk);
                MessageBox(hwnd, "IMalloc supported",
                            szAppName, MB_OK);
                }
           else
                {
                MessageBox(hwnd, "IMalloc not supported",
                            szAppName, MB_OK);
                }
           return 0;
           }

      case IDM_IMARSHAL :
           {
           LPUNKNOWN pUnk;
           HRESULT hr =
                pMalloc->lpVtbl->QueryInterface(pMalloc,
                                                 IID_IMarshal,
                                                (void **) &pUnk);
           if(SUCCEEDED(hr))
                {
                pUnk->lpVtbl->Release(pUnk);
                MessageBox(hwnd, "IMarshal supported",
                            szAppName, MB_OK);
                }
           else
                {
                MessageBox(hwnd, "IMarshal not supported",
                            szAppName, MB_OK);
                }
           return 0;
           }

      case IDM_ALLOCATE_CUSTOM :
           if(szLine[iCurLine] != NULL)
                {
                MessageBox(hwnd, "Error: Free First",
                            szAppName, MB_OK);
                return 0;
                }

           // Allocate from IAllocate interface
           szLine[iCurLine] =
               (char *) pMalloc->lpVtbl->Alloc(pMalloc, 100);
           lstrcpy(szLine[iCurLine], "*IMalloc memory*");
```

```
                InvalidateRect(hwnd, NULL, TRUE);
                return 0;

           case IDM_ALLOCATE_DEFAULT :
                if(szLine[iCurLine] != NULL)
                     {
                     MessageBox(hwnd, "Error: Free First",
                                 szAppName, MB_OK);
                     return 0;
                     }

                // Allocate from default heap
                szLine[iCurLine] =(char *) malloc(100);
                lstrcpy(szLine[iCurLine], "-Malloc memory-");

                InvalidateRect(hwnd, NULL, TRUE);
                return 0;

           case IDM_FREE :
                if(szLine[iCurLine] == NULL)
                     {
                     MessageBox(hwnd, "Error: Nothing to free",
                                 szAppName, MB_OK);
                     return 0;
                     }

                if(pMalloc == NULL)
                     {
                     goto FreeMalloc;
                     }

                // Free allocated object
                if(pMalloc->lpVtbl->DidAlloc(pMalloc,
                                          szLine[iCurLine]))
                     {
                     pMalloc->lpVtbl->Free(pMalloc,
                                       szLine[iCurLine]);
                     }
                else
                     {
           FreeMalloc:
                     free(szLine[iCurLine]);
                     }

                szLine[iCurLine] = NULL;

                InvalidateRect(hwnd, NULL, TRUE);
                return 0;
                }

      case WM_DESTROY :
           // Disconnect from & free allocator
           if(pMalloc)
                {
                pMalloc->lpVtbl->Release(pMalloc);
                pMalloc = NULL;
                }

           PostQuitMessage(0);  // Handle application shutdown
           return 0;

      case WM_INITMENU :
           {
           HMENU hMenu =(HMENU) wParam;
```

```
          if(pMalloc)
               {
               EnableMenuItem(hMenu, IDM_CREATE,          MF_GRAYED);
               EnableMenuItem(hMenu, IDM_DESTROY,         MF_ENABLED);
               EnableMenuItem(hMenu, IDM_ALLOCATE_CUSTOM, MF_ENABLED);
               EnableMenuItem(hMenu, IDM_IUNKNOWN,        MF_ENABLED);
               EnableMenuItem(hMenu, IDM_IMALLOC,         MF_ENABLED);
               EnableMenuItem(hMenu, IDM_IMARSHAL,        MF_ENABLED);
               }
          else
               {
               EnableMenuItem(hMenu, IDM_CREATE,          MF_ENABLED);
               EnableMenuItem(hMenu, IDM_DESTROY,         MF_GRAYED);
               EnableMenuItem(hMenu, IDM_ALLOCATE_CUSTOM, MF_GRAYED);
               EnableMenuItem(hMenu, IDM_IUNKNOWN,        MF_GRAYED);
               EnableMenuItem(hMenu, IDM_IMALLOC,         MF_GRAYED);
               EnableMenuItem(hMenu, IDM_IMARSHAL,        MF_GRAYED);
               }
          return 0;
          }

     case WM_LBUTTONDOWN :
          {
          int i;
          int x = LOWORD(lParam);
          int y = HIWORD(lParam);
          POINT pt = { x, y };

          for(i = 0; i < 10; i++)
               {
               if(PtInRect(&rHit[i], pt))
                    {
                    if(iCurLine != i)  // Minimize screen blink
                         {
                         InvalidateRect(hwnd, &rHit[iCurLine], TRUE);
                         InvalidateRect(hwnd, &rHit[i], TRUE);
                         iCurLine = i;
                         }
                    break;
                    }
               }
               return 0;
          }

     case WM_PAINT :
          {
          char         szBuff[10];
          COLORREF     crText, crBack;
          HDC          hdc;
          int          cc;
          int          i;
          int          XCount, XText, Y;
          int          cyLineHeight;
          PAINTSTRUCT  ps;
          RECT         rOpaque;
          TEXTMETRIC   tm;

          hdc = BeginPaint(hwnd, &ps);

          // Fetch line height
          GetTextMetrics(ps.hdc, &tm);
          cyLineHeight = tm.tmHeight + tm.tmExternalLeading;

          // Fetch current text colors
          crText = GetTextColor(ps.hdc);
```

```
               crBack = GetBkColor(ps.hdc);

               XCount = tm.tmAveCharWidth * 3;
               XText  = XCount + tm.tmAveCharWidth * 7;
               Y      = tm.tmHeight;

               for(i = 0; i < 10; i++, Y += cyLineHeight)
                    {
                    // Set colors to highlight current line
                    if(i == iCurLine)
                         {
                         SetTextColor(ps.hdc, crBack);
                         SetBkColor(ps.hdc, crText);

                         SetRect(&rOpaque, 0, Y, 9999, Y + cyLineHeight);
                         ExtTextOut(ps.hdc, 0, 0, ETO_OPAQUE, &rOpaque,
                                    NULL, 0, NULL );
                         }
                    else
                         {
                         SetTextColor(ps.hdc, crText);
                         SetBkColor(ps.hdc, crBack);
                         }

                    // Display line count
                    cc = wsprintf(szBuff, "Line %d", i);
                    TextOut(ps.hdc, XCount, Y, szBuff, cc);

                    // Display text if a string has been defined
                    if(szLine[i] != NULL)
                         {
                         cc = lstrlen(szLine[i]);
                         TextOut(ps.hdc, XText, Y, szLine[i], cc);
                         }

                    // Calculate hit test rectangle
                    SetRect(&rHit[i], 0, Y, 9999, Y + cyLineHeight);
                    }

               EndPaint(hwnd, &ps);
               return 0;
               }
          default :
               return DefWindowProc(hwnd, iMsg, wParam, lParam);
          }
     }
```

CALLER.H

```
#define IDM_CREATE           1
#define IDM_DESTROY          2
#define IDM_IUNKNOWN         3
#define IDM_IMALLOC          4
#define IDM_IMARSHAL         5
#define IDM_ALLOCATE_CUSTOM  6
#define IDM_ALLOCATE_DEFAULT 7
#define IDM_FREE             8
#define IDM_CHECK            9
```

CALLER.RC

```
#include "caller.h"

MAIN MENU
     {
     POPUP "&Allocator"
          {
```

```
        MENUITEM "&Create",                     IDM_CREATE
        MENUITEM "&Destroy",                    IDM_DESTROY
        MENUITEM SEPARATOR
        MENUITEM "QueryInterface IID_IUnknown", IDM_IUNKNOWN
        MENUITEM "QueryInterface IID_IMalloc",  IDM_IMALLOC
        MENUITEM "QueryInterface IID_IMarshal", IDM_IMARSHAL
        }

    POPUP "&Memory"
        {
        MENUITEM "&Allocate(IMalloc)", IDM_ALLOCATE_CUSTOM
        MENUITEM "&Allocate(malloc)",  IDM_ALLOCATE_DEFAULT
        MENUITEM "&Free",               IDM_FREE
        }
}
```

Рис. 20.4 Программа CALLER

Программа CALLER непосредственно не использует библиотеки OLE и, следовательно, как и представленная ранее библиотека IMALLOC.DLL, программа CALLER не инициализирует библиотеки OLE. (Это изменится после модернизации программы CALLER в программу CALLPUB, представленную далее в этой главе; так же как и наследник библиотеки IMALLOC — библиотека PUBMEM, тоже вынуждена иметь дело с библиотеками OLE и, следовательно, иметь соответствующие функции инициализации и удаления.)

Весь доступ к услугам интерфейса *IMalloc* в программе CALLER происходит посредством выбора пунктов меню, что обеспечивает полный контроль над тем, что вызывается и тем, когда это вызывается. Первое, что необходимо сделать при запуске программы CALLER — это создать компонент для выделения памяти, что достигается посредством выбора опции Create из меню Allocator. Это приводит к вызову функции *CreateAllocator* библиотеки IMALLOC, возвращаемым значением которой является указатель на интерфейс, требуемый для доступа к интерфейсу процедуры выделения памяти библиотеки IMALLOC.

При выборе опции Destroy из меню Allocator, компонент для выделения памяти удаляется благодаря уменьшению значения счетчика ссылок при вызове функции *Release*:

```
pMalloc -> lpVtbl -> Release(pMalloc);
pMalloc = NULL;
```

Устанавливать значение только что освобожденного указателя на интерфейс в NULL очень полезно. Это гарантирует, что его невозможно будет снова использовать без генерации исключения (что означает программную ошибку, от которой вы, будем надеяться, избавитесь до того, как начнете распространять вашу программу).

Программа CALLER вызывает функцию-член *QueryInterface* из процедуры выделения памяти для проверки поддерживаемых интерфейсов. С позиции клиента, важно всегда вызывать функцию *Release*, когда вызов функции *QueryInterface* возвращает указатель на интерфейс. В противном случае, вы рискуете оставить компонент в оперативной памяти со счетчиком ссылок большим нуля, хотя клиент считает, что он освободил ресурсы.

Теперь обратимся к тому, что представляет собой создание сервера открытого компонента.

Сервер открытого компонента

Библиотека PUBMEM.DLL демонстрирует создание сервера открытого компонента OLE. Отличие между открытым и закрытым компонентами заключается не в самом интерфейсе; фактически открытый интерфейс *IMalloc*, поставляемый библиотекой PUBMEM идентичен (за небольшим исключением) закрытому интерфейсу, поставляемому в представленном ранее образце библиотеки IMALLOC. Открытый компонент OLE — это компонент, созданный из класса открытого компонента OLE.

Класс открытого компонента OLE требует координации нескольких элементов. Класс компонента имеет уникальный идентификатор класса (CLSID) и элемент реестра (registry), который связывает идентификатор класса с инсталлированными файлами с расширением .EXE и .DLL. Сам компонент должен обеспечивать стандартный механизм для создания компонента по запросу, который называется фабрикой классов (class factory). Фабрика классов является закрытым компонентом OLE, созданным посредством вызова экспортируемой функции *DllGetClassObject* сервера DLL, которая обеспечивает услуги интерфейса *IClassFactory*. Библиотеки OLE периодически вызывают вторую точку входа — функцию *DllCanUnloadNow*, для проверки того, безопасно ли для состояния DLL удалить компонент. Будучи более сложной, чем закрытый компонент, фабрика классов обеспечивает базовый механизм для выдачи многих типов классов компонентов, каждый из которых предусматривает несколько разных интерфейсов. На рис. 20.5 представлены файлы с исходным текстом библиотеки PUBMEM.

PUBMEM.MAK

```
#----------------------
# PUBMEM.MAK make file
#----------------------
pubmem.dll : pubmem.obj classfac.obj compobj.obj
     $(LINKER) /EXPORT:DllGetClassObject /EXPORT:DllCanUnloadNow \
     $(DLLFLAGS) -OUT:pubmem.dll pubmem.obj \
classfac.obj compobj.obj $(GUILIBS) uuid.lib ole32.lib

pubmem.obj : pubmem.cpp pubmem.h
     $(CC) $(CFLAGS) pubmem.cpp

classfac.obj : classfac.cpp pubmem.h
     $(CC) $(CFLAGS) classfac.cpp

compobj.obj : compobj.cpp pubmem.h
     $(CC) $(CFLAGS) compobj.cpp
```

PUBMEM.CPP

```
/*-----------------------------------------------
   PUBMEM.CPP -- Define a public imalloc component
                 (c) Paul Yao, 1996
  ----------------------------------------------*/
#include <windows.h>
#include "pubmem.h"

extern int cObject;

//---------------------------------------------------------------------
// CreateAllocator -- Exported function to create allocator
//---------------------------------------------------------------------
EXPORT LPMALLOC CreateAllocator()
     {
     DAlloc *pAllocator = NULL;

     pAllocator = new DAlloc();
     if(pAllocator != NULL && pAllocator->Initialize())
          {
          pAllocator->AddRef();
          }
     else
          {
          delete pAllocator;
          }

     return(LPMALLOC) pAllocator;
     }

//---------------------------------------------------------------------
DAlloc::DAlloc()
     {
     RefCount = 0;
     hHeap = NULL;
     }
//---------------------------------------------------------------------
DAlloc::~DAlloc()
     {
     if(hHeap)
          HeapDestroy(hHeap);
     }

//---------------------------------------------------------------------
BOOL DAlloc::Initialize()
     {
     hHeap = HeapCreate(0, 4096, 65535);
     return(BOOL) hHeap;
```

```
     }

//------------------------------------------------------------------
STDMETHODIMP
DAlloc::QueryInterface(REFIID riid, LPVOID FAR *ppvObject)
     {
     // Always initialize "out" parameters to NULL
     *ppvObject = NULL;

     // Everyone supports IUnknown
     if(riid == IID_IUnknown)
          *ppvObject =(LPUNKNOWN) this;

     // We support IMalloc
     if(riid == IID_IMalloc)
          *ppvObject =(LPMALLOC) this;

     if(*ppvObject == NULL)
          {
          // Interface not supported
          return E_NOINTERFACE;
          }
     else
          {
          // Interface supported, so increment reference count
         ((LPUNKNOWN) *ppvObject)->AddRef();
          return S_OK;
          }
     }

//------------------------------------------------------------------
STDMETHODIMP_(ULONG)
DAlloc::AddRef(void)
     {
     return ++RefCount;
     }

//------------------------------------------------------------------
STDMETHODIMP_(ULONG)
DAlloc::Release(void)
     {
         if(0L != --RefCount)
                  return RefCount;

         --cObject;
     delete this;
         return 0L;
     }

//------------------------------------------------------------------
STDMETHODIMP_(void *)
DAlloc::Alloc(ULONG cb)
     {
     return HeapAlloc(hHeap, HEAP_ZERO_MEMORY, cb);
     }

//------------------------------------------------------------------
STDMETHODIMP_(void *)
DAlloc::Realloc(void *pv, ULONG cb)
     {
     return HeapReAlloc(hHeap, HEAP_ZERO_MEMORY, pv, cb);
     }

//------------------------------------------------------------------
STDMETHODIMP_(void)
```

```
DAlloc::Free(void *pv)
     {
     HeapFree(hHeap, 0, pv);
     }

//------------------------------------------------------------------
STDMETHODIMP_(ULONG)
DAlloc::GetSize(void *pv)
     {
     return HeapSize(hHeap, 0, pv);
     }

//------------------------------------------------------------------
STDMETHODIMP_(int)
DAlloc::DidAlloc(void *pv)
     {
     PROCESS_HEAP_ENTRY phe;
     ZeroMemory(&phe, sizeof(PROCESS_HEAP_ENTRY));

     while(HeapWalk(hHeap, &phe))
          {
          if(phe.lpData == pv)
               return 1;
          }
     return 0;
     }

//------------------------------------------------------------------
STDMETHODIMP_(void)
DAlloc::HeapMinimize(void)
     {
     HeapCompact(hHeap, 0);
     }
```

CLASSFAC.CPP

```
/*-------------------------------------------
   CLASSFAC.CPP -- OLE Class Factory component
                (c) Paul Yao, 1996
  -------------------------------------------*/
#include <windows.h>
#include <initguid.h>
#include "pubmem.h"

extern int cObject;
extern int cLockCount;

//------------------------------------------------------------------
DClassFactory::DClassFactory()
     {
     RefCount = 0;
     }

//------------------------------------------------------------------
DClassFactory::~DClassFactory()
     {
     }

//------------------------------------------------------------------
STDMETHODIMP
DClassFactory::QueryInterface(REFIID riid, LPVOID FAR *ppvObj)
     {
     // Init recipient's pointer
     *ppvObj = NULL;

     // If asking for IUnknown, we can provide
```

```
     if(riid == IID_IUnknown)
          *ppvObj =(LPUNKNOWN) this;

     // If asking for IClassFactory, we can provide
     if(riid == IID_IClassFactory)
          *ppvObj =(LPCLASSFACTORY) this;

     // Make sure reference count reflects access
     if(*ppvObj == NULL)
          {
          // Interface not supported
          return E_NOINTERFACE;
          }
     else
          {
          // Interface supported to increment reference count
         ((LPUNKNOWN) *ppvObj)->AddRef();
          return S_OK;
          }
     }

//------------------------------------------------------------------
STDMETHODIMP_(ULONG)
DClassFactory::AddRef()
     {
     return ++RefCount;
     }

//------------------------------------------------------------------
STDMETHODIMP_(ULONG)
DClassFactory::Release()
     {
     if(0L != --RefCount)
                 return RefCount;

         delete this;
         return 0L;
     }

//------------------------------------------------------------------
STDMETHODIMP
DClassFactory::CreateInstance(LPUNKNOWN pUnkOuter, REFIID riid,
                              LPVOID FAR *ppvObject)
     {
     // Initialize return pointer
     *ppvObject = NULL;

     // If trying to aggregate, fail
     if(pUnkOuter != NULL)
          return CLASS_E_NOAGGREGATION;

     // Create memory allocation object
     LPMALLOC pMalloc = CreateAllocator();

     if(pMalloc == NULL)
          {
          return E_OUTOFMEMORY;
          }
     else
          {
          // Fetch interface requested by caller
          HRESULT hr = pMalloc->QueryInterface(riid, ppvObject);

          // Decrement reference count produced by CreateAllocator
          pMalloc->Release();
```

```
          // Increment count of objects
                  if(SUCCEEDED(hr))
               ++cObject;

          return hr;
          }
     }

//------------------------------------------------------------------
STDMETHODIMP
DClassFactory::LockServer(BOOL fLock)
     {
     if(fLock)
          {
          ++cLockCount;
          }
     else
          {
          --cLockCount;
          }
     return NOERROR;
     }
```

COMPOBJ.CPP

```
/*---------------------------------------------
   COMPOBJ.CPP -- Component Object registration
                 (c) Paul Yao, 1996
  ---------------------------------------------*/
#include <windows.h>
#include "pubmem.h"

int cObject    = 0;
int cLockCount = 0;

//------------------------------------------------------------------
HRESULT APIENTRY
DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID *ppvObj)
     {
     // Initialize "out" pointer to known value
         *ppvObj = NULL;

     if(rclsid != CLSID_ALLOCATOR)
          {
          return CLASS_E_CLASSNOTAVAILABLE;
          }

         DClassFactory *pClassFactory = new DClassFactory();
         if(pClassFactory == NULL)
          {
          return E_OUTOFMEMORY;
          }
     else
          {
          return pClassFactory->QueryInterface(riid, ppvObj);
              }
     }

//------------------------------------------------------------------
HRESULT APIENTRY
DllCanUnloadNow(void)
     {
     if(cObject > 0 || cLockCount > 0)
          {
          return S_FALSE;
```

```
          }
     else
          {
          return S_OK;
          }
     }
```

PUBMEM.H

```
//--------------------------------------------------------------------
// C Interface to private allocator
//--------------------------------------------------------------------
#define EXPORT extern "C" __declspec(dllexport)

EXPORT LPMALLOC CreateAllocator();

// {308D0430-1090-11cf-B92A-00AA006238F8}
DEFINE_GUID(CLSID_ALLOCATOR,
     0x308d0430, 0x1090, 0x11cf, 0xb9,
     0x2a, 0x0, 0xaa, 0x0, 0x62, 0x38, 0xf8);

//--------------------------------------------------------------------
// Implementation of allocator interface
//--------------------------------------------------------------------
#undef  INTERFACE
#define INTERFACE DAlloc

DECLARE_INTERFACE_(DAlloc, IMalloc)
     {
     // *** IUnknown methods ***
     STDMETHOD (QueryInterface)(THIS_ REFIID riid, LPVOID FAR *ppv);
     STDMETHOD_(ULONG, AddRef) (THIS);
     STDMETHOD_(ULONG, Release)(THIS);

     // *** IMalloc methods ***
     STDMETHOD_(void *, Alloc)       (THIS_ ULONG cb);
     STDMETHOD_(void *, Realloc)     (THIS_ void *pv, ULONG cb);
     STDMETHOD_(void,   Free)        (THIS_ void *pv);
     STDMETHOD_(ULONG,  GetSize)     (THIS_ void *pv);
     STDMETHOD_(int,    DidAlloc)    (THIS_ void *pv);
     STDMETHOD_(void,   HeapMinimize)(THIS);

#ifndef CINTERFACE
public :
     DAlloc();
     ~DAlloc();
     BOOL Initialize();
private :
     ULONG RefCount;
     HANDLE hHeap;
#endif
     };

// Class Factory
#undef  INTERFACE
#define INTERFACE DClassFactory

DECLARE_INTERFACE_(DClassFactory, IClassFactory)
     {
     // *** IUnknown methods ***
     STDMETHOD (QueryInterface)(THIS_ REFIID riid, LPVOID FAR* ppv);
     STDMETHOD_(ULONG, AddRef) (THIS);
     STDMETHOD_(ULONG, Release)(THIS);

     // *** IClassFactory methods ***
     STDMETHOD(CreateInstance)(THIS_ LPUNKNOWN pUnkOuter,
```

```
                                REFIID riid, LPVOID FAR *ppvObject);
     STDMETHOD(LockServer)    (THIS_ BOOL fLock);

#ifndef CINTERFACE
public :
     DClassFactory();
     ~DClassFactory();
private :
     ULONG RefCount;
#endif
     };
```

Рис. 20.5 Библиотека PUBMEM

В библиотеке PUBMEM имеется две экспортируемые точки входа: функции *DllGetClassObject* и *DllCanUnloadNow*. Поскольку библиотеки OLE ожидают, что сервер DLL предлагает точные имена этих функций, то мы не можем определять эти функции с использованием ключевого слова EXPORT. Вместо этого мы помещаем эти функции в make-файл PUBMEM.MAK с опцией компоновщика /EXPORT. Функция *DllGetClassObject* обеспечивает механизм для фабрики классов. Функция *DllCanUnloadNow* вызывается, когда библиотекам OLE нужно узнать, есть ли у конкретного сервера какие-либо действующие соединения или безопасно ли отключать сервер и удалять его из оперативной памяти. Об обеих функциях будет рассказано ниже, но сначала давайте разберемся с реестром — ключевым элементом, делающим компонент открытым.

Назначение реестра

Реестр является центральным хранилищем, в котором хранится информация об общих настройках в системах Windows 95 и Windows NT. Впервые реестр появился в Windows 3.1 для отображения деталей классов OLE, заданных по умолчанию расширениях оболочки операционной системы и нескольких дополнительных команд DDE. В Windows 95 и Windows NT в содержимое реестра включено состояние системы, которое ранее сохранялось в файлах с расширением .INI. Реестр содержит данные об инсталлированной в данный момент аппаратной части, опциях панели управления и выбранными пользователем установками программного обеспечения. Хотя в том, как каждая операционная система использует реестр имеются важные отличия, все, что в реестре связано с OLE, одинаково в обеих системах.

Реестр имеет иерархическую структуру, он подразделяется на ключи, подключи и значения. В Windows 95 определяемые корневые ключи включают в себя: HKEY_CLASSES_ROOT, HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE, HKEY_USERS, HKEY_CURRENT_CONFIG и HKEY_DYN_DATA. (В Windows NT имеется только первые четыре ключа.) Данные, относящиеся к компонентам OLE, находятся в HKEY_CLASSES_ROOT. (Обратите внимание, что истинное положение компонентов OLE в иерархии реестра следующее: HKEY_LOCAL_MACHINE\SOFTWARE\Classes, но для совместимости с приложениями OLE Windows 3.1, для классов OLE выделен собственный суррогатный корневой узел.)

Корневые ключи HKEY_CLASSES_ROOT состоят из элементов трех типов: расширений файлов, имен классов и системных входов. Элементы расширений файлов начинаются с точки (.), которая связывает файл с сервером составного документа (например, ".vsd" с Visio фирмы Visio Corporation). Имена классов представляют собой понятные человеку идентификаторы классов компонентов OLE, что в настоящее время можно использовать двумя способами. Первым применением имен классов является поддержка серверов составного документа OLE 1.0; при этом имена классов идентифицируются, например, так: "Visio.Drawing.4". Вторым использованием имен классов является автоматизация OLE. Примитивы макрокоманд автоматизации по имени создают объекты автоматизации OLE — в этом контексте имя класса называется "идентификатором программы" или "ProgID". Например, используя "Visio.Application" ProgID, программа Visual Basic может применять методы и свойства автоматизации, создавать объекты Visio и управлять ими.

Имеется три типа системных элементов реестра, каждый из которых является корневым и имеет собственную иерархию: *TypeLib*, *Interface* и CLSID. Иерархия *TypeLib* идентифицирует положение инсталлированных в данный момент библиотек типов, которые являются базами данных, описывающими содержимое компонентов OLE. Широко используемая для поддержки автоматизации библиотека типов описывает прототипы функций для всех поддерживаемых интерфейсов, а также включает в себя ссылки на файлы-подсказки, и поэтому инструменты разработки могут вызвать соответствующую страницу подсказки и помочь создателям макрокоманд правильно воспользоваться серверами автоматизации.

Иерархия *Interface* содержит список, отсортированный по идентификаторам интерфейса, всех инсталлированных в системе интерфейсов. Это дает возможность прочитать имя интерфейса (*IUnknown, IMalloc* и т. д.) и подробности о каждом интерфейсе (количество функций в каждом интерфейсе и его базовый класс).

Последняя иерархия, иерархия CLSID детализирует все инсталлированные в данный момент (открытые) компоненты OLE. CLSID — это идентификатор класса. Также, как и идентификаторы интерфейсов (типы данных

IID и REFIID), идентификаторы классов (типы данных CLSID и REFCLSID) являются 128-разрядными (16 шестнадцатеричных цифр) числами, обеспечивающими машинный способ точной идентификации класса компонента. Также, как тип IID, CLSID имеет тип GUID:

```
typedef struct _GUID
{
       DWORD  Data1;
       WORD   Data2;
       WORD   Data3;
       BYTE   Data4[8];
} GUID;
```

CLSID похож на телефонный номер конкретного компонента OLE. При предыдущем обсуждении идентификаторов интерфейса (IID), мы сравнивали их с добавочными телефонными номерами, которые используются для связи между отделами одного и того же учреждения. С такой точки зрения, CLSID обеспечивает связь между учреждениями, или в терминах OLE, между компонентами. Точно также, как при звонке сотруднику конкретной компании часто требуется сначала набрать основной телефонный номер компании, а затем номер местного телефона, соединение с конкретным интерфейсом конкретного компонента требует сначала идентифицировать CLSID для доступа к компоненту, а затем IID для получения желаемого интерфейса.

Продолжая аналогию с телефоном, реестр похож на всемирную телефонную книгу компонентов OLE. На компонент можно сослаться через расширение файла, имя класса или идентификатор класса. Из этих трех типов элементов самым важным является элемент идентификатор класса, поскольку детали модуля компонента (файл с расширением .DLL или .EXE) хранятся в иерархии CLSID. Хотя вам (или вашей программе инсталляции) потребуется создавать элементы в иерархии, программный доступ к иерархии не потребуется. Вместо этого сделайте вызов библиотек OLE для получения доступа к идентификатору класса, найдите реестр и загрузите себе желаемый компонент.

Следующий элемент реестра делает доступным компонент библиотеки PUBMEM:

```
HKEY_CLASSES_ROOT\
       CLSID\
              {308D0430 – 1090 – 11cf – B92A – 00AA006238F8}\
                     InprocServer32 = C:\PETZOLD\CHAP20\PUBMEM.DLL
```

InprocServer32 означает, что файл, на который ссылаются, является 32-разрядным файлом с расширением .DLL. Другими ключевыми словами являются *LocalServer32* для 32-разрядного файла с расширением .EXE, *InprocServer* для 16-разрядного файла с расширением .DLL и *LocalServer* для 16-разрядного файла с расширением .EXE. Чтобы показать наличие этих трех типов серверов, требуются дополнительные элементы реестра.

Имеется два способа добавления элементов реестра: вызов интерфейса программирования приложений реестра (registry API) или использование инструмента, что в конечном итоге ведет к вызову API. Чтобы добавить элемент в реестр без написания какой бы то ни было программы, запустите редактор реестра. Для каждой операционной системы Microsoft имеется своя версия: для Windows NT 3.51 — это REGEDT32.EXE, а для Windows 95 — это REGEDIT.EXE. Как уже упоминалось в этой главе, связанные с OLE элементы появляются в иерархии HKEY_CLASSES_ROOT.

Чтобы программно модифицировать реестр, вызываются различные функции для редактирования реестра, в таких функциях имеется префикс "Reg". Каждый новый уровень в иерархии реестра представляется описателем ключа реестра HKEY. Регистрация открытой процедуры выделения памяти в PUBMEM.DLL включает в себя открытие существующего ключа реестра (*RegOpenKeyEx*), создание двух новых ключей (*RegCreateKeyEx*), задание одного значения (*RegSetValueEx*) и затем закрытие трех открытых ключей (*RegCloseKey*). Если предположить, что путь к серверу компонента C:\PETZOLD\CHAP20\PUBMEM.DLL, то на рис. 20.6 представлена программа внесения изменений в реестр для библиотеки PUBMEM.DLL.

```
{
       DWORD         dwDisp;
       HKEY          hkMain;
       HKEY          hkClass;
       HKEY          hkPath;
       LPCTSTR       lpClsid = "{308D0430-1090-11CF-B92A-00AA006238F8}";
       LPCTSTR       lpPath = "InprocServer32";
       LPCTSTR       lpValue = "C:\\PETZOLD\\CHAP20\\PUBMEM.DLL";

       // Открываем "HLEY_CLASSES_ROOT\CLSID"
       RegOpenKeyEx(
              HKEY_CLASSES_ROOT,
              "CLSID",
              0,
              KEY_ALL_ACCESS,
              &hkMain
       );
```

```
        // Добавляем \HKEY_CLASSES_ROOT\CLSID\{308...8F8}
        RegCreateKeyEx(
                hkMain,
                lpClsid,
                0,
                "",
                REG_OPTION_NON_VOLATILE,
                KEY_ALL_ACCESS,
                NULL,
                &hkClass,
                &dwDisp
        );


if(dwDisp == REG_CREATED_NEW_KEY)
        {
                // Добавляем \...\{308...8F8}\InprocServer32
                RegCreateKeyEx(
                        hkClass,
                        lpPath,
                        0,
                        "",
                        REG_OPTION_NON_VOLATILE,
                        KEY_ALL_ACCESS,
                        NULL,
                        &hkPath,
                        &dwDisp
                );

                RegSetValueEx(
                        hkPath,
                        "",
                        0,
                        REG_SZ,
                        (CONST BYTE *) lpValue,
                        lstrlen(lpValue) + 1
                );

        }
RegCloseKey(hkPath);

RegCloseKey(hkClass);

RegCloseKey(hkMain);

}
```

Рис. 20.6 Программа внесения изменений в реестр для библиотеки PUBMEM.DLL

Реестр предоставляет клиенту доступ к серверу открытого компонента, при этом достаточно знать только идентификатор CLSID компонента. Давайте подробнее рассмотрим процесс генерации уникальных значений и приемы программирования для этого типа данных.

Способы генерации и использования идентификаторов CLSID

При любых операциях с компонентами OLE уникальный идентификатор класса является таким же важным, каким является уникальный телефонный номер при любых телефонных операциях. Без специальных соглашений два учреждения не смогут пользоваться одним и тем же номером телефона, то же можно сказать и о компонентах OLE: чтобы избежать сложностей, для каждого требуется уникальный идентификатор класса. Проблема усугубляется для сетевого OLE, поскольку возможность конфликтов между идентификаторами классов увеличивается с увеличением числа компьютеров — в каждом из которых имеются собственные компоненты OLE — все это добавляет в уравнение новые неизвестные.

Решение, которое используется для компонентов OLE, фактически пришло из мира компьютерных сетей. Элементизация Open Software Foundation (OSF) создала универсальные уникальные идентификаторы (Universally Unique Identifiers, UUID) для своего стандарта распределенных компьютерных систем (Distrributed Computing Environment, DCE). При программировании OLE идентификаторы UUID называются идентификаторами GUID и

используются как в качестве идентификаторов класса компонента (CLSIDs), так и в качестве идентификаторов интерфейса (IIDs).

Уникальный GUID — для идентификации либо классов компонентов, либо собственных пользовательских интерфейсов — генерируется с помощью утилиты GUIDGEN или утилиты UUIDGEN. Как писал Kraig Brockschmidt в книге "*Inside OLE*", эти программы генерируют уникальные значения с помощью разных комбинаций уникальных идентификаторов IEEE для сетевых адаптеров, текущих даты и времени и значения счетчика высокочастотных запросов на выделение памяти. Результатом является число с чрезвычайно низкой вероятностью воспроизведения другими разработчиками.

Утилита GUIDGEN генерирует уникальный GUID и помещает полученное значение в папку обмена в формате CF_TEXT в одной из нескольких форм, включая специальный формат MFC, естественный формат OLE (не MFC) и отформатированную для реестра строку GUID. Далее представлены выходные данные утилиты GUIDGEN, используемые библиотекой PUBMEM для идентификации класса компонентов процедуры выделения памяти *IMalloc*:

```
// {308D0430 — 1090 — 11cf — B92A — 00AA006238F8}

DEFINE_GUID(CLSID_ALLOCATOR, \
       0x308d0430, 0x1090, 0x11cf, 0xb9, \
       0x2a, 0x0, 0xaa, 0x0, 0x62, 0x38, 0xf8);
```

В комментарии содержится строка, необходимая для реестра и имеющая символы "тире" в конкретных позициях, а также открывающую и закрывающую фигурную скобку. Символьный идентификатор CLSID_ALLOCATOR, был добавлен к результатам, полученным с помощью утилиты GUIDGEN в качестве имени соответствующего компонента библиотеки PUBMEM.

Чтобы правильно использовать макрокоманду DEFINE_GUID требуется быть чуть внимательней, поскольку она имеет два разных определения. Стандартное определение ссылается на имя идентификатора GUID как на внешнее (*extern*) значение. В нем — и только в нем — необходимо ссылаться на другое определение, что гораздо легче сделать, если перед макрокомандой DEFINE_GUID вставить заголовочный файл INITGUID.H:

```
// Переопределение DEFINE_GUID для того, чтобы выделить память для GUID
#include <initguid.h>

// Выделение памяти и инициализация CLSID_ALLOCATOR

DEFINE_GUID(CLSID_ALLOCATOR, \
       0x308d0430, 0x1090, 0x11cf, 0xb9, \
       0x2a, 0x0, 0xaa, 0x0, 0x62, 0x38, 0xf8);
```

При создании соединения OLE клиент/сервер, значение идентификатора CLSID передается от клиента в библиотеки OLE, которые передают это значение серверу. Клиент использует идентификатор класса, запрашивая доступ к конкретному компоненту. Библиотеки OLE используют идентификатор класса для поиска в реестре системы положения программы или динамически подключаемой библиотеки, соответствующей идентификатору класса. Когда, в конечном итоге компонент обнаруживается, элемент внутри сервера компонента динамически подключаемой библиотеки — фабрики классов — сообщает о поддержке сервером требуемого идентификатора CLSID. После того, как эта поддержка установлена, сама фабрика классов создает требуемые компоненты и передает клиенту первый указатель интерфейса.

Компонент фабрика классов

Фабрика классов является закрытым компонентом OLE, поддерживающим два интерфейса: *IUnknown* и *IClassFactory*. Она является закрытым компонентом, поскольку не имеет идентификатора класса, и ее присутствие не отмечено в реестре системы. Как и в случае с процедурой выделения памяти в программе IMALLOC, клиент получает доступ к фабрике классов через закрытую точку входа: функцию *DllGetClassObject*. Однако, в отличие от процедуры выделения памяти в программе IMALLOC, обычно сами библиотеки OLE получают доступ к этой точке входа и сами создают фабрику классов.

Прототип функции *DllGetClassObject* таков:

```
HRESULT DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID *ppv);
```

- *rclsid* задает желаемое значение идентификатора класса (CLSID).
- *riid* задает желаемое значение идентификатора интерфейса (IID), обычно это IID_ClassFactory.
- *ppv* указывает положение, куда должно быть записано возвращаемое значение.

Из этой единственной экспортируемой точки входа сервер динамически подключаемой библиотеки способен поддерживать множество различных классов. Первый параметр функции *DllGetClassObject, rclsid*, позволяет серверу узнать, какой конкретный класс должен быть создан. Как и в случае функции *QueryInterface*, если функция *DllGetClassObject* может обеспечить требуемый интерфейс, то она копирует указатель интерфейса — обычно, это

указатель интерфейса *IClassFactory* — в область памяти, заданную последним параметром. Если запрос удовлетворить невозможно, этот параметр получает значение NULL.

Основным назначением функции *DllGetClassObject* является проверка возможности поддержки определенного класса (*rclsid*) и определенного интерфейса (*riid*). Если требуемый класс поддерживается, функция *DllGetClassObject* библиотеки PUBMEM создает фабрику классов. Затем вызывается функция *QueryInterface* фабрики классов — как для увеличения своего счетчика ссылок, так и для проверки того, что требуемый интерфейс на самом деле поддерживается. В библиотеке PUBMEM версия этой функции выглядит следующим образом:

```
HRESULT APIENTRY DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID *ppvObj)
{
	// Инициализация указателя "вывода" известным значением
	*ppvObj = NULL;


	if(rclsid != CLSID_ALLOCATOR)
	{
		return CLASS_E_CLASSNOTAVAILABLE;
	}


	DClassFactory *pClassFactory = new DClassFactory();

	if(pClassFactory ==NULL)
	{
		return E_OUTOFMEMORY;
	}
	else
	{

		return pClassFactory -> QueryInterface(riid, ppv);
	}
}
```

Как и функция *QueryInterface*, функция *DllGetClassObject* устанавливает возвращаемый указатель интерфейса в NULL, если не поддерживается требуемый класс или требуемый интерфейс фабрики классов.

Сам интерфейс *IClassFactory* определяется следующим образом:

```
#undef INTERFACE
#define INTERFACE IClassFactory


DECLARE_INTERFACE_(IClassFactory, IUnknown)
{
	// *** методы IUnknown ***

	STDMETHOD(QueryInterface)(THIS_    REFIID riid, LPVOID FAR *ppvObj) PURE;

	STDMETHOD_(ULONG, AddRef)(THIS)  PURE;

	STDMETHOD_(ULONG, Release)(THIS)  PURE;

	// *** методы IClassFactory ***

	STDMETHOD(CreateInstance)(THIS_ LPUNKNOWN pUnkOuter, REFIID riid,
				LPVOID FAR *ppvObject) PURE;

	STDMETHOD(LockServer)(THIS_BOOL fLock) PURE;
};
```

Как и все интерфейсы OLE, интерфейс *IClassFactory* имеет общий набор функций-членов *IUnknown*. Все, что делает *IClassFactory*, это создание собственного отдельного и отличного от других компонента. Чтобы поддерживать это различие между объектами сервера и объектами фабрики классов, функция *QueryInterface* не должна создавать для фабрики классов указатель на объект сервера. И наоборот — функция *QueryInterface* не должна создавать для объекта сервера указатель на интерфейс фабрики классов. Вместо этого фабрика классов задается отдельно, как уникальный компонент, который может создаваться и использоваться по мере необходимости.

У интерфейса *IClassFactory* имеется две характерные для него услуги: создание компонента и управление счетчиком захвата сервера.

Если клиент хочет создать экземпляр компонента, то вызывается функция-член *CreateInstance*. Однако, большинство клиентов не хотели бы непосредственно вызывать эту функцию. Вместо этого, для запроса на создание компонента

OLE, клиенты вызывают вспомогательную функцию библиотеки OLE — функцию *CoCreateInstance*. Эта функция определяется следующим образом:

```
STDAPI CoCreateInstance(
              REFCLSID rclsid,            // Идентификатор класса
              LPUNKNOWN pUnkOuter,        // Указатель IUnknown
              DWORD dwClsContext,         // Контекст сервера
              REFIID riid,                // Идентификатор интерфейса
              LPVOID *ppv
);            // Возвращаемое значение
```

- *rclsid* задает идентификатор класса компонента для его создания.
- *pUnkOuter* задает указатель интерфейса *IUnknown* для создания агрегированных объектов.
- *dwClsContext* идентифицирует контекст компонента сервера. Может быть равен CLSCTX_INPROC_SERVER для сервера динамически подключаемой библиотеки, CLSCTX_INPROC_HANDLER для обработчика динамически подключаемой библиотеки (функции-посредники для сервера вне процесса клиента) и CLSCTX_LOCAL_SERVER для локального сервера (различные процессы на одной машине). Для сетевого сервера никакого флага не задается.
- *riid* идентифицирует требуемый в качестве возвращаемого значения интерфейс.
- *ppv* указывает на положение значения возвращаемого интерфейса, в случае успешного создания компонента или на NULL, если компонента с требуемым интерфейсом создать невозможно.

Функция *LockServer* интерфейса *IClassFactory* как увеличивает, так и уменьшает на 1 значение счетчика захвата сервера. Она сочетает в одной функции сервера задачи, которые выполняются функциями-членами *AddRef* и *Release* интерфейса *IUnknown*. Исключение состоит в том, что сервер никогда не освобождает сам себя, как это делает объект компонента. Это приводит к важному результату в реализации сервера, а именно, в управлении временем жизни сервера.

Управление временем жизни сервера

Время жизни сервера является столь же важной проблемой при реализации сервера, как и время жизни компонента — при реализации компонента. Это верно и для однокомпонентного сервера, как в библиотеке PUBMEM, и для многокомпонентного сервера. Механизмы управления временем жизни у обоих типов одинаковы. Мы рассматриваем время жизни сервера, основанного на динамически подключаемых библиотеках. Хотя факторы, влияющие на время жизни сервера, основанного на файлах с расширением .EXE те же самые, детали реализации разные. (За подробностями обращайтесь к книге "*Inside OLE*" (OLE изнутри), автора Brockschmidt.)

Сервер динамически подключаемой библиотеки загружается в память также, как и другие динамически подключаемые библиотеки Windows — с помощью вызова функции *LoadLibrary*. Факт, что в библиотеках OLE имеется собственная версия этой функции *CoLoadLibrary*, не меняет того, что элемент, не относящийся к динамически подключаемым библиотекам загружает их в память. (Функция *CoLoadLibrary* создавалась для упрощения установки OLE на платформы, не являющиеся платформами Windows.) Динамически подключаемую библиотеку выгружает из памяти последующий вызов функции *CoFreeLibrary*.

Время жизни сервера динамически подключаемой библиотеки является проблемой, поскольку клиент непосредственно не загружает библиотеку сервера в оперативную память — это делают библиотеки OLE. Поэтому вовремя выгрузить сервер тоже является задачей библиотек OLE. Но большая часть взаимодействий между клиентом и сервером игнорирует библиотеки OLE, которые поэтому, кроме как по запросу, не имеют возможности узнать о наступлении подходящего времени для удаления сервера из оперативной памяти.

Библиотеки OLE периодически вызывают точку входа сервера-функцию *DllCanUnloadNow*, с запросом о том, не пора ли выгружать сервер. Функция отвечает"Да" (S_OK) или "Нет" (S_FALSE):

```
HRESULT APIENTRY DllCanUnloadNow(void)
{
      if(cObject > 0 || cLockCount > 0)
      {
            return S_FALSE;
      }
      else
      {
            return S_OK;
      }
}
```

Ответ основан на одном из двух факторов: существует ли хоть один компонент и зафиксирован ли в памяти сервер клиентом. Последнее означает ситуацию, в которой ни одного компонента не существует, но — чтобы избежать

дополнительной перезагрузки в память динамически подключаемой библиотеки — клиент хочет, чтобы сервер не выгружался.

Счетчик захвата сервера управляется функцией-членом *LockServer* интерфейса *IClassFactory*. Эта функция делает несколько больше, чем просто увеличивает и уменьшает на 1 счетчик захвата, который представляет из себя нечто большее, чем просто глобальную переменную, которую поддерживает сервер. Далее представлена функция-член *LockServer* интерфейса *IClassFactory* из библиотеки PUBMEM:

```
STDMETHODIMP DClassFactory::LockServer(BOOL fLock)
{
	if(fLock)
	{
		++cLockCount;
	}
	else
	{
		--cLockCount;
	}
	return NOERROR;
}
```

Другим фактором, управляющим временем жизни сервера, является счетчик компонентов. В библиотеке PUBMEM этот счетчик увеличивается на 1 внутри функции-члена *DClassFactory::CreateInstance* при удачном создании запрашиваемого компонента OLE:

```
// Создание объекта выделения памяти
LPMALLOC pMalloc = CreateAllocator();

if(pMalloc == NULL)
{
	return E_OUTOFMEMORY;
}
else
{
	// Выбор интерфейса, требуемого процедурой вызова
	HRESULT hr = pMalloc -> QueryInterface(riid, ppvObject);

	// Уменьшение на 1 счетчика ссылок, увеличенного функцией CreateAllocator
	pMalloc -> Release();

	// Увеличение на 1 счетчика объектов
	If(SUCCEEDED(hr))
		++cObject;

	return hr;
}
```

Этот счетчик уменьшается на 1, когда компонент удаляет сам себя, что происходит, когда в функции *Release* члене интерфейса *IMalloc* счетчик ссылок становится равным нулю:

```
STDMETHODIMP_(ULONG) DAlloc::Release(void)
{
	if(0L != RefCount) return RefCount;

	--cObject;
	delete this;
	return 0L;
}
```

Между прочим, это все изменения, которые необходимо внести при преобразовании закрытого компонента в открытый.

Настало время рассмотреть создание клиента открытого компонента, что демонстрирует следующий пример программы.

Клиент открытого компонента

Программа-клиент открытого компонента CALLPUB.EXE была создана из программы клиента закрытого компонента CALLER.EXE. Обе программы вызывают сервер компонента OLE, который обеспечивает интерфейс *IMalloc* для выделения и освобождения блоков памяти для хранения символьных строк.

Хотя для превращения закрытого сервера в открытый нужно было сделать несколько модификаций — специальные точки входа и фабрика классов — для превращения клиента закрытого компонента в клиента открытого компонента достаточно одного очень незначительного изменения. Вместо вызова функции создания закрытого компонента, программа CALLPUB вызывает функцию библиотеки OLE *CoCreateInstance*, единственной задачей которой должно быть создание компонентов.

Файлы с исходным текстом программы CALLPUB представлены на рис. 20.7.

CALLPUB.MAK

```
#----------------------
# CALLPUB.MAK make file
#----------------------

callpub.exe : callpub.obj callpub.res pubmem.lib
     $(LINKER) $(GUIFLAGS) -OUT:callpub.exe callpub.obj callpub.res \
         pubmem.lib $(GUILIBS) uuid.lib ole32.lib

callpub.obj : callpub.c callpub.h pubmem.h
     $(CC) -DCINTERFACE $(CFLAGS) callpub.c

callpub.res : callpub.rc callpub.h
     $(RC) $(RCVARS) callpub.rc
```

CALLPUB.C

```
/*-------------------------------------------
   CALLPUB.C -- Call into public OLE component
                (c) Paul Yao, 1996
  ------------------------------------------*/
#include <windows.h>
#include <initguid.h>
#include "pubmem.h"
#include "callpub.h"

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

char szWndClass[] = "CallerWindow";
char szAppName[]  = "Calls Public Malloc";

//--------------------------------------------------------------------
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR lpszCmdLine, int cmdShow)
     {
     HWND        hwnd;
     MSG         msg;
     WNDCLASSEX  wc;

     wc.cbSize        = sizeof(wc);
     wc.lpszClassName = szWndClass;
     wc.hInstance     = hInstance;
     wc.lpfnWndProc   = WndProc;
     wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
     wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
     wc.lpszMenuName  = "MAIN";
     wc.hbrBackground =(HBRUSH)(COLOR_WINDOW + 1);
     wc.style         = 0;
     wc.cbClsExtra    = 0;
     wc.cbWndExtra    = 0;
     wc.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);
```

```
     RegisterClassEx(&wc);

     hwnd = CreateWindowEx(0L, szWndClass, szAppName,
                           WS_OVERLAPPEDWINDOW,
                           CW_USEDEFAULT, CW_USEDEFAULT,
                           CW_USEDEFAULT, CW_USEDEFAULT,
                           NULL, NULL, hInstance, NULL);
     ShowWindow(hwnd, cmdShow);
     UpdateWindow(hwnd);

     // Connect to OLE libraries
     HRESULT hr = CoInitialize(NULL);
     if(FAILED(hr))
          {
          // Fail app initialization
          return FALSE;
          }

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

     // Disconnect from OLE libraries
     CoUninitialize();

     return msg.wParam;
     }

//---------------------------------------------------------------------
LRESULT CALLBACK
WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int       iCurLine = 0;
     static LPMALLOC  pMalloc = NULL;
     static LPSTR     szLine[10];
     static RECT      rHit[10];

     switch(iMsg)
          {
          case WM_CREATE :
               // Initialize data pointer array
               ZeroMemory(szLine, sizeof(szLine));
               return 0;

          case WM_COMMAND :
               switch(LOWORD(wParam))
                    {
                    case IDM_CREATE :
                         {
                         HRESULT hr =
                              CoCreateInstance(CLSID_ALLOCATOR,
                                               NULL,
                                               CLSCTX_INPROC_SERVER,
                                               IID_IMalloc,
                                              (void **) &pMalloc);

                         if(FAILED(hr))
                              {
                              MessageBox(hwnd, "Error: No allocator",
                                         szAppName, MB_OK);
                              return 0;
                              }
```

```
          InvalidateRect(hwnd, NULL, TRUE);
          return 0;
          }

     case IDM_DESTROY :
          {
          int i;

          // Mark allocated blocks as invalid
          for(i = 0; i < 10; i++)
               {
               if((szLine[i] != NULL) &&
                  (pMalloc->lpVtbl->DidAlloc(pMalloc,
                    szLine[i])))
                    {
                    szLine[i] = NULL;
                    }
               }

          // Disconnect from & free allocator
          pMalloc->lpVtbl->Release(pMalloc);
          pMalloc = NULL;

          InvalidateRect(hwnd, NULL, TRUE);
          return 0;
          }

     case IDM_IUNKNOWN :
          {
          LPUNKNOWN pUnk;
          HRESULT hr =
               pMalloc->lpVtbl->QueryInterface(pMalloc,
               IID_IUnknown,
              (void **) &pUnk);
          if(SUCCEEDED(hr))
               {
               pUnk->lpVtbl->Release(pUnk);
               MessageBox(hwnd, "IUnknown supported",
                           szAppName, MB_OK);
               }
          else
               {
               MessageBox(hwnd, "IUnknown not supported",
                           szAppName, MB_OK);
               }
          return 0;
          }

     case IDM_IMALLOC :
          {
          LPUNKNOWN pUnk;
          HRESULT hr =
               pMalloc->lpVtbl->QueryInterface(pMalloc,
               IID_IMalloc,
              (void **) &pUnk);
          if(SUCCEEDED(hr))
               {
               pUnk->lpVtbl->Release(pUnk);
               MessageBox(hwnd, "IMalloc supported",
                           szAppName, MB_OK);
               }
          else
               {
               MessageBox(hwnd, "IMalloc not supported",
                           szAppName, MB_OK);
```

```
                    }
               return 0;
               }

          case IDM_IMARSHAL :
               {
               LPUNKNOWN pUnk;
               HRESULT hr =
                    pMalloc->lpVtbl->QueryInterface(pMalloc,
                    IID_IMarshal,
                   (void **) &pUnk);
               if(SUCCEEDED(hr))
                    {
                    pUnk->lpVtbl->Release(pUnk);
                    MessageBox(hwnd, "IMarshal supported",
                               szAppName, MB_OK);
                    }
               else
                    {
                    MessageBox(hwnd, "IMarshal not supported",
                               szAppName, MB_OK);
                    }
               return 0;
               }

          case IDM_ALLOCATE_CUSTOM :
               if(szLine[iCurLine] != NULL)
                    {
                    MessageBox(hwnd, "Error: Free First",
                               szAppName, MB_OK);
                    return 0;
                    }

               // Allocate from IMalloc interface
               szLine[iCurLine] =
              (char *) pMalloc->lpVtbl->Alloc(pMalloc, 100);
               lstrcpy(szLine[iCurLine], "*IMalloc memory*");

               InvalidateRect(hwnd, NULL, TRUE);
               return 0;
          case IDM_ALLOCATE_DEFAULT :
               if(szLine[iCurLine] != NULL)
                    {
                    MessageBox(hwnd, "Error: Free First",
                               szAppName, MB_OK);
                    return 0;
                    }

               // Allocate from default heap
               szLine[iCurLine] =(char *) malloc(100);
               lstrcpy(szLine[iCurLine], "-Malloc memory-");

               InvalidateRect(hwnd, NULL, TRUE);
               return 0;

          case IDM_FREE :
               if(szLine[iCurLine] == NULL)
                    {
                    MessageBox(hwnd, "Error: Nothing to free",
                               szAppName, MB_OK);
                    return 0;
                    }

               if(pMalloc == NULL)
                    {
```

```
                    goto FreeMalloc;
                    }

               // Free allocated object
               if(pMalloc->lpVtbl->DidAlloc(pMalloc,
                   szLine[iCurLine]))
                    {
                    pMalloc->lpVtbl->Free(pMalloc,
                    szLine[iCurLine]);
                    }
               else
                    {
          FreeMalloc:
                    free(szLine[iCurLine]);
                    }

               szLine[iCurLine] = NULL;

               InvalidateRect(hwnd, NULL, TRUE);
               return 0;
          }

     case WM_DESTROY :
          // Disconnect from & free allocator
          if(pMalloc)
               {
               pMalloc->lpVtbl->Release(pMalloc);
               pMalloc = NULL;
               }

          PostQuitMessage(0);  // Handle application shutdown
          return 0;

     case WM_INITMENU :
          {
          HMENU hMenu =(HMENU) wParam;
          if(pMalloc)
               {
               EnableMenuItem(hMenu, IDM_CREATE,           MF_GRAYED);
               EnableMenuItem(hMenu, IDM_DESTROY,          MF_ENABLED);
               EnableMenuItem(hMenu, IDM_ALLOCATE_CUSTOM, MF_ENABLED);
               EnableMenuItem(hMenu, IDM_IUNKNOWN,         MF_ENABLED);
               EnableMenuItem(hMenu, IDM_IMALLOC,          MF_ENABLED);
               EnableMenuItem(hMenu, IDM_IMARSHAL,         MF_ENABLED);
               }
          else
               {
               EnableMenuItem(hMenu, IDM_CREATE,           MF_ENABLED);
               EnableMenuItem(hMenu, IDM_DESTROY,          MF_GRAYED);
               EnableMenuItem(hMenu, IDM_ALLOCATE_CUSTOM, MF_GRAYED);
               EnableMenuItem(hMenu, IDM_IUNKNOWN,         MF_GRAYED);
               EnableMenuItem(hMenu, IDM_IMALLOC,          MF_GRAYED);
               EnableMenuItem(hMenu, IDM_IMARSHAL,         MF_GRAYED);
               }
          return 0;
          }

     case WM_LBUTTONDOWN :
          {
          int i;
          int x = LOWORD(lParam);
          int y = HIWORD(lParam);
          POINT pt = { x, y };

          for(i = 0; i < 10; i++)
```

```
               {
               if(PtInRect(&rHit[i], pt))
                    {
                    if(iCurLine != i)  // Minimize screen blink
                         {
                         InvalidateRect(hwnd, &rHit[iCurLine], TRUE);
                         InvalidateRect(hwnd, &rHit[i], TRUE);
                         iCurLine = i;
                         }
                    break;
                    }
               }
          return 0;
          }

     case WM_PAINT :
          {
          char         szBuff[10];
          COLORREF     crText, crBack;
          HDC          hdc;
          int          cc;
          int          i;
          int          XCount, XText, Y;
          int          cyLineHeight;
          PAINTSTRUCT  ps;
          RECT         rOpaque;
          TEXTMETRIC   tm;

          hdc = BeginPaint(hwnd, &ps);

          // Fetch line height
          GetTextMetrics(ps.hdc, &tm);
          cyLineHeight = tm.tmHeight + tm.tmExternalLeading;

          // Fetch current text colors
          crText = GetTextColor(ps.hdc);
          crBack = GetBkColor(ps.hdc);

          XCount = tm.tmAveCharWidth * 3;
          XText  = XCount + tm.tmAveCharWidth * 7;
          Y      = tm.tmHeight;

          for(i = 0; i < 10; i++, Y += cyLineHeight)
               {
               // Set colors to highlight current line
               if(i == iCurLine)
                    {
                    SetTextColor(ps.hdc, crBack);
                    SetBkColor(ps.hdc, crText);

                    SetRect(&rOpaque, 0, Y, 9999, Y + cyLineHeight);
                    ExtTextOut(ps.hdc, 0, 0, ETO_OPAQUE, &rOpaque,
                              NULL, 0, NULL );
                    }
               else
                    {
                    SetTextColor(ps.hdc, crText);
                    SetBkColor(ps.hdc, crBack);
                    }

               // Display line count
               cc = wsprintf(szBuff, "Line %d", i);
               TextOut(ps.hdc, XCount, Y, szBuff, cc);

               // Display text if a string has been defined
```

```
                    if(szLine[i] != NULL)
                         {
                         cc = lstrlen(szLine[i]);
                         TextOut(ps.hdc, XText, Y, szLine[i], cc);
                         }

                    // Calculate hit test rectangle
                    SetRect(&rHit[i], 0, Y, 9999, Y + cyLineHeight);
                    }

               EndPaint(hwnd, &ps);
               return 0;
               }

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

CALLPUB.H

```
#define IDM_CREATE            1
#define IDM_DESTROY           2
#define IDM_IUNKNOWN          3
#define IDM_IMALLOC           4
#define IDM_IMARSHAL          5
#define IDM_ALLOCATE_CUSTOM   6
#define IDM_ALLOCATE_DEFAULT  7
#define IDM_FREE              8
#define IDM_CHECK             9
```

CALLPUB.RC

```
#include "callpub.h"

MAIN MENU
     {
     POPUP "&Allocator"
          {
          MENUITEM "&Create",                     IDM_CREATE
          MENUITEM "&Destroy",                    IDM_DESTROY
          MENUITEM SEPARATOR
          MENUITEM "QueryInterface IID_IUnknown", IDM_IUNKNOWN
          MENUITEM "QueryInterface IID_IMalloc",  IDM_IMALLOC
          MENUITEM "QueryInterface IID_IMarshal", IDM_IMARSHAL
          }

     POPUP "&Memory"
          {
          MENUITEM "&Allocate(IMalloc)", IDM_ALLOCATE_CUSTOM
          MENUITEM "&Allocate(malloc)",  IDM_ALLOCATE_DEFAULT
          MENUITEM "&Free",               IDM_FREE
          }
     }
```

Рис. 20.7 Программа CALLPUB

Поскольку она вызывает функцию библиотеки OLE, программа CALLPUB в начале функции *WinMain* инициализирует библиотеки OLE:

```
// Соединение с библиотеками OLE

HRESULT hr = CoInitialize(NULL);

if(FAILED(hr))
{
       // Неудачная инициализация
       return FALSE;
}
```

Перед окончанием работы программа CALLPUB отсоединяется от OLE:

```
// Отсоединение от библиотек OLE
CoUninitialize( );
```

Оба вызова необходимы, поскольку программа CALLPUB получает свой интерфейс *IMalloc* из сервера компонента PUBMEM с помощью вызова функции *CoCreateInstance*:

```
HRESULT hr = CoCreateInstance(CLSID_ALLOCATOR, NULL, CLSCTX_INPROC_SERVER, IID_IMalloc,
                      (void **) &pMalloc);

if(FAILED(hr))
{
       MessageBox(hwnd, "Error: No allocator",   szAppName, MB_OK);
       return 0;
}
```

Эта функция библиотеки OLE ищет в реестре компонент, который может обеспечить класс CLSID_ALLOCATOR, символ, определенный в заголовочном файле PUBMEM.H. Чтобы заставить это слово выделять память, а не просто ссылаться на дополнительное значение, в список включенных файлов библиотеки PUBMEM добавляется следующее:

```
#include <initguid.h>
#include "pubmem.h"
```

Когда писалась эта книга были определены три контекста для типов запускаемых серверов (параметр *dwClsContext* функции *CoCreateInstance*). CLSCTX_INPROC_SERVER, который запрашивается из PUBMEM, является самостоятельным DLL-сервером. Другим типом DLL-сервера, CLSCTX_INPROC_HANDLER, является локальный обработчик для сервера вне процесса клиента. Он работает как внутренний заменитель сервера процесса (быстрее, с меньшими накладными расходами) до тех пор, пока не понадобится сервер вне процесса. Третий тип контекста, CLSCTX_LOCAL_SERVER, запускается в качестве отдельного процесса на той же машине.

В отличие от закрытой функции *CreateAllocator*, которая всегда возвращает указатель на интерфейс *IMalloc*, функция *CoCreateInstance* позволяет вызывающей процедуре задать тип возвращаемого функцией интерфейса. PUBMEM запрашивает IDD_IMalloc, идентификатор интерфейса *IMalloc*. Но с тем же результатом она могла бы потребовать идентификатор IDD_IUnknown и затем выполнить вызов функции *QueryInterface* для указателя на интерфейс *IMalloc*. Конечно, если бы клиент запомнил оба указателя интерфейса, то для правильного уменьшения счетчика ссылок в конце концов потребовалось бы два вызова функции *Release*.

Заключение

Это введение в основы модели составного объекта (COM) OLE должно обеспечить хорошую базу для начала работы с технологиями OLE, где бы не пришлось с ними встретиться. Создаются ли контейнеры составных документов или серверы, объекты автоматизации или контроллеры, контейнеры элементов управления OLE или элементы управления OLE — в этой главе описано все, что имеет отношение к интерфейсам модели составного объекта OLE. Услуги оболочки Windows 95 доступны только через интерфейсы модели составного объекта OLE. Даже если вы собираетесь использовать библиотеки классов, например MFC компании Microsoft или OWL компании Borland, то уже при поверхностном знакомстве с ними вы обнаружите в них модель составного объекта OLE.

Тогда, когда Windows 95 только-только начала появляться перед взорами своих создателей, Microsoft разослал разработчикам программного обеспечения важное сообщение о том, что будущее принадлежит Win32 и OLE. С появлением Windows 95 и обещанным приходом технологий OLE становится ясно, что будущее уже пришло.

Теперь у вас есть все, что нужно для того, чтобы создавать грандиозные приложения для Windows завтрашнего дня.