П.В.Румянцев

АЗБУКА ПРОГРАММИРОВАНИЯ В WIN32 API

Изложены вопросы создания программных приложений для Windows 95 и Windows NT. Описаны основные типы переменных, макросов, функций. Материал книги иллюстрируется многочисленными примерами. Настоящее издание (второе вышло в 2000 г.) дополнено описанием тех возможностей Windows, которые не были упомянуты в предыдущих изданиях.

Для программистов.

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ К ТРЕТЬЕМУ ИЗДАНИЮ

Предлагаемая читателю книга является плодом долгих раздумий автора. Дело в том, что подавляющее большинство книг, посвященных программированию для Windows, написаны с таким расчетом, чтобы пользователь чуть ли не через пару прочитанных страниц мог начать программировать. Возможно, эта задача успешно выполняется. Но в этом случае возникает другая проблема: у многих программистов после получения первоначальных знаний, достаточных для создания окна и простых диалогов, отпадает желание двигаться дальше. Подавляющее большинство возможностей, предоставляемых системой, остаются вне поля зрения.

По моему мнению, такой подход к изложению основ Win32 API методологически не верен. Я решил пойти по другому пути: сначала рассказать читателю, какие возможности предоставляет Win32 API, и только потом научить его создавать пользовательский интерфейс. Тем самым я постараюсь достичь того, что программист будет понимать возможности системы. Это позволит ему создавать более интересные программы. Когда я начал изучение программирования Windows, в поле моего зрения была одна единственная книга - «Programming Windows» Чарльза Петцольда (Charles Petzold). Сейчас я могу сказать, что эта книга просто великолепна! Буквально первые несколько глав позволили мне, как говорится, «въехать» и далее работать самостоятельно, используя впоследствии эту книгу как справочное пособие.

При написании книги я предполагал дать быстрое и возможно более полное введение в программирование в Windows NT и Windows'95. При этом мне хотелось, чтобы материал, изложенный в ней, был полезен как начинающему программисту, так и специалисту, имеющему опыт в написании програм для Windows (в этой книге понятие «Windows» соответствует, как правило, Windows NT и Windows'95). Кроме того, мне хотелось избежать влияния на фантазию программистов, поэтому приведенные примеры не являются законченными. В них показано, как можно использовать возможности Win32 API. Предполагается, что читатель знаком с языком программирования C и имеет опыт работы с Windows'95 или Windows NT в качестве пользователя.

А теперь – ВНИМАНИЕ! Автор рассчитывает на то, что в момент чтения этой книги читатель будет сидеть за компьютером и вслед за автором пройдет по заголовочным файлам Win32, файлам ее системы помощи, и будет экспериментировать с теми заголовками программ, которые приводятся автором. В книге дано описание некоторых типов, применяемых в Win32, но, тем не менее, автор предполагает, что при описании переменных, типы которых не относятся к основным типам, определенным в языке C/C++, читатель проявит

любознательность и найдет описание переменной в одном из заголовочных файлов Win32.

В книге неоднократно делаются ссылки на «заголовочные файлы» (header'ы) Win32. Одним из отличий Win32 от Windows 3.x является наличие в SDK не одного файла заголовков windows.h, а множества заголовочных файлов. Их объем по сравнению с Windows 3.x вырос не менее чем на порядок. Изучайте их! Ответы на множество вопросов вы найдете только там! По возможности, упоминаемые макросы и значения приведены в виде таблиц. Как правило, таблицы состоят из трех колонок – макрос, числовое значение и описание. Это сделано для того, чтобы читатель смог сам определить, в каком виде ему использовать то или иное значение – в числовом (скажем, для использования в цикле) или в виде макроса. Если в таблице пропуск, то это означает, что у автора нет полной информации о том или ином макросе.

Автор обращает внимание читателя на одну из особенностей программирования в Win32 API. Windows нельзя знать частично. Даже самые первые программы уже требуют глубоких знаний. Поэтому читатель должен с пониманием отнестись к многочисленным ссылкам на последующие разделы книги типа «А об этом мы поговорим позже». На каком-то этапе чтения книги все станет на свое место.

Под словом «Win32 API» автор понимает совокупность функций, предоставляющих программисту возможность создавать программы (или приложения, что то же самое) для Windows NT и Windows'95. Естественно, что эти платформы разнятся между собой. Но выбор функций, составляющих API, для них один и тот же. Все функции этого набора являются 32-битными, что отражено в названии интерфейса. При употреблении термина «Win32 API» подразумевается именно набор функций. Когда в тексте встречается термин «Win32», читатель должен понимать, что речь идет о совокупности платформ (Windows NT и Windows'95), поддерживающих 32-битный интерфейс программирования. В тех случаях, когда говорится о Windows, автор говорит о двух упоминаемых выше операционных системах. Случаи упоминания Windows 3.x оговорены особо.

После выхода в свет первых двух изданий этой книги автор и издательство получили множество писем. Наряду с хорошими словами в адрес книги, читатели одновременно критиковали меня за то, что за рамками книги осталось множество вопросов, представляющих интерес для начинающих программистов в операционной системе Windows, и высказывали пожелания, чтобы будущие издания были дополнены описанием тех возможностей Windows, которые не были описаны в предыдущих изданиях. Конечно же, в одной книге нельзя описать всего. Тем не менее, я дополнил третье издание книги описанием тех возможностей Windows, которые не были упомянуты в предыдущих изданиях. И как всегда, читателям судить о том, получилась книга или нет.

GETTING STARTED - ДАВАЙТЕ НАЧНЕМ! «HELLO, WORLD!» ДЛЯ WIN32 API

После появления книги Кернигана и Ритчи «Язык программирования С» в мире программирования одним стандартом стало больше. С их легкой руки сейчас практически каждое руководство по обучению программированию начинается с написания программы, осуществляющей вывод на экран строки «Hello, world!». Я не буду нарушать эту традицию и изучение программирования для Win32 мы начнем с программы, выводящей строку «Hello, world!» не просто на экран, а в окно.

ЧТО НЕОБХОДИМО ДЛЯ ПОЛУЧЕНИЯ ИСПОЛНЯЕМОГО МОДУЛЯ?

Для получения исполняемого модуля необходимо иметь:

установленную на вашем компьютере операционную система Windows'95 или Windows NT;

систему подготовки программ для Win32 (автор для написания программ, приведенных в книге, использовал Borland C++ 5.0);

отладчик для отладки программ (автор пользовался Turbo Debugger'ом для 32-битовых программ, входящим в комплект поставки Borland C++ 5.0).

Для повседневной работы было бы неплохо иметь под рукой распечатки стандартных файлов заголовков, которые используются при программировании в Win32. С этим связана определенная трудность. Если в Windows 3.x был один файл заголовков «windows.h», то в Win32 число файлов возросло минимум на порядок. Соответственно, вырос и объем этих файлов. Сейчас он приближается к мегабайту.

ФАЙЛЫ ПРОГРАММЫ ДЛЯ WINDOWS

Перед началом любого дела необходимо представлять, с чего начать, что необходимо получить в результате и каким образом можно выполнить эту работу.

Желаемый результат очевиден - мы хотим получить программу для Windows NT или Windows'95, осуществляющую вывод в окно строки «Hello, world!». Но если в DOS для получения исполняемого файла нужен, как минимум, один файл исходного модуля, то в Windows дело обстоит несколько иначе. Как минимум, в Windows проект состоит из двух, а иногда - из трех файлов. Этими файлами являются:

программа на C/C++. Является основным файлом программы и, как правило, имеет расширение .C или .CPP. После его успешной компиля-

ции возникает файл с расширением .OBJ, который используется линкером для получения исполняемого модуля;

файл ресурсов. Иногда может не присутствовать в простом проекте. Как правило, имеет расширение .RC. После успешной компиляции его компилятором ресурсов возникает файл с расширение .RES, который используется линкером для получения исполняемого модуля;

файл определения модуля. Обычно имеет расширение .DEF и компиляции не подлежит. Используется линкером для определения некоторых характеристик исполняемого модуля. С появлением Win32 файл определения модуля почти не используется.

Для программиста, привыкшего к DOS, все это выглядит непривычно и громоздко. Тем не менее, в самом ближайшем будущем мы увидим, какие громадные возможности предоставляют файл ресурсов и файл определения модуля!

ТИПЫ ДАННЫХ, ПРИМЕНЯЕМЫЕ В WINDOWS

При первом взгляде на программу, написанную для Windows, бросается в глаза странный внешний вид этой программы. В программе используются переменные каких-то необычных типов, например, HINSTANCE, HWND, LPSTR и так далее. Разобраться в них совсем не сложно. Все они определены в заголовочных файлах Win32, общим «предком» которых является знаменитый «windows.h». Возникает закономерный вопрос: для чего были определены столько новых типов? Почему для определения переменных нельзя было воспользоваться стандартными типами, определенными в C/C++? Во-первых, что станет очевидно даже при небольшом опыте программирования для Win32, это очень удобно. Использование типов, специально «изобретенных» для Windows, упрощает написание программы, не заставляя запоминать последовательности многочисленных описаний, а применять один описатель. Во-вторых, программы, написанные с применением такого стиля, легко читаются и ошибки, связанные с типами переменных, легче обнаружить.

Возможно, истинная причина подобных нововведений лежит несколько глубже. Дело в том, что применение такого двухступенчатого определения типов (стандартный тип→заголовки Win32→программа) облегчает перенос программ для Windows в другие операционные системы. При переходе на новую систему достаточно будет изменить только файл заголовков. Изменять отлаженное программное обеспечение нет необходимости. Поэтому один и тот же код можно использовать в различных системах.

ВЕНГЕРСКАЯ НОТАЦИЯ

Помимо использования нестандартных описаний типов, при чтении программ для Windows можно заметить еще одну странность. Почти все идентификаторы начинаются с непонятных буквосочетаний. Но, оказывается, все очень просто. В любой книге, посвященной программированию под Windows, вы найдете упоминание о том, что один из первых разработчиков Windows Чарльз Симонаи, венгр по происхождению, начал использовать в своих программах способ именования переменных, который впоследствии назвали венгерской системой. Суть этой системы (крайне простой и потрясающе эффективной) можно определить несколькими правилами:

- каждое слово в имени переменной пишется с прописной буквы и слитно с другими словами, например, идентификатор для обозначения какой то переменной может выглядеть следующим образом - MyVariable, YourVariable, VariableForSavingAnotherVariable и т. д.;
- каждый идентификатор предваряется несколькими строчными буквами, определяющими его тип. Например, целая переменная MyVariable должна выглядеть как nMyVariable (n - общепринятая для целых переменных), символьная (char) переменная YourVariable превратиться в cYourVariable. Указатель на строку символов заканчивающуюся нулевым байтом, VariableForSavingAnotherVariable, pszVariableForSavingAnotherVariable (psz - сокращение от Point to String with Zero). Примеры подобных префиксов приведены в табл. 1.

Это позволяет упростить процесс чтения и понимания программ, а также делает переменные в некотором смысле самоопределенными - имя переменной определяется ее типом. Говорят, когда Симонаи спрашивали о странном внешнем виде его программ, он невозмутимо отвечал, что эти программы написаны по-венгерски.

WINDOWS КАК ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ СИСТЕМА

Когда вы начинаете программировать для Win32, необходимо уяснить, что хотя формально Windows не является объектно-ориентированной системой, она придерживается объектно- ориентиро- ванной идеологии. Каждое окно фактически является объектом. Что такое объект? Фактически объект есть совокупность полей (данных) и методов (процедур и функций управления полями). У окна есть масса полей, об этом мы еще будем говорить. Функция окна фактически является совокупностью методов.

Таблица 1. Префиксы, применяемые в венгерской нотации

Префикс	Тип данных
b	BYTE (unsigned char)
cx, cy	short (используются как ширина и длина объектов типа RECT и окон)
dw	DWORD (unsigned long)
fn	function
h	HANDLE
i	int
l	LONG (long)
n	int или short
s	string
sz	string terminated by zero
w	WORD (unsigned int)
x, y	short (используются как координаты)
c	char

«КРОВЕНОСНАЯ СИСТЕМА» ПРОГРАММЫ ДЛЯ WINDOWS

Для программиста, привыкшего к DOS, непривычной является и организация взаимодействия между операционной системой Windows (NT или 95) и другими программами. В DOS только программы обращаются к операционной системе. Обратной связи (вызов системой прикладной программы) в DOS нет (исключения типа перехвата прерываний не в счет). В Windows с момента создания окна и до момента его уничтожения не только программа обращается к Windows, но и самое операционная система при возникновении определенных событий обращается к окну, вызывая связанную с ним оконную процедуру, или, как говорят, посылая окну сообщение. Что значит послать окну сообщение? Это, значит, записать определенную информацию о каком-либо событии в область памяти, доступную оконной процедуре. Эта область памяти, которая вмещает в себя несколько сообщений, действует по принципу стека FIFO (First Input - First Output) и называется очередью программы. В Windows прикладная программа тоже вызывает систему не напрямую, а посылает сообщения системе. Но раз системе, как и прикладной программе, посылаются сообщения, то, значит, существует и общесистемная очередь сообщений! Итак, мы пришли к выводу о существовании **одной общесистемной очереди** сообщений и **очереди сообщений у каждого окна**.

Неясной остается одна деталь. Откуда система знает о том, что пришло сообщение? Каким образом сообщение из очереди становится из-

вестным программе? Вероятно, как программа, так и система с какой-то периодичностью обращаются к очереди и проверяют, нет ли в очереди сообщений. Здесь мы приходим ко второму выводу - **у каждой программы, а также и у системы должны существовать (и существуют!) циклы, в ходе которых опрашивается очередь и выбирается информация о сообщениях в ней**. Остановка цикла опроса очереди приведет к «зависанию» программы, программа «умрет», если сравнивать программу для Windows с человеческим организмом. Если продолжать сравнение, то будет видно, что сообщения протекают через функцию окна, как кровь по организму. Кровь в организме прокачивается сердцем, а сообщения «качаются» циклом обработки сообщений.

WINMAIN () + ФУНКЦИЯ ОКНА = МИНИМАЛЬНАЯ ПРОГРАММА ДЛЯ WINDOWS

Теперь мы уже знаем, что при запуске программы должны происходить, по меньшей мере, два события - должно быть создано окно и запущен цикл обработки сообщений, из которого с наступлением какого-то события должен быть осуществлен выход и работа программы должна завершиться. Все это происходит, как правило, в функции WinMain(), которая является стандартной точкой входа во все программы для Windows. (Обратите внимание - функция main() является точкой входа DOS'овских программ, функция WinMain() - программ, написанных для Windows). Для удобства функция окна отделена от WinMain() (к функции окна постоянно обращается система). Она может находиться либо в той же программе, что и WinMain(), либо вызываться из какой-либо библиотеки. Тем самым становится возможным создавать множество окон, использующих одну и ту же оконную функцию (другими словами, объектов, использующих одни и те же методы!), но имеющих разные характеристики (но имеющих разные значения полей!). А не напоминает ли это каким-то образом полиморфизм объектов? Попутно отмечу, что совокупность окон, использующих одну и ту же оконную функцию, представляет собой класс окон. Каждое окно принадлежит какому-либо классу. Примером такого класса могут быть кнопки, работающие совершенно одинаково, но имеющие разные размеры, надписи и так далее. Так что же получается? Мы сначала должны создать класс, а только потом создавать окно созданного класса? Да! Попробуем резюмировать сказанное.

На некотором псевдоязыке программу для Windows можно записать следующим образом:

```
WinMain (список аргументов)
 {
```

```
    Подготовка и создание класса окон с заданными характеристиками
    Создание экземпляра окна только что созданного класса;
    Пока не произошло необходимое для выхода событие
      Опрашивать очередь сообщений и передавать сообщения
      оконной функции;
    Возврат из программы;
  }

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

ПЕРВАЯ ПРОГРАММА ДЛЯ WINDOWS

Ниже приведен текст программы (вывод в окно строки, о ней говорилось ранее). Мне бы хотелось, чтобы читатель быстро просмотрел программу и попытался разделить ее на части, соответствующие операторам псевдоязыка. Вот текст этой программы:

```
#include <windows.h>

LRESULT CALLBACK HelloWorldWndProc ( HWND, UINT, UINT, LONG );
int WINAPI WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                     LPSTR lpszCmdParam, int nCmdShow )
  {
    HWND hWnd ;
    WNDCLASS WndClass ;
    MSG Msg;
    char szClassName[] = «HelloWorld»;
/* Регистрируем создаваемый класс */
/* Заполняем структуру типа WNDCLASS */
    WndClass.style = CS_HREDRAW | CS_VREDRAW;
    WndClass.lpfnWndProc = HelloWorldWndProc;
    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 = szClassName;

    if ( !RegisterClass(&WndClass) )
      {
```

```
        MessageBox(NULL,»Cannot register class»,»Error»,MB_OK);
        return 0;
      }

   hWnd = CreateWindow(szClassName, «Program No 1»,
                       WS_OVERLAPPEDWINDOW,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                       NULL, NULL,
                       hInstance, NULL);
   if(!hWnd)
      {
        MessageBox(NULL,»Cannot create window»,»Error»,MB_OK);
        return 0;
      }

/* Show our window */
   ShowWindow(hWnd,nCmdShow);
   UpdateWindow(hWnd);

/* Beginning of messages cycle */
   while(GetMessage(&Msg, NULL, 0, 0))
    {
      TranslateMessage(&Msg);
      DispatchMessage(&Msg);
    }
   return Msg.wParam;
 }

LRESULT CALLBACK HelloWorldWndProc (HWND hWnd, UINT Message,
                                    UINT wParam, LONG lParam )
 {
   HDC hDC;
   PAINTSTRUCT PaintStruct;
   RECT Rect;
   switch(Message)
    {
      case WM_PAINT:
        hDC = BeginPaint(hWnd, &PaintStruct);
        GetClientRect(hWnd,&Rect);
        DrawText (hDC,»Hello, World!», -1, &Rect,
                  DT_SINGLELINE | DT_CENTER | DT_VCENTER);
        EndPaint(hWnd,&PaintStruct);
        return 0;
      case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    }
   return DefWindowProc(hWnd,Message,wParam, lParam);
 }
```

Листинг № 1. Основной файл программы «Hello, world».

Предлагаю читателю откомпилировать основной файл программы. Для начинающих программистов проще всего воспользоваться интегрированной средой, например, Borland IDE. При этом не имеет значения, какой системой подготовки программ вы пользуетесь. Единственное, эта система должна позволять разработку программ для Win32. (В этой программе нет ничего, что присуще только Win32 и не присуще Windows 3.x. Но это только в первой программе.) Надеюсь, что эта программа пройдет у вас без ошибок с первого раза. Если же появятся ошибки, сверьте набранный вами текст модуля с приведенным в книге. По всей вероятности, вы допустили ошибку при наборе программы.

МИНИМУМ КОДА ПРИ МАКСИМУМЕ ВОЗМОЖНОСТЕЙ

На рис. 1 приведен результат работы этой программы. Естественно, что для простого вывода строки на экран эта программа велика. Но в том-то и состоит прелесть окна «Hello, world!», что оно обладает всеми характеристиками нормального окна, «умеет» изменять свой размер, минимизироваться (отображаться в виде иконки), максимизироваться (занимать все пространство экрана). У него есть системное меню в левом верхнем углу и три кнопки (максимизации, минимизации и закрытия) вверху справа. Окно может перемещаться по экрану. При изменении размеров окна строка «Hello, world!» автоматически перемещается в новый центр окна. Попробуйте в DOS достичь того же самого программой такого же объема! В DOS для достижения таких же результатов потребуются либо месяцы упорного труда для реализации собствен ной оконной библиотеки, либо придется использовать чужую оконную библиотеку, что приведет ко многим нежелательным эффектам, типа резкого увеличения объема исполняемого кода. С этой позиции объем «Hello, world!» кажется слишком компактным по сравнению с обычной программой, обладающей такой же функциональностью!

Теперь, когда мы увидели возможности «Hello, world!», попробуем разобрать ее построчно.

Практически каждая программа (а наша программа исключением не является) начинается со строки

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

Думаю, что к этому моменту строка в пояснениях уже не нуждается. В тело программы включается файл заголовков «windows.h».

Следом за строкой идет объявление оконной процедуры:

```
LRESULT CALLBACK HelloWorldWndProc (HWND, UINT, UINT, LONG);
```

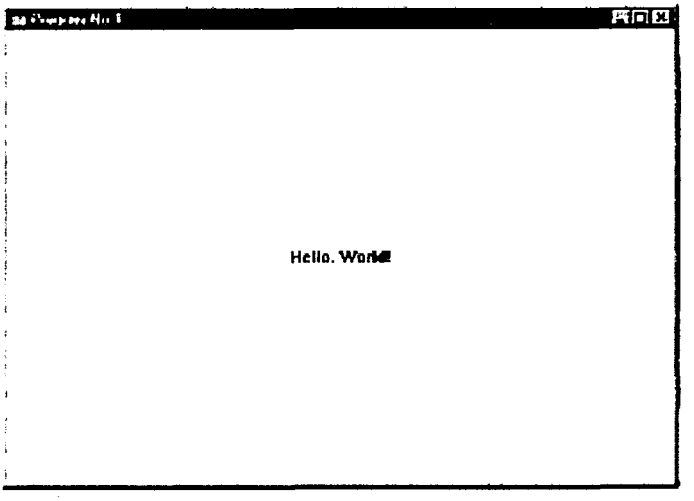


Рис. 1. Результат работы «HelloWorld»

Его мы разберем при рассмотрении непосредственно оконной функции.

Третьей строкой является функция WinMain(), о которой мы сейчас и поговорим.

ФУНКЦИЯ WINMAIN() И ЕЕ АРГУМЕНТЫ

С легкой руки автора одной из книг, посвященных программированию для Windows, функция WinMain() называется «стандартным заклинанием». Без этих или подобных строк не обходится почти ни одна программа. Как правило, программирующие для Windows хранят это «заклинание» в отдельном файле. В начале разработки нового проекта в этом файле просто изменяют несколько слов или строк - и функция WinMain() вновь готова к работе! Определение WinMain() всегда имеет вид, подобный следующему:

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

Сразу видно, что функция возвращает вызвавшей ее системе целое значение и ничего интересного этот момент собой не представляет. Следующая характеристика - WINAPI - определяет порядок передачи

параметров при вызове процедуры. Наименование характеристики говорит само за себя - WINdows Application Programming Interface - применяются соглашения о передаче параметров, принятые в системах Windows NT и Windows'95. Если вы не планируете писать приложения на ассемблере, вам нужно это просто запомнить.

А вот переменные hInstance и hPrevInstance заслуживают более подробного обсуждения. Так как Windows NT и Windows'95 являются многозадачными системами, то очевидно, что одна и та же программа может быть запущена несколько раз. Для того чтобы различать экземпляры программ, каждому экземпляру присваивается условный номер - хэндл (handle). Справедливости ради, надо отметить, что в Win32 присваиваются хэндлы чему угодно - окну, меню, курсору, иконке и т. д. Фактически хэндл - это указатель на блок памяти, в котором размещен тот или иной объект. В заголовочных файлах тип HANDLE определен как void*, а тип HINSTANCE как HANDLE. Согласно венгерской нотации, идентификаторы переменных типа HANDLE должны начинаться с буквы h.

Уважаемый читатель! Обратите внимание на вытекающее из этого положения следствие. Раз уж объект имеет хэндл, который является УКАЗАТЕЛЕМ, то, значит, этот объект сам расположен в памяти! Другими словами, в тех случаях, когда мы должны получить хэндл того или иного объекта, фактически мы должны получить адрес загруженного в память объекта!

Но вернемся к hInstance. Когда вызывается WinMain(), Windows через эту переменную сообщает программе хэндл экземпляра программы. В Windows 3.1 hPrevInstance являлся хэндлом предыдущего экземпляра программы. Если запускался первый экземпляр программы, то параметр hPrevInstance был равен нулю. Этот факт можно было использовать для того, чтобы не позволять системе запускать более одного экземпляра программы. В Win32 hPrevInstance оставлен ИСКЛЮЧИТЕЛЬНО для совместимости с предыдущими версиями Windows, он не несет никакой нагрузки и постоянно равен нулю. Так просто, как в более ранних версиях Windows, определить наличие ранее запущенного экземпляра программы не удастся. Придется нам и этот вопрос оставить на потом, до изучения основ многозадачности Windows.

Следующий параметр - pszCmdLine - представляет собой указатель на строку, ту командую строку, которая набирается после имени запускаемой программы. При необходимости программа может проанализировать этот аргумент и выполнить те или иные действия.

И последний параметр - nCmdShow - определяет, в каком виде создаваемое окно будет появляться на экране. Окно может появляться в максимизированном виде либо в виде иконки (минимизированном),

может иметь произвольный размер, определяемый программой и другие характеристики. В Win32 API определяются десять возможных значений этого параметра. Их идентификаторы начинаются с SW (вероятно, от названия функции ShowWindow, которая использует эти значения). Наиболее часто используются значения SW_SHOWNORMAL и SW_SHOWMINNOACTIVE. Возможные значения этого параметра приведены в табл. 2. Большинство идентификаторов являются самоопределенными (вряд ли, скажем, SW_SHOWMAXIMIZED приводит к отображению окна в виде иконки!). Вы можете поэкспериментировать с ними. Их полное описание можно найти в файлах системы помощи. Теперь вспомним, что перед созданием окна мы должны сначала определить его класс, поэтому у нас на очереди

Регистрация класса окна

Сразу после входа в WinMain() нам необходимо создать класс окна и сообщить о нем системе. Класс создается и регистрируется функцией **RegisterClass()**. Единственным аргументом этой функции является указатель на структуру типа **WNDCLASS**, в которой хранятся характеристики создаваемого класса. Из этого следует, что у нас добавилось головной боли - перед регистрацией класса заполнить процедуру типа WNDCLASS. В приведенной выше программе структура была определена следующим образом:

```
WNDCLASS WndClass;
```

Таблица 2. Возможные значения второго параметра функции ShowWindow()

Параметр	Значение	Параметр	Значение
SW_HIDE	0	SW_SHOWNOACTIVE	4
HIDE_WINDOW	0	SHOW_OPENNOACTIVE	4
SW_SHOWNORMAL	1	SW_SHOW	5
SW_NORMAL	1	SW_MINIMIZE	6
SHOW_OPENWINDOW	1	SW_SHOWMINNOACTIVE	7
SW_SHOWMINIMIZED	2	SW_SHOWNA	8
SHOW_ICONWINDOW	2	SW_RESTORE	9
SW_SHOWMAXIMIZED	3	SW_SHOWDEFAULT	10
SHOW_FULLSCREEN	3	SW_MAX	10
SW_MAXIMIZE	3		

Не забывайте, что в языке C, в отличие, скажем, от PASCAL'я, прописные и строчные буквы различаются. Для того чтобы заполнить эту структуру, нам необходимо знать тип и назначение каждого ее поля. Посмотрим, как описана эта структура в заголовочных файлах (winuser.h):

```
typedef struct tagWNDCLASSA {
    UINT        style;
    WNDPROC     lpfnWndProc;
    int         cbClsExtra;
    int         cbWndExtra;
    HINSTANCE   hInstance;
    HICON       hIcon;
    HCURSOR     hCursor;
    HBRUSH      hbrBackground;
    LPCSTR      lpszMenuName;
    LPCSTR      lpszClassName;
} WNDCLASSA, *PWNDCLASSA, NEAR *NPWNDCLASSA,
  FAR *LPWNDCLASSA;
typedef struct tagWNDCLASSW {
    UINT        style;
    WNDPROC     lpfnWndProc;
    int         cbClsExtra;
    int         cbWndExtra;
    HINSTANCE   hInstance;
    HICON       hIcon;
    HCURSOR     hCursor;
    HBRUSH      hbrBackground;
    LPCWSTR     lpszMenuName;
LPCWSTR     lpszClassName;
} WNDCLASSW, *PWNDCLASSW, NEAR *NPWNDCLASSW,
  FAR *LPWNDCLASSW;
#ifdef UNICODE
typedef WNDCLASSW WNDCLASS;
typedef PWNDCLASSW PWNDCLASS;
typedef NPWNDCLASSW NPWNDCLASS;
typedef LPWNDCLASSW LPWNDCLASS;
#else
typedef WNDCLASSA WNDCLASS;
typedef PWNDCLASSA PWNDCLASS;
typedef NPWNDCLASSA NPWNDCLASS;
typedef LPWNDCLASSA LPWNDCLASS;
#endif // UNICODE
```

О том, почему структура объявляется так странно и что такое Unicode, мы поговорим в разделе, посвященном Unicode. А пока давайте считать, что это просто разные описания одной и той же структуры.

Особое внимание следует обратить на первое, второе и последнее поля. Почему? - Сейчас станет ясно.

Стиль окна я определяю оператором

WndClass.style = CS_HREDRAW | CS_VREDRAW

В winuser.h описаны тринадцать стилей окна. Наименования их идентификаторов начинаются с CS, что, вероятно, означает «Class style». Для стиля окна отведено 16 битов и только один из этих битов установлен в единицу. Другими словами, стили, упомянутые в winuser.h, используются как битовые флаги, т. е. с этими стилями можно производить операции логического сложения и логического умножения для получения комбинированных стилей. Перечень флагов приведен в табл. 3.

Я отдаю себе отчет в том, что сейчас описания многих флагов совершенно непонятны, но через несколько разделов все будет ясно.

По причине, известной только Microsoft, отсутствуют стили со значениями 0x0010 и 0x0400. Те флаги, которые использует программа «Hello, world!», означают, что окну необходимо полностью перерисоваться (запомните это слово! О перерисовке мы еще не раз вспомним!) при изменении его размеров по горизонтали и по вертикали.

Попробуйте поиграть с размерами окна и сделать так, чтобы строка появилась не в середине! Надеюсь, вам это не удастся. Как бы вы не дергали его, текст постоянно будет оставаться в центре экрана.

Не будет преувеличением сказать, что второе поле структуры WNDCLASS особенно значимо, - в нем хранится указатель на оконную функцию создаваемого в программе класса окон. Эта функция будет производить обработку абсолютно всех сообщений, получаемых окном, другими словами, значение этого поля полностью определяет все поведение окна. В программе это поле инициализируется следующим образом:

```
WndClass.lpfnWndProc = HelloWorldWndProc;
```

Два следующих поля оставлены нулевыми. Дело в том, что для каждого класса Windows создает где-то в недрах своей памяти структуру с характеристиками класса. Другая структура создается для каждого окна. При создании этих структур может быть зарезервировано некоторое количество памяти для нужд программиста. Поля **cbClsExtra** и **cbWndExtra** указывают размер резервируемой памяти в структурах класса и окна соответственно. Эти поля и раньше использовались достаточно редко, а с появлением Windows'95 и Windows NT будут использоваться еще реже.

Таблица 3. Перечень битовых флагов

Флаг	Значение	Описание
CS_VREDRAW	0x0001	Перерисовать окно при изменении высоты окна
CS_HREDRAW	0x0002	Перерисовать окно при изменении ширины окна
CS_KEYCVTWINDOW	0x0004	
CS_DBLCLKS	0x0008	Посылать сообщение оконной функции при двойном щелчке мышью, если курсор находится в пределах окна
CS_OWNDC	0x0020	Для каждого окна класса выделяется собственный контекст
CS_CLASSDC	0x0040	Один и тот же контекст устройства разделяется всеми окнами этого класса
CS_PARENTDC	0x0080	Дочерние окна наследуют контекст родительского окна
CS_NOKEYCVT	0x0100	
CS_NOCLOSE	0x0200	Убрать команду «Close» из системного меню
CS_SAVEBITS	0x0800	Сохранять часть области экрана, закрытую окном, как bitmap, при удалении восстанавливать перекрытую область
CS_BYTEALIGNCLIENT	0x1000	Выравнивает границу рабочей области окна (в горизонтальном направлении) таким образом, чтобы для отображения строки требовалось целое число байтов
CS_BYTEALIGNWINDOW	0x2000	То же, но действие затрагивает все окно
CS_GLOBALCLASS	0x4000	Разрешается создавать класс, не зависящий от текущего hInstance
CS_IME	0x00010000L	

Поле hInstance в объяснении не нуждается - классу окна сообщается хэндл программы.

Оператор

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

определяет хэндл иконки, которая будет символом окна данного класса. Действие, производимое функцией LoadIcon(), очевидно из ее названия - загрузить иконку. Заметим, что программист может использовать собственную иконку, которую он сам разработал, а может применить одну из иконок, хранящихся в глубинах системы (они называются предопределенными). В случае использования собственной иконки первый параметр функции LoadIcon() должен быть равным хэндлу программы (hInstance). Если мы используем предопределенную иконку, первый параметр равен

нулю (забегая вперед, отметим, что если при загрузке в память какого-либо объекта хэндл программы равен нулю, то объект загружается либо из «глубин» Windows, либо из внешнего файла). Второй параметр - это идентификатор иконки. Все идентификаторы предопределенных иконок начинаются с букв IDI (возможно, «IDentificator of Icon»). Пока еще мы не знаем, как формировать иконки, воспользуемся одной из предопределенных иконок.

Сказанное об иконке можно полностью отнести и к курсору мыши, которым будут пользоваться окна создаваемого класса (не путать с курсором, применяемым при редактировании текстовых файлов). Поле **WndClass.hCursor** определяет хэндл курсора. Все идентификаторы предопределенных курсоров начинаются с IDC (возможно, «IDentificator of Cursor»).

Поле **WndClass.hbrBackground** определяет хэндл та называемой кисти (brush), которой будет закрашен фон окна.

К иконкам, курсорам, кистям и перьям мы еще неоднократно будем возвращаться. А сейчас неплохо было бы попробовать поменять идентификаторы иконок, курсоров, кистей и посмотреть, к чему это приведет. Для этого в табл. 4 приведен список объектов этих типов, индификаторы которых я нашел.

Поле **WndClass.lpszMenuName** хранит указатель на строку, содержащую имя меню для данной программы. Наша программа с меню не работает, поэтому мы сделали его нулевым.

И последнее, завершающее поле - **WndClass.lpszClassName**. Как явствует из его названия, поле содержит указатель на строку, содержащую имя создаваемого нами класса окна. Указав это имя, мы тем самым поставили логическую точку в формировании структуры WNDCLASS.

Указатель на эту структуру передается функции **RegisterClass()**. С вызовом этой функции, данные о создаваемом нами классе становятся известными системе Windows, и с этого момента мы можем создавать окна этого класса. Пожалуйста, не забывайте в своих программах проверять, зарегистрировался класс или нет. Если класс не зарегистрирован, работать ваша программа не будет, как бы правильно она не была написана. В нашей программе, в случае, если класс окна не зарегистрирован, просто выдается сообщение об ошибке (функция MessageBox()) и осуществляется выход из программы. Кстати, мы уже говорили о предопределенных иконках и курсорах? В Win32 API существует множество предопределенных классов окон, например класс кнопок, списков и т. д. При необходимости создания окна предопределенного класса регистрировать класс окна уже не нужно.

Таблица 4. Список предопределенных объектов в Win32 API

Иконка	Перо	Курсор	Кисть
IDI_APPLICATION IDI_HAND IDI_QUESTION IDI_EXCLAMATION IDI_ASTERISK IDI_WINLOGO IDI_WARNING IDI_ERROR IDI_INFORMATION	WHITE_PEN BLACK_PEN NULL_PEN	IDC_ARROW IDC_IBEAM IDC_WAIT IDC_CROSS IDC_UPARROW IDC_SIZE IDC_ICON IDC_SIZENWSE IDC_SIZENESW IDC_SIZEWE IDC_SIZENS IDC_SIZEALL IDC_NO IDC_APPSTARTING IDC_HELP	WHITE_BRUSH LTGRAY_BRUSH GRAY_BRUSH DKGRAY_BRUSH BLACK_BRUSH NULL_BRUSH HOLLOW_BRUSH

Создание экземпляра окна

Следующим шагом на нашем большом пути является создание экземпляра окна. Как и экземпляр программы, каждое окно в системе имеет свой уникальный номер хэндл (handle). Обычно окно создается посредством функции **CreateWindow()**, которая и возвращает хэндл созданного окна. Если (увы!) функция CreateWindow() вернула нуль, то по каким-то причинам окно не создано. Причины могут быть как внутри вашей программы, так и в системе. Но, в отличие от регистрации класса, о том, что окно не создано, вы можете узнать, просто взглянув на экран. Теперь подошло время рассказать о каждом из одиннадцати аргументов функции CreateWindow().

Первый аргумент - указатель на строку с именем класса, к которому будет принадлежать создаваемое нами окно. В большинстве случаев значение этого аргумента совпадает со значением последнего поля структуры типа WNDCLASS, передаваемой RegisterClass() (Может быть целесообразно использовать одну и ту же переменную в функциях RegisterClass() и CreateWindow()?). Второй аргумент - указатель на строку, содержащую тот текст, который появится в заголовке окна.

Третий аргумент определяет стиль окна (не общие характеристики всех окон класса, а индивидуальные характеристики конкретного окна). Стиль определяет, будет ли окно иметь заголовок, иконку системного меню, кнопки минимизации, максимизации, характер границы окна, определяет также взаимоотношения окон типа предок-потомок и т. д. Под

это поле отводится 32 бита. В файле winuser.h определены несколько десятков стилей окон. Их идентификаторы начинаются с букв WS. Как и в случае со стилями класса, эти значения используются как битовые флаги, т. е. комбинируя их с помощью логических операций, можно получить тот стиль окна, который требуется нам в программе. Рекомендую поэкспериментировать с различными стилями окна. Их список приведен в табл. 5.

Некоторые стили, приведенные в winuser.h, представляют собой комбинации из других стилей. В частности, тот стиль, который используем мы, WS_OVERLAPPEDWINDOW, тоже является комбинацией. Выбирая этот стиль, мы определяем наличие у нашего окна заголовка, системного меню, ограничивающей рамки, а также кнопок минимизации и максимизации.

Следующие четыре аргумента определяют положение окна на экране. Значение этих полей представляют измеренные в пикселах отступы левого верхнего угла окна от левого края экрана, от верхней границы экрана, ширину и высоту окна соответственно. Особых пояснений эти параметры не требуют. Используемые нами идентификаторы CW_USEDEFAULT, допустимые, кстати, только для окон со стилем WS_OVERLAPPED, позволяют Win32 API установить размер окна самостоятельно.

На очереди следующий аргумент - хэндл окна, являющимся родительским по отношению к нашему. Пока оставим это поле нулевым, а о взаимоотношениях окон типа предок - потомок - сосед мы узнаем, когда будем изучать иерархию окон.

Windows присваивает хэндлы чему угодно, в том числе и меню. Очередной аргумент - это хендл меню нашего окна. До разговора о меню в нашей программе меню не будет, поэтому оставим его нулевым.

Предпоследний аргумент - hInstance - должен быть понятен из предыдущих объяснений. Да-да, именно тот самый хэндл экземпляра программы, который мы запускаем.

Последний аргумент - данные, которые используются в некоторых случаях для создания окна. Как правило, в это поле записывается указатель на структуры с дополнительной информацией. Для того чтобы добраться до него, мы потратили столько сил! А оно используется достаточно редко и в нашем примере, естественно, остается без дела.

Напоминаю: не забудьте в программе проверить факт создания окна и отреагировать на него соответствующим образом! Справедливости ради, в одном из примеров, поставляемых с Borland C++ v. 5.0, все эти проверки называются параноидальными, но я пришел к выводу, что на этапе отладки программы лучше все эти проверки оставить. Когда программа заработает полностью - это дело другое.

Таблица 5. Список различных стилей окна

Стиль	Значение	Описание
WS_OVERLAPPED	0x00000000L	Окно имеет заголовок и обрамляющую рамку
WS_TABSTOP	0x00010000L	Окно может получать клавиатурный фокус при нажатии пользователем клавиши Tab
WS_MAXIMIZEBOX	0x00010000L	У окна есть кнопка максимизации
WS_GROUP	0x00020000L	Окно является первым окном группы
WS_MINIMIZEBOX	0x00020000L	У окна есть кнопка минимизации
WS_THICKFRAME	0x00040000L	У окна есть достаточно толстая рамка, позволяющая окну изменять размеры, используется в основном для окон диалога. Заголовка у окна нет
WS_SYSMENU	0x00080000L	У окна есть системное меню
WS_HSCROLL	0x00100000L	У окна есть горизонтальная линейка прокрутки
WS_VSCROLL	0x00200000L	У окна есть вертикальная линейка прокрутки
WS_DLGFRAME	0x00400000L	У окна есть рамка, которая обычно бывает у диалоговых окон
WS_BORDER	0x00800000L	У окна есть тонкая ограничивающая рамка
WS_CAPTION	0x00C00000L	WS_BORDER | WS_DLGFRAME
WS_MAXIMIZE	0x01000000L	Создается изначально максимизированное окно
WS_CLIPCHILDREN	0x02000000L	При прорисовке родительского окна область, занимаемая дочерними окнами, не прорисовывается
WS_CLIPSIBLINGS	0x04000000L	Дочернее окно, имеющее этот стиль, и перекрывающее другое дочернее окно, при прорисовке перекрываемой области не прорисовывается
WS_DISABLED	0x08000000L	Создается изначально запрещенное окно
WS_VISIBLE	0x10000000L	Создается изначально отображаемое окно
WS_MINIMIZE	0x20000000L	Создается изначально минимизированное окно
WS_CHILD	0x40000000L	Создается дочернее окно, имеющее по умолчанию только рабочую область, меню окна этого стиля не имеют никогда
WS_POPUP	0x80000000L	Создается всплывающее (popup) окно
WS_TILED		WS_OVERLAPPED
WS_ICONIC		WS_MINIMIZED
WS_SIZEBOX		WS_THICKFRAME
WS_TILEDWINDOW		WS_OVERLAPPEDWIINDOW
WS_OVERLAPPED-WINDOW		WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_THICKFRAME | WS_MINIMIZEBOX | WS_MAXIMIZEBOX
WS_POPUP-WINDOW		WS_SYSMENU | WS_BORDER | WS_POPUP
WS_CHILDWINDOW		WS_CHILD

Но что значит создать окно? Это, значит, создать внутренние для Win32 API структуры, которые будут использоваться в течение всего

периода существования окна. Но это не означает, что окно тут же появится на экране. Для того чтобы увидеть окно, мы должны осуществить

Отображение окна на экране

Теперь мы подошли к функции ShowWindow(). Функция отображает окно на экране (отметьте - отображает окно как набор атрибутов, таких как заголовок, рамка, кнопки и т. д.). Первый аргумент этой функции - хэндл созданного только что окна. Второй аргумент определяет, в каком виде окно будет отображено на экране. В нашем случае мы просто взяли и подставили значение nCmdShow, указанное при вызове WinMain(). Как правило, при первом запуске окна функции WinMain() передается значение SW_SHOWDEFAULT, при последующих запусках значение этого параметра может изменяться в соответствии со сложившимися обстоятельствами. Я не рекомендую указывать в качестве второго параметра функции ShowWindow() значение, отличное от передаваемого при вызове WinMain(). Тем самым вы лишите Win32 API некоторых возможностей по управлению окном.

Функция UpdateWindow() посылает функции окна сообщение WM_PAINT, которое заставляет окно ПЕРЕРИСОВАТЬСЯ, т. е. прорисовать не набор атрибутов, за прорисовку которых отвечает Windows, а изображение в рабочей области окна, за что должна отвечать непосредственно программа.

Итак, класс окна зарегистрирован, экземпляр окна создан и выдан на отображение. На очереди -

Запуск и завершение цикла обработки сообщений

Мы уже говорили, что по аналогии с человеческим организмом, сообщения играют роль крови, а цикл обработки - роль сердца, которое эту кровь прокачивает.

Что такое сообщение? Сообщение - это небольшая структура, определенная в заголовочном файле следующим образом:

```
typedef struct tagMSG {
  HWND      hwnd;
  UINT      message;
  WPARAM    wParam;
  LPARAM    lParam;
  DWORD     time;
  POINT     pt;
} MSG, *PMSG, NEAR *NPMSG, FAR *LPMSG;
```

Первое поле этой структуры вопросов не вызывает - это хэндл окна, которому адресовано сообщение. Второе поле - номер сообщения. Каждое сообщение имеет свой идентификатор. Все идентификаторы сообщений начинаются с букв WM, что, возможно, означает «Windows Message». Третье и четвертое поля содержат параметры сообщения. Для каждого сообщения они различны. Назначение поля time в объяснении не нуждается - оно очевидно из названия. В последнее поле (pt) записывается позиция, на которой находился курсор в момент выработки сообщения. Все эти поля могут обрабатываться оконной процедурой.

Из очереди приложений сообщение выбирается с помощью функции GetMessage(). Первый аргумент этой функции - указатель на структуру типа MSG, в которую будет записана информация о сообщении. Второй - хэндл окна, созданного программой. Сообщения, адресованные только этому окну, будут выбираться функцией GetMessage() и передаваться оконной функции. Если вы хотите, чтобы обрабатывались сообщения для всех созданных программой окон, установите этот параметр равным NULL.

Третий и четвертый параметры позволяют передавать оконной функции не все сообщения, а только те, номера которых попадают в определенный интервал. Третий параметр - нижняя граница этого интервала, четвертый - верхняя граница.

Функция GetMessage() всегда, за исключением одного случая, возвращает ненулевое значение. Исключение в данном случае очевидно - при получении этого единственного сообщения работа цикла, а следовательно, и программы, прекращается. Это сообщение называется WM_QUIT и приводит к нормальному завершению программы. Но давайте пока оставим WM_QUIT в покое и посмотрим, что происходит внутри цикла при получении им нормального - не WM_QUIT - сообщения.

Давайте на время представим себе, что для унификации процесса обработки сообщений некоторые из них необходимо преобразовать в более удобный для обработки вид. Эти преобразования выполняет функция TranslateMessage(). К этой функции мы еще вернемся при изучении системы обработки ввода с клавиатуры.

После преобразования сообщение готово к обработке. Функция DispatchMessage() передает сообщение на обработку в оконную процедуру.

ВСЕ! Мы сформировали структуру типа WNDCLASS, зарегистрировали класс окна и создали экземпляр окна этого класса. Мы выдали окно на отображение, после чего запустили цикл обработки сообщений и предусмотрели условие, при котором работа цикла закончится. Теперь все зависит от того, как сообщения будут обрабатываться оконной про-

цедурой. Функция WinMain (), если, конечно, не считать работающего цикла обработки сообщений, на этом свою задачу выполнила.

ФУНКЦИЯ ОКНА И ЕЕ АРГУМЕНТЫ

Мы закончили изучение функции WinMain. Теперь нашей ближайшей задачей будет рассмотрение оконной процедуры.

Как обработать множество различных сообщений?

Первый вопрос, возникающий при изучении функции окна, касается реакции этой функции на огромное число сообщений. Ведь в файле заголовков прямо записано, что Майкрософт резервирует для использования в Win32 все сообщения с номерами менее 1024! Неужели функция окна должна обрабатывать все эти сообщения и в программе должен быть код, обеспечивающий эту обработку? Ответ на этот вопрос достаточно парадоксален. Да, оконная функция должна обрабатывать все сообщения, приходящие в адрес окна. Нет, программист не должен писать кода для обработки всех сообщений! Дело в том, что в Windows предусмотрена обработка всех сообщений по умолчанию. В программе должен присутствовать код обработки только тех сообщений, обработка которых по умолчанию не устраивает программу. Все остальные «протекают» сквозь функцию и передаются на обработку по умолчанию. Делается это простым обращением к функции DefWindowProc (Default Window Procedure). Именно эта функция, спрятанная глубоко в «недрах» Windows, производит обработку подавляющего большинства сообщений, получаемых окном. Обратите внимание на предпоследнюю строчку текста программы. В этой строке все необработанные сообщения передаются процедуре обработки по умолчанию. На долю оконной процедуры остается «самая малость» - обработать сообщения, которые нуждаются в нестандартной обработке. В нашей программе их два - WM_PAINT и WM_DESTROY.

О сообщении WM_PAINT стоит сказать особо. В большинстве оконных библиотек при создании окна в памяти формировался буфер, в который записывалось то, что отображалось на закрываемой окном части экрана. Содержимое менялось при изменении положения и размеров окна. При этом, кстати, одно окно (перекрывающее) управляло отображением другого (перекрываемого). И, если этот подход был приемлемым для текстового режима, то можно представить, сколько памяти «пожирали» бы подобные буферы в графических режимах! В Windows пошли по другому пути. В Windows каждое окно «знает» о том, что оно

должно проделать в тех случаях, когда условия его отображения изменились. К этим случаям я отношу изменение размеров окна, его положения на экране, восстановления на экране после полного или частичного перекрытия другим окном. В этих случаях окну посылается сообщение WM_PAINT, говорящее о том, что окно должно **само** перерисоваться полностью или частично, обработав сообщение WM_PAINT. Таким образом, помимо сокращения объема требуемой памяти (намного легче хранить небольшую процедуру обработки сообщения WM_PAINT, чем буфер с графической информацией), в Windows решается ее одна проблема - выполняется одно из требований объектно-ориентированного программирования - полями объекта управляют методы того же (и только того же) объекта. Таким образом, мы пришли еще к одному важному выводу - каждое окно должно обрабатывать сообщение WM_PAINT, иначе оно не сможет восстановить изображение в своей рабочей области.

Упомяну еще одно сообщение, которое в нашей программе не присутствует, но которое иногда бывает очень полезным. Дело в том, что это сообщение окно получает после создания, но до отображения, точнее, до прорисовки рабочей области. Оно называется WM_CREATE и обычно используется для инициализации окна. В некотором смысле оно является антиподом сообщения WM_DESTROY, применяемого для деинициализации окна.

Громадный switch

Фактически вся оконная процедура состоит из одного единственного оператора switch. Он пользуется недоброй славой и, наверное, заслуженно. Иногда человек просто не в силах осмыслить многочисленные case'ы, внутри которых спрятаны очередные switch'и и т. д. По ходу изучения мы увидим, что написаны макрокоманды, позволяющие отказаться от громадного switch первого уровня. Но пока давайте придерживаться «классического» стиля. При написании небольших программ оператор switch очень наглядно (почти графически!) показывает ход обработки сообщений. На данном этапе я не буду рассматривать оконную процедуру столь же подробно, как и WinMain(). Обработка сообщения WM_PAINT просто приводит к выводу сообщения на экран. До этого сообщения мы еще дойдем. Структура оконной процедуры ясна практически любому, хоть немного знакомому с языком С. Остановимся на моменте, касающемся завершения работы программы.

WM_DESTROY и WM_QUIT

При необходимости закрыть окно, Windows дает окну возможность «осмотреться» и провести процедуру деинициализации. За счет чего это достигается? В ходе закрытия окна (я напоминаю, что окно - это не только прямоугольная область, видимая на экране, но и совокупность структур данных в глубине системы) сразу после снятия его с отображения оконная функция получает сообщение WM_DESTROY, которое является сигналом о необходимости произвести процедуру деинициализации. Получив это сообщение и произведя все необходимые действия, функция окна, как правило, вызывает функцию PostQuitMessage(), которая, как следует из ее названия, посылает окну сообщение WM_QUIT, которое, в свою очередь, попав в цикл обработки сообщений, вызывает его прекращение. А посему - ура! Мы прошли путь от начала до завершения программы. Мы узнали достаточно много о структуре программы для Windows, научились регистрировать классы окон, создавать экземпляры окон зарегистрированного класса, запускать и прекращать цикл обработки сообщений. Мы получили первоначальные знания об обработке сообщений и написании оконной процедуры.

Следующим шагом в разработке полноценной программы является подключение к программе ресурсов. В последующих главах мы остановимся на этом. Это еще на шаг приблизит нас к концу пути. Но на этом пути есть несколько камней, которые нам придется убрать с дороги.

Давайте первым уберем камень под названием

UNICODE

Если говорить честно, то я не думаю, что в настоящее время понимание Unicode является определяющим фактором для программиста. Дело в том, что в дальнейшем нам часто будут встречаться описания функций, зависящих от Unicode, а при изучении элементов управления, возможно, придется самим создавать строки в коде Unicode. Для того чтобы исключить в будущем лихорадочные метания читателя в поисках сведений об этом непонятном Unicode, я решил поместить в книгу небольшую обзорную главу.

ЧТО ТАКОЕ UNICODE

В обычной кодировке ANSI каждый символ определяется восемью битами, т. е. всего допускается 256 символов, что, конечно, очень и очень

мало. Каждому известно, что при русификации компьютер приобретает символы кириллицы, но теряет некоторые стандартные символы, определенные в коде ANSI. В некоторых алфавитах, например, в японской кане, столько символов, что одного байта для их кодировки просто недостаточно. В таких случаях приходится искусственно вводить всякие условности (типа двухбайтовых наборов символов, double byte character sets - DBCS) и представлять символы то одним, то двумя байтами. К чему мог привести программиста этот кошмар! Для поддержки таких языков, и, в свою очередь, для облегчения «перевода» программ на другие языки, была создана кодировка Unicode.

Каждый символ в Unicode состоит из двух байтов. С одной стороны, это позволяет преодолеть все сложности по искусственному представлению символов двумя байтами. С другой стороны, это позволяет расширить набор допустимых символов до 65 536 символов. Разница с 256 символами ANSI достаточно ощутима, не так ли?

Windows NT - это первая операционная система, полностью построенная на Unicode. Если функции передается ANSI-строка, она преобразуется в Unicode. Если программа пользователя ожидает результат в виде ANSI-строки, то перед возвращением строка Unicode преобразуется в ANSI.

К сожалению, Windows'95, которая выросла из 16-битной Windows 3.x, поддерживает Unicode не в полной мере. Поэтому внутри Windows'95 вся обработка идет в ANSI коде, хотя допускается написание приложений, поддерживающих Unicode. Тем не менее, уважаемый читатель, если вы захотите перейти к работе с Unicode, то придется изучить эту кодировку и способы работы с ней.

UNICODE В WINDOWS NT И WINDOWS'95

Win 32 в большинстве случаев позволяет работу с символами как в традиционной ANSI кодировке, так и в кодировке Unicode. Почти все функции, получающие в качестве аргумента символы или строки, имеют ANSI и Unicode версии. Программист может выбрать, с каким набором символов будет работать его программа. Более того, программист может написать программу таким образом, что она сможет быть откомпилирована для работы как с ANSI, так и с Unicode. Для этого программисту достаточно знать два-три макроса, несколько новых типов данных, и, естественно, новые наименования некоторых хорошо знакомых функций. Но обо всем по порядку.

РАБОТА С UNICODE ПРИ ИСПОЛЬЗОВАНИИ C/C++ RUN-TIME LIBRARY

Все функции RTL (Run-Time Library - библиотека времени выполнения), работающие со строками и символами, имеют ANSI- и Unicode-версии. Unicode-версии функций используют новый тип данных, введенный для описания символов Unicode. Этот тип описан следующим образом:

```
typedef unsigned short wchar_t;
```

ANSI-версии, которые по старинке применяют старый добрый char, используют те имена функций, к которым привыкли программисты на языке C. В то же время, имена функций, использующих Unicode, не совпадают с привычными нам старыми именами типа printf(), strcat() и т. д. Для того чтобы написать приложение, которое легко адаптировалось бы к обеим кодировкам, нужно уметь манипулировать именами функций и типами данных. Принцип понятен - условная компиляция. В RTL для того, чтобы определить, какую версию программы строить, введен символ препроцессора _UNICODE. В зависимости от того, определен этот символ или нет, строится та или иная версия программы.

Кроме этого, вместо файла string.h используется файл tchar.h, который обеспечивает универсальность. В нем определен громадный список макросов, которые необходимо использовать для того, чтобы препроцессор знал, какой набор функций ему необходимо вызывать, ANSI или Unicode. Этот список макросов приведен в приложении. Предлагаю читателю обратить внимание на то, что для написания кода, который мог бы компилироваться как для ANSI, так и для Unicode, необходимо вместо функций, приведенных в правых колонках, использовать имена, приводимые в левых колонках.

Для того чтобы указать препроцессору, как нужно строить компилируемый файл, применяется символ _UNICODE. Этот тип данных используется при работе с символами Unicode. Для того чтобы писать приложения, работающие как с ANSI, так и с Uncode, пррименяется другой макрос - TCHAR, который в зависимости от факта определения _UNICODE определяется либо

```
typedef wchar_t TCHAR;
```

либо

```
typedef unsigned char TCHAR.
```

Таким образом, используя этот тип, мы можем объявлять данные, которые будут восприниматься в разных обстоятельствах то как ANSI-, то как Unicode-строки или символы. Например, строка

```
TCHAR* pszMyString = «This is my string»;
```

в зависимости от того, определен ли символ _UNICODE, будет считаться либо состоящей из символов ANSI и занимать 18 байтов, либо состоящей из символов ANSI и занимать памяти в два раза больше.

Теперь возникает проблема с компилятором. По умолчанию, компилятору почему-то наплевать (извините за такое слово, но я долго не мог понять, в чем же состоит моя ошибка при определении строки), что мы описываем строку как состоящую из символов Unicode. Для него если строка, так уж обязательно ANSI! Попробуйте откомпилировать следующую «программу»:

```
#define _UNICODE
#include <windows.h>
#include <tchar.h>

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow )
 {
  TCHAR* pszMyString = "This is my string";
  return 0;
 }
```

Кажется, что все сделано правильно, но попробуйте посмотреть полученный .exe-файл обычным ..., в ... е. Вы увидите «This is my string» в обычной ANSI-кодировке, т. е. определение _UNICODE никакого влияния на представление строки не оказывает. Таким образом, необходимо явно объявлять строку, как состоящую из символов Unicode. Попробуйте внести небольшое изменение.

```
TCHAR* pszMyString = L"This is my string";
```

Буква L перед строкой указывает компилятору, что строка состоит из символов Unicode. В .exe-файле мы получим символы Unicode. Но в таком случае мы не можем получить ANSI-строку! Замкнутый круг! Проблема разрешается введением еще одного макроса - _TEXT. Определен он примерно так:

```
#ifdef _UNICODE
  typedef _TEXT(x) L ## x
#else
  typedef _TEXT(x) x
```

Пользуясь этим макросом, можено описать наши данные таким образом, что они будут компилироваться в обоих случаях так, как мы того хотим.

Попробуйте записать

```
TCHAR* pszMyString = _TEXT("This is my string");
```

и откомпилировать программу с определением _UNICODE и без такового. Ну и как? Получилось?

К этому моменту трудности совместной работы Unicode и Run-Time Library уже преодолены. На очереди -

РАБОТА С UNICODE В WIN32 API

Предлагаю читателю обратить внимание на то, каким образом определяются типы данных и указатели в заголовочных файлах Win 32 (обратите внимание - символ UNICODE без знака подчеркивания, со знаком подчеркивания он используется в RTL):

```
#ifdef UNICODE
 typedef wchar_t TCHAR;
#else
 typedef unsigned char TCHAR;
#endif
typedef TCHAR * LPTSTR, *LPTCH;
typedef unsigned char CHAR;
typedef CHAR *LPSTR, *LPCH;
typedef unsigned wchar_t WCHAR;
typedef WCHAR *LPWSTR, *LPWCH;
```

Что же касается работы с функциями, то в Win32, по сравнению с RTL, при работе с функциями программист может чувствовать себя более комфортно. Здесь все сделано намного проще и, по-моему, более изящно. При работе с обеими кодировками используются одинаковые имена функций. Достигается это следующим образом:

- каждая функция, как и в RTL, имеет ANSI- и Unicode-версии. При этом имена функций формируются так:
  - к обычному имени (думаю, не нужно объяснять, что я подразумеваю под обычным именем?) в случае ANSI-версии добавляется символ A (латинская буква, не путать с русской), например для функции DispatchMessage() имя ANSI-версии будет DispatchMessageA();
  - для Unicode-версии к обычному имени функции добавляется символ W. Например, для функции DispatchMessage() имя Unicode-версии будет DispatchMessageW();

- создается макрос, имя которого совпадает с обычным именем функции. При определении символа UNICODE макрос разворачивается в имя Unicode-версии функции. Если символ UNICODE не определен, то макрос разворачивается в имя ANSI-версии функции. Например, для уже известной нам функции DispatchMessageW() эти определения выглядят следующим образом:

```
WINUSERAPI LONG WINAPI DispatchMessageA(CONST MSG *lpMsg);
WINUSERAPI LONG WINAPI DispatchMessageW(CONST MSG *lpMsg);
#ifdef UNICODE
#define DispatchMessage  DispatchMessageW
#else
#define DispatchMessage  DispatchMessageA
#endif // !UNICODE
```

Все гениальное просто! Единственное, я прошу читателя обратить внимание на то, что при компиляции программ обычно приходится определять оба символа препроцессора - UNICODE и _UNICODE.

НЕСКОЛЬКО ДОБРЫХ СОВЕТОВ

К этим советам читатель может не прислушиваться, если не захочет. Просто мне бы хотелось поделиться тем небольшим опытом, который появился у меня при изучении Unicode. Самое главное, что должен осознать программист, это то, что, включив Unicode в свои системы Windows NT и Windows'95, Microsoft на весь мир заявила о том, что у Unicode существует мощная поддержка в лице этой фирмы. А к словам Microsoft (за которыми в большинстве случаев следуют дела), прислушаться стоит. Так что хотят этого программисты или не хотят, рано или поздно им придется переходить на работу с Unicode. Поэтому неплохо было бы позаботиться о возможной работе приложения с обеими кодировками сейчас, чтобы избежать «большой крови» потом.

Что для этого нужно? Немногое:

- для символов и строк использовать типы TCHAR и LPTSTR;
- при определении литералов использовать макрос _TEXT;
- не забывать о правилах строковой арифметики.

На этом заканчивается рассмотрение вопросов, связанных с Unicode. Мне думается, я рассказал достаточно, чтобы у читателя появился «направляющий косинус» в этом вопросе. На страницах этой книги мы еще не раз встретим имя этой кодировки.

ОСНОВЫ РИСОВАНИЯ И КОПИРОВАНИЯ ИЗОБРАЖЕНИЙ

НЕМНОГО ЛИРИКИ

Когда я обдумывал план этой книги, мне казалось, что за главой, в которой описана структура программы, должна идти глава о взаимодействии программы с пользователем. Действительно, чего тянуть? Ведь подавляющее большинство программ для Windows написано с расчетом, что пользователь будет работать с программой в интерактивном режиме. Я начал было писать главу, посвященную меню и диалоговым окнам, но тут же вынужден был остановиться. Мне пришлось чуть ли не на каждой строчке извиняться и говорить, что все это мы узнаем чуть позже. Bitmap'ы в меню и кнопках - это была последняя капля, переполнившая чашу. Поэтому было принято следующее решение: на время несколько отклониться от нашего курса и изучить основы работы с изображениями (убрать тот самый камень с нашей дороги), после чего идти дальше. Думаю, что время, затраченное на изучение работы с изображениями, окупится сторицей. Я понимаю, что читателю не терпится написать какую-нибудь грандиозную программу. Постараюсь, чтобы наше отклонение от курса было недолгим, и изложение буду вести почти в конспективном стиле (не в ущерб содержанию, надеюсь).

К этому моменту читатель уже представляет, как должна выглядеть программа для Windows, знает кое-что о сообщениях и о том, что программа сама отвечает за перерисовку окна в случае необходимости.

Давайте, уважаемый читатель, обсудим вопрос: что должно происходить в тех случаях, когда программе необходимо отобразить в рабочей области окна некоторое изображение? Для начала рассмотрим случаи, когда программе не нужно рисовать изображение, а необходимо скопировать одно изображение на другое. Поговорим о том, что такое

КОНТЕКСТ УСТРОЙСТВА

Наверное, читатель уже знает о том, что с точки зрения программиста Windows является системой, не зависящей от устройств (device independent). Эту независимость со стороны Windows обеспечивает библиотека GDI32.dll, а со стороны устройства - драйвер этого устройства. С точки зрения программы связующим звеном между программой и устройством является контекст устройства (Device Context - DC). Если

программе нужно осуществить обмен с внешним устройством, программа должна оповестить GDI о необходимости подготовить устройство для операции ввода-вывода. После того, как устройство подготовлено, программа получает хэндл контекста устройства, т. е. хэндл структуры, содержащей набор характеристик этого устройства. В этот набор входят:

bitmap (битовая карта, изображение), отображаемый в окне,
перо для прорисовки линий,
кисть,
палитра,
шрифт

и т. д. Программа никогда напрямую не обращается к контексту устройства (кстати, эта структура не документирована Microsoft), она обращается к нему опосредствованно, через определенные функции. После того, как все действия произведены, и необходимость в использовании устройства отпала, программа должна освободить контекст устройства, чтобы не занимать память. Есть еще одна причина, из-за которой необходимо освобождать контекст устройства. В системе может существовать одновременно только ограниченное число контекстов устройств. Если контекст устройства не будет освобождаться после операций вывода, то через несколько перерисовок окна система может зависнуть. Так что не забывайте освобождать контексты устройств!

Когда программа требует контекст устройства, она получает его уже заполненным значениями по умолчанию. Объект в составе контекста называется текущим объектом. Само слово - текущий - говорит о том, что контекст устройства можно изменить. Программа может создать новый объект, скажем, bitmap или шрифт, и сделать его текущим. Замещенный объект автоматически из памяти не удаляется, его необходимо позже удалить отдельно. Само собой разумеется, что программа может получить характеристики текущего устройства. А вот изменить эти характеристики, увы, можно только через замену объекта (впрочем, это и так понятно).

ТИПЫ КОНТЕКСТА УСТРОЙСТВА

В Windows поддерживаются следующие типы контекстов устройств:
контекст дисплея (обеспечивает работу с дисплеем);
контекст принтера (обеспечивает работу с принтером);
контекст в памяти (моделирует в памяти устройство вывода);
информационный контекст (служит для получения данных от устройства).

Контекст дисплея

Windows поддерживает три типа контекста дисплея - контекст класса, приватный контекст и общий контекст. Первые два типа используются в приложениях, которые выводят на экран большое количество информации. Ярким примером такого рода приложений являются настольные издательские системы, графические пакеты и т.д.

Приложения, которые не очень интенсивно работают с экраном, используют общий контекст. Контекст класса является устаревшим и поддерживается только для обеспечения совместимости с предыдущими версиями Windows. Microsoft не рекомендует использовать его при разработке новых приложений и рекомендует использовать только приватный контекст. Исходя из этого, я решил пожалеть свое и читателя время и не рассматривать контекст класса.

Контексты устройств хранятся в кэше, управляемом системой. Хэндл общего контекста программа получает с помощью функций GetDC(), GetDCEx() или BeginPaint(). После того, как программа отработает с дисплеем, она должна освободить контекст, вызвав функцию ReleaseDC() или EndPaint() (в случае, если контекст получался с помощью BeginPaint()). После того, как контекст дисплея освобожден, все изменения, внесенные в него программой, теряются и при повторном получении контекста все действия по изменению контекста необходимо повторять заново.

Приватный контекст отличается от общего тем, что сохраняет изменения даже после того, как прикладная программа освободила его. Приватный контекст не хранится в кэше, поэтому прикладная программа может не освобождать его. Естественно, что в этом случае за счет использования большего объема памяти достигается более высокая скорость работы с дисплеем.

Для работы с приватным контекстом необходимо при регистрации класса окна указать стиль CS_OWNDC. После этого программа может получать хэндл контекста устройства точно так же, как и в случае общего контекста. Система удаляет приватный контекст в том случае, когда удаляется окно.

При работе с контекстами необходимо запомнить, что хэндлы контекста устройства с помощью функции BeginPaint() необходимо получать только в случае обработки сообщения WM_PAINT. Во всех остальных случаях необходимо использовать функции GetDC() или GetDCEx().

Контекст принтера

При необходимости вывода на принтер программа должна создать контекст устройства с помощью функции CreateDC(). Аргументы этой функции определяют имя драйвера устройства, тип устройства и инициализационные данные для устройства. Используя эти данные, система может подготовить принтер и распечатать требуемые данные. После распечатки прикладная программа должна удалить контекст принтера с помощью функции DeleteDC() (а не ReleaseDC()).

Информационный контекст

Информационный контекст фактически не является контекстом устройства и служит только для получения информации о действительном контексте устройства. К примеру, для того, чтобы получить характеристики принтера, программа создает информационный контекст, используя для этого функцию CreateIC(), а затем из него выбирает требующиеся характеристики. Естественный вопрос: а для чего нужно использовать информационный контекст? Почему нельзя те же самые данные получить из действительно контекста? Дело в том, что этот тип контекста создается и работает намного быстрее, а также занимает меньше памяти по сравнению с действительным контекстом. После того, как надобность в информационном контексте миновала, программа должна удалить его с помощью функции DeleteDC().

Чаще всего для вывода информации на устройство используется

Контекст в памяти

Этот контекст используется для хранения изображений, которые затем будут скопированы на устройство вывода. Сам по себе контекст в памяти не создается. Он обязательно создается как совместимый с тем устройством или окном, на которое предполагается копировать информацию (вот он - совместимый контекст - переходник между программой и драйвером устройства!). Алгоритм работы с контекстом в памяти состоит из нескольких шагов:

1. Получения хэндла контекста устройства (назовем его hDC - handle of Device Context) для окна, в которое будет осуществляться вывод изображения.

2. Получения хэндла bitmap'а, который будет отображаться в окне.

3. Получения совместимого с hDC контекста в памяти (для хранения изображения) с помощью функции CreateCompatibleDC() (обратите внимание на название функции - создать СОВМЕСТИМЫЙ контекст).

4. Выбора изображения (hBitmap) как текущего для контекста в памяти (hCompatibleDC).

5. Копирования изображения контекста в памяти (hCompatibleDC) на контекст устройства (hDC).

6. Удаления совместимого контекста (hCompatibleDC);

7. Принятия мер для того, чтобы замещенный bitmap из контекста в памяти не остался в памяти.

8. Освобождения контекста устройства (hDC).

При необходимости шаги 6 и 7 можно поменять местами. Когда и как удалять замещенный bitmap, зависит от программиста и поставленной перед ним задачи.

Именно этот способ и используется в большинстве программ для копирования изображения.

Но, как известно, лучше один раз увидеть, чем сто раз услышать (по-английски это звучит еще более категорично - seeing is believing - увидеть, значит поверить). Поэтому давайте напишем небольшую программу, в которой продемонстрируем возможности Windows по манипулированию изображениями.

SEEING IS BELIEVING

Перед тем, как начинать писать программу, мы должны прояснить для себя еще одну тему. В каких единицах измеряются размеры окна и, соответственно, все смещения в окне? Для того чтобы ответить на этот вопрос, мы должны рассмотреть

Режимы отображения

Подавляющее большинство функций, работающих с оконными координатами, определяют координаты относительно начала рабочей области окна, т. е. от левого верхнего угла.

Таким образом, даже при перемещении окна координаты объектов внутри окна остаются неизменными. При этом единицы, в которых измеряются координаты, зависят от режима отображения (mapping mode), установленного для данного окна. Единицы измерения, зависящие от режима отображения, называют логическими единицами, а координаты в этом случае называют логическими координатами.

При выводе информации на конкретное устройство единицы логических координат преобразуются в физические единицы, которыми являются пиксели.

Таблица 6. Идентификаторы, применяемые для обозначения режимов отображения

Идентификатор	Значение	Эффект
MM_TEXT	1	Логическая единица равна пикселю, начало координат - левый верхний угол окна, положительное значение x - вправо, положительное значение y - вниз (обычный отсчет)
MM_LOMETRIC	2	Логическая единица равна 0,1 мм, отсчет координат - обычный
MM_HIMETRIC	3	Логическая единица равна 0,01 мм, отсчет координат - обычный
MM_LOENGLISH	4	Логическая единица равна 0,1 дюйма, отсчет координат - обычный
MM_HIENGLISH	5	Логическая единица равна 0,001 дюйма, отсчет координат - обычный
MM_TWIPS	6	Логическая единица равна 1/12 точки на принтере (~ 1/1440 дюйма - «твип»), отсчет координат - обычный
MM_ISOTROPIC	7	Логические единицы и направление осей определяются программистом с помощью функций SetWindowExtEx() и SetViewportExtEx(), единицы по осям имеют одинаковый размер
MM_ANISOTROPIC	8	Логические единицы и направления осей определяются так же, как и для MM_ISOTROPIC, но размеры единиц по осям различны
MM_MIN		MM_TEXT
MM_MAX		MM_ANISOTROPIC
MM_MAX_FIXEDSCALE		MM_TWIPS

Для установки текущего режима отображения используется функция SetMappingMode(), которая в файле wingdi.h описана следующим образом:

```
WINGDIAPI int   WINAPI SetMapMode(HDC, int);
```

Первый аргумент этой функции - хэндл контекста устройства, для которого устанавливается данный режим. Второй аргумент определяет задаваемый режим отображения. В том же файле wingdi.h можно найти и идентификаторы, использующиеся для обозначения режимов отображения (табл. 6). Надеюсь, что после того, как была просмотрена таблица, вопросов у читателя не возникло. Теперь ясно, что иногда для решения конкретных задач (например, построения графиков) можно использовать различные режимы отображения. При создании окна по умолчанию устанавливается режим MM_TEXT, т. е. все координаты исчисляются в пикселах.

Пишем программу

Наша программа будет отображать bitmap в окне и при необходимости производить его масштабирование:

```
#include <windows.h>

LRESULT CALLBACK DCDemoWndProc ( HWND, UINT, UINT, LONG );

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow )

 {
  HWND hWnd ;
  WNDCLASS WndClass ;
  MSG Msg;
  char szClassName[] = "DCDemo";

/* Registering our window class */
/* Fill WNDCLASS structure */

  WndClass.style = CS_HREDRAW | CS_VREDRAW;
  WndClass.lpfnWndProc = DCDemoWndProc;
  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 = "MyMenu";
  WndClass.lpszClassName = szClassName;

  if ( !RegisterClass(&WndClass) )
    {
     MessageBox(NULL,"Cannot register class","Error",MB_OK);
     return 0;
    }

  hWnd = CreateWindow(szClassName, "Program No 1",
                      WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                      CW_USEDEFAULT, CW_USEDEFAULT,
                      CW_USEDEFAULT, NULL, NULL,
                      hInstance,NULL);
  if(!hWnd)
    {
     MessageBox(NULL,"Cannot create window","Error",MB_OK);
     return 0;
    }
```

```
  /* Show our window */
  ShowWindow(hWnd,nCmdShow);
  UpdateWindow(hWnd);

/* Beginning of messages cycle */
  while(GetMessage(&Msg, NULL, 0, 0))
   {
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
   }
  return Msg.wParam;

 }

LRESULT CALLBACK DCDemoWndProc (HWND hWnd, UINT Message,
                                UINT wParam, LONG lParam )
 {

  HDC hDC, hCompatibleDC;
  PAINTSTRUCT PaintStruct;
  HANDLE hBitmap, hOldBitmap;
  RECT Rect;
  BITMAP Bitmap;
  switch(Message)
   {
    case WM_PAINT:
// Получаем контекст устройства.
      hDC = BeginPaint(hWnd, &PaintStruct);
// Загружаем bitmap, который будет отображаться в окне , из файла.
      hBitmap = LoadImage(NULL, "MSDOGS.BMP",
                          IMAGE_BITMAP,
                          0, 0,
                          LR_LOADFROMFILE);
// Получаем размерность загруженного bitmap'а.
      GetObject(hBitmap, sizeof(BITMAP), &Bitmap);
// Создаем совместимый с контекстом окна контекст в памяти.
      hCompatibleDC = CreateCompatibleDC(hDC);
// Делаем загруженный из файла bitmap текущим в совместимом контексте.
      hOldBitmap = SelectObject(hCompatibleDC, hBitmap);
// Определяем размер рабочей области окна.
      GetClientRect(hWnd,&Rect);
// Копируем bitmap с совместимого на основной контекст с масштабированием.
      StretchBlt(hDC, 0, 0, Rect.right, Rect.bottom,
                 hCompatibleDC, 0, 0, Bitmap.bmWidth,
                 Bitmap.bmHeight, SRCCOPY);
// Вновь делаем старый bitmap текущим.
      SelectObject(hCompatibleDC, hOldBitmap);
// Удаляем загруженный с диска bitmap.
      DeleteObject(hBitmap);
// Удаляем совместимый контекст.
```

```
        DeleteDC(hCompatibleDC);
// Освобождаем основной контекст, завершая перерисовку рабочей области окна.
        EndPaint(hWnd,&PaintStruct);
        return 0;
      case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    }
  return DefWindowProc(hWnd,Message,wParam, lParam);
}
```

Листинг № 2. Программа, осуществляющая отображение bitmap'а с масштабированием.

Результатом работы программы является окно, показанное на рис. 2.

Если читатель набрал программу буква в букву, его при запуске ожидает одна неприятность - отобразится точно такое же окно с белым фоном, как и при запуске «Hello, world!» (только без надписи в центре экрана). Дело в том, что в моей программе отображается тот bitmap, который нравится мне и, что более важно, находится на моем компьютере в доступной директории. Предлагаю читателю в тексте программы найти оператор, который начинается с «hBitmap = LoadImage» и заменить в нем имя моего bitmap'а, «msdogs.bmp», на имя того bitmap'а, который будет отображать программа на читательском компьютере. Не забудьте при этом проверить, чтобы новый bitmap был доступен, т. е. находился бы в директории, доступной через переменную окружения PATH, или в текущей директории. Сделали? Теперь попробуйте еще раз запустить программу. Если все выполнено правильно, то bitmap точно впишется в окно, причем можно будет заметить, что он несколько растянут или сжат в обоих направлениях. Попробуйте поизменять размеры окна. Bitmap постоянно будет точно вписываться в окно, при этом растягиваясь или сжимаясь. А теперь давайте разберем эту программу и сопоставим операторы программы с шагами алгоритма работы с контекстом в памяти, о котором я говорил ранее.

Функция WinMain() стандартна - ничего интересного в данном случае она не содержит. В оконной функции, которая называется DCDemoWndProc, интерес для нас представляет обработка сообщения WM_PAINT, которое мы и рассмотрим. Первый шаг алгоритма - получить хэндл контекста устройства - мы выполняем посредством вызова функции BeginPaint(hWnd, &PaintStruct). Аргумент hWnd очевиден - мы получаем контекст данного окна. Что же касается структуры PaintStruct...

Рис. 2. Вид окна, отображаемого программой

Понятно, что окно далеко не всегда должно перерисовываться полностью. К примеру, только часть окна перекрывается другим окном. Естественно, что и перерисоваться должна только часть окна. В этом случае полная перерисовка всего окна будет только лишней тратой времени и ресурсов компьютера. Windows «знает» о том, какая часть окна перекрыта. При необходимости перерисовки (при вызове BeginPaint()) система заполняет структуру типа PAINTSTRUCT, адрес которой передается системе как второй аргумент BeginPaint(). Среди всех полей структуры типа PAINTSTRUCT основным (на мой взгляд) является поле, содержащее координаты той области (clipping region), которая должна быть перерисована. В нашем примере информация, хранящаяся в этой структуре, не используется, но я прошу читателя обратить внимание на эту структуру и в дальнейшем использовать ее. Получив от функции BeginPaint() хэндл контекста устройства (hDC), будем считать первый шаг выполненным.

Второй шаг - получение хэндла bitmap'а, который будет отображаться в окне - мы делаем, вызывая функцию LoadImage(). Я не случайно вос-

пользовался именно этой функцией. Во-первых, возможности этой функции достаточно широки. Она позволяет загружать графические образы как из ресурсов, записанных в исполняемом файле, так и из файлов, содержащих только изображения. Графическим образом может быть bitmap, курсор и иконка. Кроме этого, функция позволяет управлять параметрами отображения и загрузки образа. Во-вторых, подавляющее большинство функций работают с ресурсами, сохраненными в исполняемом файле, и у программистов, начинающих осваивать Win32, попытки загрузить что-либо из файла сопровождаются некоторыми трудностями. (Помнится, что я сам в начале изучения программирования для Windows (не Win32) несколько часов потратил на поиски в help'ах функции, позволяющей загрузить bitmap из файла). Но обо всем по порядку.

В файле winuser.h эта функция описана следующим образом:

```
WINUSERAPI HANDLE WINAPI LoadImageA(HINSTANCE, LPCSTR, UINT,
                                    int, int, UINT);
WINUSERAPI HANDLE WINAPI LoadImageW(HINSTANCE, LPCWSTR,
                                    UINT, int, int, UINT);
#ifdef UNICODE
#define LoadImage  LoadImageW
#else
#define LoadImage  LoadImageA
#endif // !UNICODE
```

Первый, второй и последний аргументы этой функции работают в связке. Первый аргумент(hInst) - хэндл программы. Как читатель должен помнить, если вместо хэндла программы указан нуль, то объект является предопределенным, т. е. хранится в «глубинах» системы. В противном случае, объект загружается откуда-то снаружи. Второй аргумент - lpszName - определяет загружаемый объект. Последний аргумент - fuLoad - содержит флаги, определяющие режим загрузки объекта. Среди этих флагов есть флаг LR_LOADFROMFILE. Его название определяет его назначение - если этот флаг установлен, загрузка происходит из внешнего файла. От значения первого и последнего аргументов зависит, как будет интерпретирован второй аргумент. Взаимодействие этих трех аргументов объясняется в табл. 7.

Третий аргумент - тип образа, он может принимать значения IMAGE_BITMAP, IMAGE_CURSOR, IMAGE_ICON и IMAGE_ENHMETAFILE. Здесь комментарии излишни. Четвертый и пятый аргументы указывают ширину и высоту иконки или курсора и в нашем примере не используются.

Таблица 7. Взаимодействие аргументов функции LoadImage()

fuLoad(флаг LR_LOADFROMFILE)	hInst	lpszName
Не установлен	NULL	Идентификатор предопределенного объекта
Не установлен	не NULL	Имя ресурса
Установлен	NULL	Имя файла, в котором содержится bitmap
Установлен	не NULL	Имя файла, в котором содержится bitmap

Таблица 8. Флаги, определяющие параметры загрузки образа в память

Флаг	Значение	Эффект
LR_DEFAULTCOLOR	0x0000	Указывает, что загружаемый образ - не монохромный
LR_MONOCHROME	0x0001	Указывает, что загружаемый образ - черно-белый
LR_COLOR	0x0002	
LR_COPYRETURNORG	0x0004	
LR_COPYDELETEORG	0x0008	
LR_LOADFROMFILE	0x0010	Образ необходимо загружать из файла, а не из ресурсов
LR_LOADTRANSPARENT	0x0020	Все пиксели, цвет которых совпадает с цветом пикселя, расположенного в левом верхнем углу bitmap'а, отображаются как прозрачные
LR_DEFAULTSIZE	0x0040	Использовать ширину и высоту образа, определенные в системных метриках для иконки и курсора, если cxDesired или cyDesired равны 0. Если этот флаг не установлен, а cxDesired и/или cyDesired равны 0, используются размеры образа, указанные в ресурсе
LR_LOADMAP3DCOLORS	0x1000	Заменить следующие оттенки серого цвета: RGB(128, 128, 128) (DkGray) - на COLOR_3DSHADOW, RGB(192, 192, 192) (Gray) - на COLOR_3DFACE, RGB(223, 223, 223) (LtGray) - на COLOR_3DLIGHT
LR_CREATEDIBSECTION	0x2000	При загрузке bitmap'а возвращает оригинальные значения цветов, не преобразуя bitmap в совместимый с данным контекстом
LR_COPYFROMRESOURCE		
LR_SHARED	0x8000	Разделять хэндл загруженного изображения, в случае, если образ загружается несколько раз. Нежелательно применять к образам нестандартных размеров

Последний аргумент - флаги, определяющие параметры загрузки образа в память. Их достаточно много, только в файле winuser.h я насчитал 12 возможных идентификаторов. Все они начинаются с букв LR. Ничего сложного в них нет, и читатель сам сможет изучить их. Краткое описание этих флагов приведено в табл. 8.

Функция LoadImage() возвращает нам хэндл загруженного bitmap'а (hBitmap) (или NULL, если где-то что-то не так), после чегомы можем считать второй шаг нашего алгоритма пройденным.

Третий шаг - получение совместимого контекста в памяти - выполняется посредством вызова функции CreateCompatibleDC(). Единственный аргумент этой функции - хэндл контекста (hDC), для которого создается совместимый контекст.

Четвертый шаг мы реализуем вызовом функции SelectObject(). Первым аргументом указываем хэндл контекста, в котором замещается текущий элемент (в данном случае это хэндл только что созданного контекста в памяти hCompatibleDC), а вторым - хэндл элемента, которым замещается текущий элемент (хэндл загруженного bitmap'а hBitmap). Немаловажно, что эта функция возвращает хэндл ЗАМЕЩЕННОГО элемента (hOldBitmap), т. е. впоследствии с этим элементом могут также производиться манипуляции.

А вот на пятом шаге происходит то, ради чего мы заварили всю эту кашу с загрузкой bitmap'ов, совместимыми контекстами и прочим. Для копирования bitmap'а (с масштабированием) с одного контекста на другой, мы используем функцию StretchBlt(), одну из «могучих blt», по меткому выражению Чарльза Петцольда. К их числу, помимо StretchBlt(), относятся BitBlt() и PatBlt().

Наверное, StretchBlt() является самой «могучей» из них. И наверное, ее мощь и обусловила наличие у этой функции «всего-навсего» одиннадцати аргументов. В файле wingdi.h эта функция описана следующим образом:

```
WINGDIAPI BOOL  WINAPI StretchBlt(HDC, int, int, int, int, HDC, int, int, int, int,
                                  DWORD);
```

Первые пять аргументов описывают тот прямоугольник на экране, в который будет вписан bitmap (на рис. 3 он обозначен светло-серым цветом). Ту часть bitmap'а, которая будет вписана в прямоугольник на экране (на рисунке - пересекающаяся часть светло-серого и темно-серого прямоугольников), описывают следующие пять аргументов. И последний, одиннадцатый аргумент, так называемый код растровой операции, описывает, каким образом пиксели одного bitmap'а будут взаимодействовать

с пикселами другого bitmap'а. Для того чтобы лучше понять аргументы функции, обратимся к рис. 3. Представим, что в окне, в регионе, обозначенном на рисунке светло-серым цветом, нужно отобразить bitmap (обозначен на рисунке темно-серым цветом) или часть bitmap'а, при необходимости сжав или растянув ее.

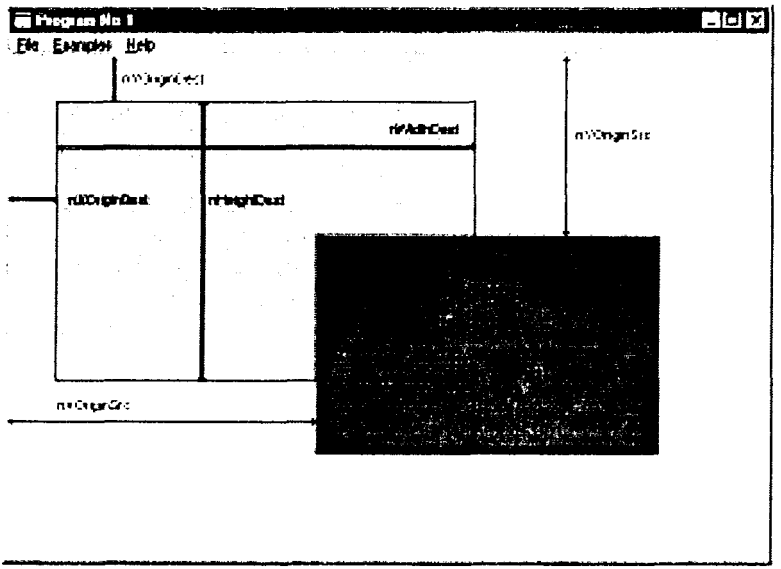


Рис. 3. Взаимодействие аргументов функции StretchBlt

Итак, первый и шестой аргументы функции - хэндлы окна (hDC) и совместимого контекста (hCompatibleDC) соответственно. Второй (nXOriginDest) и третий (nYOriginDest) аргументы содержат смещение верхнего левого угла прямоугольника, в который будет вписываться bitmap, относительно левой и верхней сторон рабочей области окна (В каких единицах выражено смещение? Напомню, мы создали окно и не меняли режим отображения, т. е. текущий режим является установ*ленным по умолчанию). Четвертый (nWidthDest) и пятый (nHeightDest) аргументы - ширина и высота этого прямоугольника. Седьмой (nXOriginSrc) и восьмой (nYOriginSrc) аргументы аналогичны второму и третьему аргументам. Девятый (nWidthSrc) и десятый (nHeightSrc) аргументы содержат ширину и высоту отображаемой части bitmap'а. Не нужно обладать развитым пространственным воображением, чтобы

понять, что, изменяя положения прямоугольников друг относительно друга, а также относительно окна, меняя их размеры, можно отобразить на экране любую часть или целый bitmap.

В примере мы хотим фактически совместить наш bitmap с рабочей областью окна, поэтому второй и третий аргументы вызываемой функции равны нулю. Четвертый и пятый аргументы равны ширине и высоте рабочей области (мы получили ширину и высоту рабочей области с помощью функции GetClientRect()). Седьмой и восьмой аргументы равны нулю (подумайте, почему?), а девятый и десятый содержат ширину и высоту bitmap'а, которые мы получили, обратившись к GetObject(). Вот, кажется, и все. Нет, не все.

Как работает одиннадцатый аргумент, определяющий взаимодействие двух bitmap'ов? Давайте закончим обсуждение нашего алгоритма, а затем вернемся к этому вопросу.

Мы прошли уже пять шагов алгоритма. Остались еще три шага - удаление совместимого контекста, объекта и контекста устройства - пусть читатель сам определит, какие операторы программы их реализуют.

Неужели мы добрались до конца нашего алгоритма? Мне он показался бесконечным! Давайте прежде чем рассматривать одиннадцатый аргумент упомянем об оставшихся «могучих Blt».

Функция BitBlt() тоже копирует bitmap с одного контекста на другой, но без масштабирования. Следовательно, ей не нужны девятый и десятый аргументы - ширина и высота отображаемой области - отображается все то, что поместится в отведенную для этого область (светло-серый прямоугольник на рис. 3).

Последняя из «могучих» - функция PatBlt() - просто закрашивает прямоугольник на экране, используя для этого выбранную в контексте устройства кисть. Раз нет отображаемого объекта, то зачем нам его контекст и координаты? Отбрасываем аргументы с шестого по десятый включительно и получаем список аргументов PatBlt().

Наконец мы подошли к тому, чтобы уяснить, что же представляет собой одиннадцатый аргумент функции StretchBlt(). Сейчас мы поговорим о том, что же такое

КОДЫ РАСТРОВЫХ ОПЕРАЦИЙ

Выше уже было сказано, что одиннадцатый аргумент функции StretchBlt() - это код растровой операции. Другими словами, это код, который определяет, как при операции будут взаимодействовать биты, определяющие заливку и изображение совместимого контекста с изобра-

жением на действительном контексте. Комбинируются биты на основе логических операций над ними. По укоренившейся в книгах по программированию для Windows традиции, эти операции записываются в обратной польской нотации (не путать с венгерской, автор польской нотации не имеет к Microsoft ни малейшего отношения).

По той же традиции, биты, определяющие bitmap совместимого контекста, обозначают буквой S (source - источник, исходный), биты заливки - буквой P (pattern - образец), а биты, на которых будет прорисовываться изображение - буквой D (destination - назначение, место назначения). Операции обозначаются следующим образом: a - побитовое И (AND), n - побитовое НЕТ (NO), o - побитовое ИЛИ (OR), x - побитовое исключающее ИЛИ (XOR).

Несколько слов о польской нотации. В ней операции записываются слева направо. Знак операции следует за операндами. Появление знака операции означает, что нужно произвести следующие действия:

- взять два последних операнда;
- произвести с ними требующуюся операцию;
- записать результат на место последних двух операндов.

Фактически польская нотация описывает действия таким образом, словно операнды и операции находятся в стеке, для чего, собственно, эта польская нотация и была изобретена.

Обозначив знак операции как Op, в польской нотации действия с битами можно записать таким образом:

PSOp

Это говорит о необходимости взять пиксель патерны и прорисовываемого bitmap'а и произвести над ними операцию. Если в операции участвуют три операнда, то получим:

DPSOp1Op2

Что мы должны сделать в этом случае? Правильно, сначала произвести действие, определяемое Op1, с битами патерны и прорисовываемым bitmap'ом, после этого произвести Op2 с полученным результатом и битами действительного контекста. Ничего сложного здесь нет.

Каждый код растровой операции представляется 32-битным целым. Старшее слово кода представляет собой индекс битовой операции, младшее - код операции. Как определяется индекс операции?

Давайте представим, что нам необходимо определить индекс растровой операции, определяемой в польской нотации записью DPSxx. Попутно можно определить и индекс операции PSx. Запишем друг под другом ОПРЕДЕЛЕННЫЕ значения P, S и D, а под ними - результаты побитовых операций PSx и DPSxx:

P	1	1	1	1	0	0	0	0
S	1	1	0	0	1	1	0	0
D	1	0	1	0	1	0	1	0
PSx	0	0	1	1	1	1	0	0
DPSxx	1	0	0	1	0	1	1	0

Таблица 9. Краткое описание кодов растровых операций

Наименование	Индекс операции	Польская запись	Эффект
BLACKNESS	0x00	0	Заполнение действительного контекста черным цветом
NOTSRCERASE	0x11	DSon	
NOTSRCCOPY	0x33	Sn	Прорисовываемый bitmap отображается в негативном виде
DSTINVERT	0x55	Dn	Изображение действительного контекста проявляется негативным
PATINVERT	0x5A	DPx	
SRCINVERT	0x66	DSx	
SRCAND	0x88	DSa	
MERGEPAINT	0xBB	DSno	
MERGECOPY	0xC0	PSa	
SRCCOPY	0xCC	S	Копирование прорисовываемого bitmap'а на действительный контекст
SRCPAINT	0xEE	DSo	
PATCOPY	0xF0	P	Копирование патерны на действительный контекст
PATPAINT	0xFB	DPSnoo	
WHITENESS	0xFF	1	Заполнение действительного контекста белым цветом

Итак, индекс операции PSx - 0x3c, а индекс операции DPSxx - 0x96.

Уважаемый читатель, я прошу обратить внимание на то, что друг под другом записываются не произвольные, а строго определенные значения. Эти значения позволяют перебрать все возможные комбинации патерны, исходного и целевого bitmap'ов. Теперь, когда все стало ясно, вы можете попробовать попрактиковаться в определении индексов любых операций. Несмотря на то, что существуют 256 индексов растровых операций, на практике используются только некоторые из них. В файле wingdi.h для наиболее часто используемых растровых операций определены идентификаторы, которые приведены в табл. 9.

На основании данных табл. 9 я затрудняюсь объяснить, как изменяется изображение при использовании разных растровых операций. Рекомендую читателю запустить приведенную выше программу несколько раз, и каждый раз в функции StretchBlt() указывать новую растровую операцию. Seeing is believing!

Теперь и одиннадцатый аргумент PatBlt() стал ясным и понятным - я просто копирую bitmap в окно. Только и всего. Кстати, понимание логики работы с растровыми операциями может позволить избежать трудоемких преобразований bitmap'ов перед копированием.

ПОЛОСЫ ПРОКРУТКИ

Мне кажется, что у читателя может возникнуть вопрос: что делать в тех случаях, когда нам не нужно масштабировать bitmap, но нужно иметь возможность просматривать все части изображения? Ответ заключен в названии данного раздела - использование полос прокрутки.

Давайте, уважаемый читатель, чуть-чуть изменим предыдущую программу для того, чтобы продемонстрировать использование полос прокрутки, и посмотрим, как она будет работать. Текст программы с внесенными изменениями:

```
#include <windows.h>

long WINAPI DCDemoWndProc ( HWND, UINT, UINT, LONG );

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow )
 {
  HWND hWnd ;
  WNDCLASS WndClass ;
  MSG Msg;
  char szClassName[] = "DCDemo";

/* Registering our window class */
/* Fill WNDCLASS structure */

  WndClass.style = CS_HREDRAW | CS_VREDRAW;
  WndClass.lpfnWndProc = DCDemoWndProc;
  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 = "MyMenu";
  WndClass.lpszClassName = szClassName;
```

```
  if ( !RegisterClass(&WndClass) )
    {
     MessageBox(NULL,"Cannot register class","Error",MB_OK);
     return 0;
    }

  hWnd = CreateWindow(szClassName, "Program No 1",
                      WS_OVERLAPPEDWINDOW |
                      WS_VSCROLL | WS_HSCROLL,
                      CW_USEDEFAULT, CW_USEDEFAULT,
                      CW_USEDEFAULT, CW_USEDEFAULT,
                      NULL, NULL,
                      hInstance,NULL);
  if(!hWnd)
    {
     MessageBox(NULL,"Cannot create window","Error",MB_OK);
     return 0;
    }

  /* Show our window */
ShowWindow(hWnd,nCmdShow);
  UpdateWindow(hWnd);

/* Beginning of messages cycle */
  while(GetMessage(&Msg, NULL, 0, 0))
   {
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
   }
  return Msg.wParam;
 }

LRESULT CALLBACK DCDemoWndProc (HWND hWnd, UINT Message,
                                UINT wParam, LONG lParam )
 {

  HDC hDC, hCompatibleDC;
  PAINTSTRUCT PaintStruct;
  static HANDLE hBitmap;
  HANDLE hOldBitmap;
  RECT Rect;
  BITMAP Bitmap;
  static int nHorizDifference = 0, nVertDifference = 0;
  static int nHorizPosition = 0, nVertPosition = 0;

  switch(Message)
   {
    case WM_CREATE:
     hBitmap = LoadImage(NULL, "MSDOGS.BMP",
                         IMAGE_BITMAP,
```

```
                        0, 0,
                        LR_LOADFROMFILE);
  return 0;
 case WM_PAINT:
  hDC = BeginPaint(hWnd, &PaintStruct);
  GetObject(hBitmap, sizeof(BITMAP), &Bitmap);
  hCompatibleDC = CreateCompatibleDC(hDC);
  hOldBitmap = SelectObject(hCompatibleDC, hBitmap);
  GetClientRect(hWnd,&Rect);
  BitBlt(hDC, 0, 0, Rect.right, Rect.bottom,
        hCompatibleDC, nHorizPosition, nVertPosition, SRCCOPY);
  if( (nHorizDifference = (Bitmap.bmWidth - Rect.right)) > 0)
    SetScrollRange(hWnd, SB_HORZ, 0, nHorizDifference, TRUE);
   else
    SetScrollRange(hWnd, SB_HORZ, 0, 0, TRUE);
  if( (nVertDifference = (Bitmap.bmHeight - Rect.bottom)) > 0)
    SetScrollRange(hWnd, SB_VERT, 0, nVertDifference, TRUE);
   else
    SetScrollRange(hWnd, SB_VERT, 0, 0, TRUE);
  SelectObject(hCompatibleDC, hOldBitmap);
  DeleteDC(hCompatibleDC);
  EndPaint(hWnd,&PaintStruct);
  return 0;
 case WM_VSCROLL:
  switch(LOWORD(wParam))
   {
    case SB_LINEDOWN:
     if(nVertPosition < nVertDifference)
       nVertPosition++;
     break;
    case SB_LINEUP:
     if(nVertPosition > 0)
       nVertPosition--;
     break;
    case SB_THUMBTRACK:
     nVertPosition = HIWORD(wParam);
     break;
   }
  SetScrollPos(hWnd, SB_VERT, nVertPosition, TRUE);
  InvalidateRect(hWnd, NULL, TRUE);
  return 0;
 case WM_HSCROLL:
  switch(LOWORD(wParam))
   {
    case SB_LINEDOWN:
     if(nHorizPosition < nHorizDifference)
       nHorizPosition++;
     break;
    case SB_LINEUP:
     if(nHorizPosition > 0)
```

```
            nHorizPosition--;
            break;
          case SB_THUMBTRACK:
            nHorizPosition = HIWORD(wParam);
            break;
        }
        SetScrollPos(hWnd, SB_HORZ, nHorizPosition, TRUE);
        InvalidateRect(hWnd, NULL, TRUE);
        return 0;
      case WM_DESTROY:
        DeleteObject(hBitmap);
        PostQuitMessage(0);
        return 0;
    }
    return DefWindowProc(hWnd,Message,wParam, lParam);
}
```

Вид окна, создаваемого программой, показан на рис. 4.

Если в окне не отображаются горизонтальная и вертикальная полосы прокрутки, необходимо уменьшить размеры окна по горизонтали и вертикали.

Рис. 4. Пример работы полос прокрутки

Появились полосы прокрутки? Давайте разберемся, благодаря чему это произошло.

Во-первых, если сравнить вызовы функций CreateWindow() в этой и предыдущей программах, то можно увидеть, что у окна появились два новых стиля - WS_HSCROLL и WS_VSCROLL. Эти стили определяют наличие у окна горизонтальной и вертикальной полос прокрутки соответственно. Первый шаг сделан. Этот шаг можно было бы сделать и по-другому, определив полосы прокрутки как дочерние окна, но о дочерних окнах мы будем говорить позже. Разница между полосами прокрутки, являющимися частью окна, и полосами прокрутки - дочерними окнами состоит в том, что дочерние окна имеют встроенный клавиатурный интерфейс, позволяющий воздействовать на полосу прокрутки с помощью клавиатуры. Встроенным полосам прокрутки, к сожалению, досталось только управление с помощью курсора мыши.

Теперь необходимо определить диапазон прокрутки, который определяет число шагов между крайними позициями бегунка (слайдера). По умолчанию для полос прокрутки, являющихся частью окна, этот диапазон определен от 0 до 100. Для того чтобы изменить диапазон прокрутки, необходимо вызвать функцию SetScrollRange(), которая в файле winuser.h определена следующим образом:

```
WINUSERAPI BOOL WINAPI SetScrollRange(HWND hWnd, int nBar,
                                      int nMinPos, int nMaxPos,
                                      BOOL bRedraw);
```

Первый аргумент функции - хэндл окна, которому принадлежат полосы прокрутки. Второй аргумент определяет, для какой полосы прокрутки (вертикальной или горизонтальной) устанавливается диапазон. В данном случае этот аргумент может принимать значение SB_VERT или SB_HORZ, что определяет работу с вертикальной или горизонтальной полосой прокрутки. Третий и четвертый аргументы непосредственно указывают нижнюю и верхнюю границу диапазона прокрутки. Пятый аргумент представляет собой флаг, определяющий, нужно ли перерисовывать полосу прокрутки после определения диапазона. TRUE - полоса прокрутки перерисовывается, FALSE - перерисовка не нужна. Заметьте, что если диапазон прокрутки определен от 0 до 0, то полоса прокрутки становится невидимой. Это свойство используется и в приведенной выше программе. В том случае, когда размеры окна превышают размеры отображаемого bitmap'а, у полос прокрутки устанавливается диапазон от 0 до 0, следовательно, полоса прокрутки скрывается.

В данном случае с помощью функции SetScrollRange() диапазон прокрутки определен как разность между размером bitmap'а и размером окна по вертикали и по горизонтали, т. е. шаг полосы прокрутки соответствует одному пикселю.

Воздействовать на полосы прокрутки можно по-разному: во-первых, можно щелкнуть клавишей мыши на стрелах, расположенных по краям полосы; во-вторых, можно щелкнуть на полосе выше или ниже слайдера. Наконец, можно перетащить слайдер на другое место. Все эти воздействия приводят к тому, что оконная функция окна, которому принадлежат полосы прокрутки, получает сообщение WM_VSCROLL (если действия производились вертикальной полосой) или WM_HSCROLL (реакция на воздействие на горизонтальную полосу).

Характер воздействия оконная функция может определить по параметрам сообщения. Младшее слово wParam, которое и определяет характер воздействия на полосу прокрутки, может принимать значения, приведенные в табл. 10. В таблице показано, что прокрутка при нажатии клавиши мыши в некоторых случаях производится на одну строку и одну страницу. В данном случае необходимо осознать, что понятия «строка» и «страница» ничего общего с текстовой строкой и страницей не имеют. Этими понятиями я заменил условные единицы, на которые прокручивается изображение в окне. К примеру, в приведенной программе строке соответствует один пиксель, а понятие страницы вовсе не определено (что есть страница для картинки?).

После ознакомления с таблицей становится ясно, какие «LOWORD(wParam)» должна обрабатывать прикладная программа.

Старшее слово wParam используется только в тех случаях, когда LOWORD(wParam) равен SB_THUMBPOSITION или SB_THUMBTRACK. В этих случаях оно хранит позицию слайдера. В остальных случаях это значение не используется.

В тех случаях, когда полосы прокрутки реализованы как дочерние окна, lParam содержит хэндл окна полосы прокрутки. Если полоса реализована как часть окна, этот параметр не используется.

После того, как мы зафиксировали факт произведенного с полосой прокрутки действия и характер действия, программа должна правильно отреагировать на него и при необходимости изменить позицию слайдера в соответствии с произведенным воздействием. Делается это с помощью обращения к функции SetScrollPos(), которая следующим образом описана в файле winuser.h:

```
WINUSERAPI int WINAPI SetScrollPos(HWND hWnd, int nBar, int nPos,
                                   BOOL bRedraw);
```

Таблица 10. Идентификаторы характеров воздействия на полосы прокрутки

Параметр	Значение	Описание
SB_LINEUP	0	Используется только с WM_VSCROLL; щелчок мышью на стрелке вверх; приводит к прокрутке на одну «строку» вверх
SB_LUNELEFT	0	Используется только с WM_HSCROLL, щелчок мышью на стрелке влево; приводит к прокрутке на одну «колонку» влево
SB_LINEDOWN	1	Используется только с WM_VSCROLL, щелчок мышью на стрелке вниз; приводит к прокрутке на одну «строку» вниз
SB_LINERIGHT	1	Используется только с WM_HSCROLL, щелчок мышью на стрелке вправо; приводит к прокрутке на одну «колонку» вправо
SB_PAGEUP	2	Используется только с WM_VSCROLL, щелчок мышью на полосе прокрутки выше слайдера; приводит к прокрутке на одну «страницу» вверх
SB_PAGELEFT	2	Используется только с WM_HSCROLL, щелчок мышью на полосе прокрутки левее слайдера; приводит к прокрутке на одну «страницу» влево
SB_PAGEDOWN	3	Используется только с WM_VSCROLL, щелчок мышью на полосе прокрутки ниже слайдера; приводит к прокрутке на одну «страницу» вниз
SB_PAGERIGHT	3	Используется только с WM_HSCROLL, щелчок мышью на полосе прокрутки правее слайдера, приводит к прокрутке на одну «страницу» вправо
SB_THUMBPOSITION	4	Перетаскивание слайдера закончено, пользователь отжал клавишу мыши
SB_THUMBTRACK	5	Слайдер перетаскивается с помощью мыши, приводит к перемещению содержимого экрана
SB_TOP	6	Используется только с вертикальными полосами прокрутки, реализованными как дочерние окна, пользователь нажал клавишу «Home»
SB_LEFT	6	Используется только с горизонтальными полосами прокрутки, реализованными как дочерние окна, пользователь нажал клавишу «Home»
SB_BOTTOM	7	Используется только с вертикальными полосами прокрутки, реализованными как дочерние окна, пользователь нажал клавишу «End»

Окончание табл. 10

Параметр	Значение	Описание
SB_RIGHT	7	Используется только с горизонтальными полосами прокрутки, реализованными как дочерние окна, пользователь нажал клавишу «End»
SB_ENDSCROLL	8	Пользователь отпустил клавишу мыши после удержания ее нажатой на стрелке или на полосе прокрутки

Первый аргумент - это хэндл окна, содержащего полосу прокрутки (в том случае, если полоса прокрутки реализована как часть окна), второй аргумент может принимать значение SB_VERT или SB_HORZ (об этих значениях говорилось выше), третий аргумент определяет, в какую позицию должен быть установлен слайдер. И наконец, четвертый аргумент определяет, нужно ли перерисовывать полосу прокрутки после установки слайдера. Если последний аргумент равен TRUE, то полоса прокрутки будет перерисована.

Для того чтобы в соответствии с новой позицией слайдера изменилось изображение в рабочей области, окну необходимо послать сообщение WM_PAINT, которое заставит окно перерисоваться. В программе, приведенной выше, сообщение WM_PAINT окну посылается с помощью вызова функции InvalidateRect(). Из этого следует, что код обработки сообщения WM_PAINT в оконной функции должен разрабатываться с учетом того, что содержимое окна может прокручиваться (скроллироваться).

И в заключение мне бы хотелось слегка посыпать голову пеплом. Любой хоть немного понимающий в программировании человек ужаснется, когда увидит, что я загружаю изображение из файла при каждой перерисовке окна (в программе, использующей функцию StretchBlt()). Это резко замедляет работу программы и занимает слишком много ресурсов. Но в данном случае целью было не написание программы, работающей оптимальным образом, а простая демонстрация того, что должна сделать программа для того, чтобы вывести на экран изображение.

Кстати, когда я начинал писать этот раздел, я думал, что уложусь в две-три страницы. Зато теперь надеюсь, что у читателя не осталось никаких вопросов относительно манипулирования изображениями. Естественно, что манипулированием изображениями графические возможности Windows отнюдь не исчерпываются. Но я достиг своей цели.

Теперь мне не придется постоянно просить читателя подождать, когда мы потом что-нибудь изучим. И теперь я готов к тому, чтобы продолжить изложение нашей «азбуки».

КОНТЕКСТ УСТРОЙСТВА И WM_PAINT

Я уже говорил, что в Windows окно само отвечает за перерисовку себя. Для того чтобы окно осуществило перерисовку, оно должно получить сообщение WM_PAINT. Каким образом осуществляется перерисовка?

Обычно используют один из трех методов:

- рабочая область может быть восстановлена, если ее содержимое формируется с помощью каких-либо вычислений;
- последовательность событий, формирующих рабочую область, может быть сохранена, а затем "проиграна" сколь угодно раз (имеются в виду метафайлы, но их рассмотрение выходит за рамки этой книги);
- можно создать виртуальное окно и направлять весь вывод в виртуальное окно, а при получении основным окном сообщения WM_PAINT копировать содержимое виртуального окна в основное.

Думаю, что читатель догадался, что в качестве виртуального окна используется контекст в памяти. Как его копировать, мы уже знаем. Но как рисовать на нем?

РИСОВАНИЕ ГРАФИЧЕСКИХ ПРИМИТИВОВ

Наверное, в подавляющем большинстве случаев читателю для работы с графикой будет достаточно того, о чем он только что прочитал. Теперь он сможет создавать изображение в графическом редакторе и копировать его в окно.

Но что делать читателю в том случае, когда необходимо это изображение создать в ходе самое программы? Как ему быть, если, к примеру, необходимо отобразить спектр радиосигнала, принятого его приемником? Или нарисовать график функции? Без краткого введения в основы рисования не обойтись. Если читателю не нужно в ближайшее время создавать изображение в программе, он может смело пропустить этот раздел и перейти к разделу «Взаимодействие программы с пользователем», а к этому разделу вернуться только в случае надобности.

Итак, Windows - операционная система, которая предоставляет пользователю графический интерфейс. Наша задача - научиться создавать в программе изображение, которое в дальнейшем мы будем использовать.

ТО, БЕЗ ЧЕГО РИСОВАНИЕ НЕВОЗМОЖНО

А невозможно рисование, во-первых, без инструментов.

Инструментами рисования в Windows являются перо (pen) и кисть (brush). Перо является инструментом для прорисовки линий, цвет и способ заполнения замкнутых графических объектов, таких, как круги, прямоугольники, эллипсы и так называемые регионы, определяются текущей кистью. Во-вторых, рисование невозможно без определения той точки, от которой мы начинаем прорисовку того или иного графического объекта. Обычно эта точка называется текущей графической позицией.

Установка текущей позиции

Для установки текущей позиции используется функция MoveToEx(). В файле заголовков wingdi.h эта функция описывается следующим образом:

```
WINGDIAPI BOOL  WINAPI MoveToEx(HDC, int, int, LPPOINT);
```

Первый аргумент - это контекст устройства, на котором мы будем рисовать, второй и третий - координаты точки, в которую мы устанавливаем текущую графическую позицию. Последний аргумент - указатель на структуру типа POINT, в которую функция запишет координаты старой текущей позиции. Структура типа POINT описана в файле windef.h и ее описание выглядит следующим образом:

```
typedef struct tagPOINT
{
  LONG x;
  LONG y;
} POINT, *PPOINT, NEAR *NPPOINT, FAR *LPPOINT;
```

Если при вызове функции указатель на структуру типа POINT равен NULL, то координаты старой текущей позиции не возвращаются.

Прорисовка одного пикселя

Прорисовать один пиксель в определенной позиции мы можем с помощью вызова функции SetPixel(), описанной в wingdi.h:

```
WINGDIAPI COLORREF WINAPI SetPixel(HDC, int, int, COLORREF);
```

Первые три аргумента очевидны - контекст устройства вывода и координаты прорисовываемого пикселя. Но что такое COLORREF?

Здесь следует пояснить, что каждый пиксель на экране состоит из тех микроточек - красной, зеленой и синей. Каждая из этих микроточек может светиться с интенсивностью от 0 (микроточка не светится) до 255 (максимальная яркость). Например, если светится только красная составляющая, то получаются цвета от темно-бордового (почти черного) до ярко красного. Комбинируя микроточки и их интенсивность, мы можем определить почти 17 миллионов цветов (будут ли они все поддерживаться на компьютере читателя, определяется видеоподсистемой компьютера, но это уже другой разговор). Обычно в таких случаях говорят об RGB - значениях цвета (red, green, blue - красный, зеленый, голубой).

Вернемся к COLORREF. Опять обратимся к заголовочному файлу, но на сей раз не к wingdi.h, а к windef.h:

```
typedef DWORD   COLORREF;
```

Понятно, что COLORREF - это двойное слово. Оно кодируется следующим образом:

```
0x00bbggrr
```

т. е. младший байт определяет интенсивность красного, второй - зеленого, третий - синего цвета. Старший байт должен быть нулевым. Для того чтобы облегчить жизнь пользователю, Microsoft в wingdi.h вставила макрос RGB:

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

С первого взгляда в этом не разобраться. Поэтому приведу пример определения цвета с RGB - значениями 0, 100, 200

```
RGB(0, 100, 200);
```

Стало понятнее, не правда ли?

При нормальном завершении функция возвращает предыдущее значение цвета пикселя. Если возвращаемое значение равно -1, то это говорит либо о возникновении ошибки, либо о том, что координаты пикселя вышли за пределы рабочей области окна.

Теперь мы готовы прорисовывать пиксели везде, где только можно. А если мы сумеем прорисовывать один пиксель, то сможем прорисовать и много. Но для того, чтобы рисовать, скажем, прямую, необходимо знать и реализовать в программе алгоритм прорисовки линий. То же можно сказать и о кругах, эллипсах и т. д. Думаю, что перспектива самостоятельной разработки этих алгоритмов читателю вовсе не улыбается. Но нет ничего страшного, команда разработчиков Win32 и здесь сняла проблему с программистов. Перейдем к достаточно обширной теме под названием

РИСОВАНИЕ ГРАФИЧЕСКИХ ПРИМИТИВОВ

Создание пера для рисования линий

Рисование графических примитивов производится с помощью перьев. В Windows'95 есть три предопределенных пера - черное (BLACK_PEN), белое (WHITE_PEN) и прозрачное (NULL_PEN). При создании окна по умолчанию ему присваивается черное перо. Хэндл каждого из них может быть получен с помощью функции GetStockObject(). Естественно, что программиста не может удовлетворить столь малое число перьев, поэтому для прорисовки линий можно воспользоваться пером, созданным в программе посредством вызова функции CreatePen(). Как всегда, обращаемся к файлам заголовков, в данном случае - к файлу wingdi.h:

```
WINGDIAPI HPEN    WINAPI CreatePen(int, int, COLORREF);
```

Первый аргумент определяет стиль кисти. В wingdi.h эти стили описаны достаточно образно. Для того чтобы сохранить стиль этого описания (не путать со стилем кисти) я включил его третьим столбцом в табл. 11.

Не правда ли, «seeing is believing»?

Второй аргумент функции CreatePen() - толщина пера в логических единицах. Если этот аргумент равен 0, то толщина пера делается равной одному пикселю.

Третий аргумент - цвет чернил. Теперь для того, чтобы мы могли использовать наше перо, необходимо сделать его текущим в контексте устройства. Делается это уже давно знакомой нам функцией SelectObject(). После того, как мы отработаем с пером, необходимо удалить его, вызвав функцию DeleteObject().

Мы создали перо. А теперь нам необходимо научиться рисовать примитивы.

Таблица 11. Возможные стили кисти

Стиль пера	Значение	Описание	Эффект
PS_SOLID	0		Сплошная линия
PS_DASH	1	-------	Пунктирная линия
PS_DOT	2	.....	Линия из точек
PS_DASHDOT	3	_._._._	Штрих-пунктирная линия (тире-точка)
PS_DASHDOTDOT	4	_.._.._	Штрих-пунктирная линия (тире - точка - точка)
PS_NULL	5		Прозрачное перо
PS_INSIDEFRAME	6		При рисовании замкнутой фигуры граница фигуры будет определяться по внешнему краю, а не по середине линии (если толщина пера более 1 пикселя)

Рисование линии

Нарисовать линию можно с помощью функции LineTo(). Она описана в файле wingdi.h:

```
WINGDIAPI BOOL WINAPI LineTo(HDC, int, int);
```

Первый аргумент - контекст устройства. Второй и третий аргументы - координаты точки, ДО КОТОРОЙ ОТ ТЕКУЩЕЙ ПОЗИЦИИ будет проведена линия. При успешном завершении функция возвращает TRUE.

Но здесь же возникает вопрос: где будет находиться текущая позиция после успешного выполнения функции? А будет она находиться там, где закончилась линия. Это сделано для того, чтобы легко можно было рисовать ломаные линии. В таком случае не нужно многократно вызывать функцию MoveToEx() для установления новой текущей позиции.

Рисование прямоугольника

Прямоугольник можно нарисовать, обратившись к функции Rectangle(). Её описание содержится в файле wingdi.h:

```
WINGDIAPI BOOL WINAPI Rectangle(HDC, int, int, int, int);
```

Аргумент первый понятен без объяснений - хэндл контекста устройства. Остальные аргументы - координаты верхнего левого и нижнего

правого углов прямоугольника. TRUE возвращается при нормальном завершении операции. Прямоугольник автоматически заполняется цветом и способом, определяемым текущей кистью.

Рисование эллипса

Для рисования эллипса необходимо вызвать функцию Ellipse(), которая в wingdi.h описывается следующим образом:

```
WINGDIAPI BOOL WINAPI Ellipse(HDC, int, int, int, int);
```

Первый аргумент - это, как всегда, контекст устройства. Для того чтобы понять, как определяется эллипс, предлагаю читателю обратиться к рис. 5.

Как видно из рисунка, эллипс ограничен прямоугольником. Именно через координаты этого прямоугольника и определяется прорисовываемый эллипс. Второй и третий аргументы - координаты левого верхнего угла прямоугольника (на рисунке обозначены как UpX, UpY), четвертый и пятый аргументы - координаты нижнего правого угла (на рисунке обозначены как LowX, LowY).

Окружность является частным случаем эллипса. И в данном случае, если мы определим прямоугольник, у которого ширина равна высоте, т. е. квадрат, вместо эллипса получим окружность.

Как эллипс, так и окружность после прорисовки заполняются цветом и атрибутами текущей кисти.

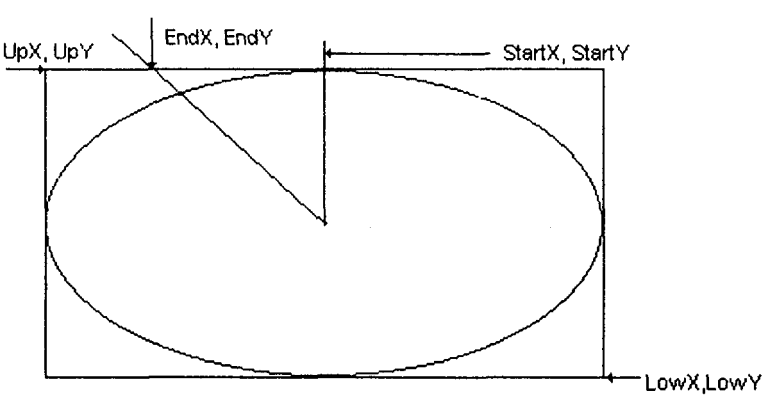


Рис. 5. Определение аргумента функции Ellipse()

Узнав, как рисуется эллипс, мы можем узнать, как рисуется прямоугольник с закругленными углами.

Рисование прямоугольника с закругленными краями

Прямоугольник с закругленными краями рисуется с помощью функции RoundRect(). Из файла wingdi.h добываем ее описание

```
WINGDIAPI BOOL  WINAPI RoundRect(HDC, int, int, int, int, int, int);
```

Первые пять аргументов полностью идентичны аргументам функции Rect(). Последние два аргумента содержат ширину и высоту эллипса, определяющего дуги. После прорисовки прямоугольник закрашивается текущей кистью. В случае успешного завершения функция возвращает TRUE.

Рисование дуги и сектора эллипса

Возьмем из файла wingdi.h описание функции Arc(), которая используется для рисования дуги:

```
WINGDIAPI BOOL  WINAPI Arc(HDC, int, int, int, int, int, int, int, int);
```

Первые пять аргументов полностью аналогичны аргументам функции Ellipse(). Непосредственно дуга определяется ещё двумя точками. Первая - начало дуги - находится на пересечении эллипса, частью которого является дуга, и прямой, проходящей через центр прямоугольника и точку начала дуги. На рис. 5 начало дуги обозначено StartX, StartY. Вторая - конец дуги - определяется аналогично. Конец дуги обозначен EndX, EndY. Таким образом, для прорисовки дуги необходимо сначала определить точки StartX, StartY и EndX, EndY, после чего прорисовывать дугу. Дуга прорисовывается против часовой стрелки.

У функции Pie(), которая применяется для рисования сектора эллипса, набор аргументов и их назначение абсолютно идентичны функции Arc().

Несколько слов о заполнении объектов

Как читатель уже знает, заполнение замкнутых графических объектов происходит с помощью текущей кисти. Программист может использовать предопределенную кисть, а может создать свою собственную, после чего сделать ее текущей с помощью функции SelectObject().

Простейшим видом кисти является так называемая сплошная кисть, которая создается с помощью функции CreateSolidBrush():

```
WINGDIAPI HBRUSH  WINAPI CreateSolidBrush(COLORREF);
```

Единственный аргумент этой функции - цвет кисти (может, лучше сказать не кисти, а краски?).

Штриховая кисть создается с помощью функции CreateHatchBrush():

```
WINGDIAPI HBRUSH  WINAPI CreateHatchBrush(int, COLORREF);
```

Первый аргумент этой функции - стиль штриховки. Возможные стили приведены в табл. 12.

Второй аргумент указывает цвет штриховки.

И наконец, с помощью функции CreatePatternBrush() мы можем создать кисть, которая при заполнении будет использовать bitmap. В wingdi.h она описана следующим образом:

```
WINGDIAPI HBRUSH  WINAPI CreatePatternBrush(HBITMAP);
```

Уже по типу аргумента видно, что единственным аргументом этой функции является хэндл bitmap'а.

Эти три функции при успешном завершении возвращают хэндл созданной кисти. В том случае, если произошла какая-то ошибка, возвращаемое значение равно NULL.

Давайте закрепим те знания, которые мы получили, рассмотрев небольшую демонстрационную программу.

К большому моему сожалению, до изучения меню я не могу написать программу, в которой действия пользователя определялись бы его выбором. Мне сейчас придется написать программу, которая только демонстрирует вывод на экран различных графических примитивов.

Таблица 12. Стили штриховки

Стиль штриховки	Значение	Описание	Эффект
HS_HORIZONTAL	0	-----	Горизонтальная штриховка
HS_VERTICAL	1	|||||	Вертикальная штриховка
HS_FDIAGONAL	2	\\\\\	Наклонная слева направо штриховка
HS_BDIAGONAL	3	/////	Наклонная справа налево штриховка
HS_CROSS	4	+++++	Штриховка крестиком
HS_DIAGCROSS	5	xxxxx	Штриховка косым крестиком

ДЕМОНСТРАЦИОННАЯ ПРОГРАММА

Ниже приведен текст программы, которая использует основные функции для вывода на экран 10 000 пикселей, поверх них несколько линий разных стилей, после которых, в свою очередь, прорисовывает прямоугольники и эллипсы:

```
#include <windows.h>

LRESULT CALLBACK GraphDemoWndProc ( HWND, UINT, UINT, LONG );

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow )

 {
  HWND hWnd ;
  WNDCLASS WndClass ;
  MSG Msg;
  char szClassName[] = "GraphDemo";

/* Registering our window class */
/* Fill WNDCLASS structure */

  WndClass.style = CS_HREDRAW | CS_VREDRAW;
  WndClass.lpfnWndProc = GraphDemoWndProc;
  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 = szClassName;
  if ( !RegisterClass(&WndClass) )
    {
     MessageBox(NULL,"Cannot register class","Error",MB_OK);
     return 0;
    }

  hWnd = CreateWindow(szClassName, "Graph Demo",
                      WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                      CW_USEDEFAULT, CW_USEDEFAULT,
                      CW_USEDEFAULT, NULL, NULL,
                      hInstance,NULL);
  if(!hWnd)
    {
     MessageBox(NULL,"Cannot create window","Error",MB_OK);
     return 0;
    }
```

```
   /* Show our window */
   ShowWindow(hWnd,nCmdShow);
   UpdateWindow(hWnd);

/* Beginning of messages cycle */
   while(GetMessage(&Msg, NULL, 0, 0))
    {
     TranslateMessage(&Msg);
     DispatchMessage(&Msg);
    }
   return Msg.wParam;

  }

LRESULT CALLBACK GraphDemoWndProc (HWND hWnd, UINT Message,
                                   UINT wParam, LONG lParam )
  {

   HDC hDC, hCompatibleDC;
   PAINTSTRUCT PaintStruct;
   RECT Rect;
   HBITMAP hCompatibleBitmap, hOldBitmap;
   HPEN hOldPen;
   static HPEN Pens[5];
   HBRUSH hOldBrush;
   static HBRUSH Brushes[6];
   int i;

   switch(Message)
    {
     case WM_PAINT:
      randomize();
      for(i = 0; i <= 4; i++)
       {
         Pens[i] = (CreatePen(i, 1, RGB(random(255),random(255), random(255))));
         Brushes[i] = (CreateHatchBrush(i, RGB(random(255), random(255),
                                        random(255))));
       }
      GetClientRect(hWnd, &Rect);
      hDC = BeginPaint(hWnd, &PaintStruct);
      hCompatibleDC = CreateCompatibleDC(hDC);
      GetClientRect(hWnd, &Rect);
      hCompatibleBitmap = CreateCompatibleBitmap(hDC, Rect.right, Rect.bottom);
      hOldBitmap = SelectObject(hCompatibleDC, hCompatibleBitmap);
      PatBlt(hCompatibleDC, 0, 0, Rect.right, Rect.bottom, PATCOPY);
// Drawing of pixels
      for(i = 0; i <= 9999; i++)
       SetPixel(hCompatibleDC, random(Rect.right), random(Rect.bottom),
              RGB(random(255), random(255), random(255)));
// Drawing of lines
```

```
        for(i = 0; i <= 9; i++)
         {
          hOldPen = SelectObject(hCompatibleDC, Pens[random(4)]);
          MoveToEx(hCompatibleDC, random(Rect.right), random(Rect.bottom),
                   NULL);
          LineTo(hCompatibleDC, random(Rect.right), random(Rect.bottom));
          SelectObject(hCompatibleDC, hOldPen);
         }
// Drawing of rectangles
        for(i = 0; i <= 5; i++)
         {
          hOldBrush = SelectObject(hCompatibleDC, Brushes[random(4)]);
          Rectangle(hCompatibleDC, random(Rect.right),
                                   random(Rect.bottom),
                                   random(Rect.right),
                                   random(Rect.bottom));
          Ellipse(hCompatibleDC, random(Rect.right),
                                 random(Rect.bottom),
                                 random(Rect.right),
                                 random(Rect.bottom));
          SelectObject(hCompatibleDC, hOldBrush);
         }
        BitBlt(hDC, PaintStruct.rcPaint.left, PaintStruct.rcPaint.top,
               PaintStruct.rcPaint.right,
               PaintStruct.rcPaint.bottom,
               hCompatibleDC,
               PaintStruct.rcPaint.left,
               PaintStruct.rcPaint.top,
               SRCCOPY);
        for(i = 0; i <= 4; i++)
         {
          DeleteObject(Pens[i]);
          DeleteObject(Brushes[i]);
         }
        SelectObject(hCompatibleDC, hOldBitmap);
        DeleteObject(hCompatibleBitmap);
        DeleteDC(hCompatibleDC);
        EndPaint(hWnd, &PaintStruct);
        return 0;
      case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    }
  return DefWindowProc(hWnd,Message,wParam, lParam);
}
```

На рис. 6 показан вид окна, создаваемого программой. Следует учесть, что положение линий, прямоугольников и эллипсов - случайное. При перерисовке их положение, размер и стиль штриховки изменяется, по-

этому при повторном запуске программы в окне может быть другое изображение.

Думаю, что после всего того, что мы обсудили в этом разделе, при разборе программы не встретится трудностей. Предоставляю вам возможность разобрать эту программу самостоятельно. У читателя может возникнуть вопрос, для чего все эти сложности с созданием контекста в памяти, копированием его на действительный контекст и прочее. Цель единственная - показать технику работы с виртуальным окном. Весь вывод осуществляется в виртуальное окно (контекст в памяти), после чего одна из «могучих Blt», BitBlt() выполняет копирование содержимого виртуального окна на действительное окно. Как я уже говорил, обычно вывод в действительное окно (т. е. копирование контекста в памяти на действительный контекст) происходит при обработке WM_PAINT.

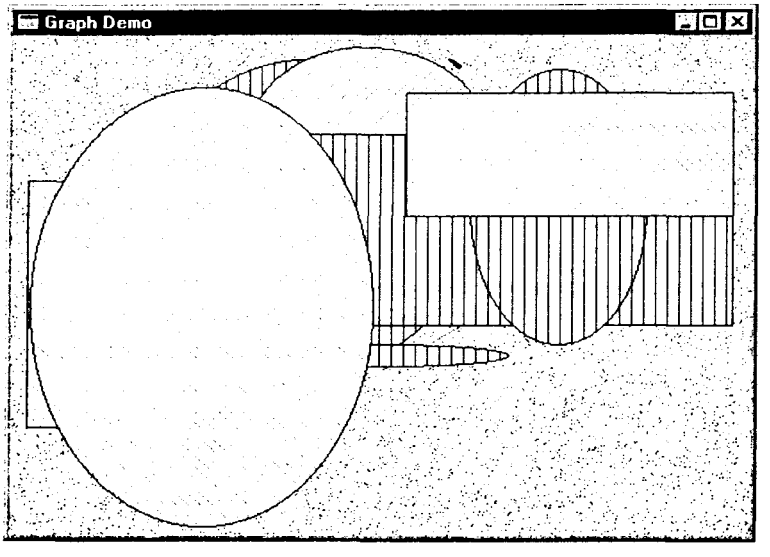


Рис. 6. Прорисовка геометрических объектов

ВЗАИМОДЕЙСТВИЕ ПРОГРАММЫ С ПОЛЬЗОВАТЕЛЕМ

НЕМНОГО О РЕСУРСАХ (ПРЕДИСЛОВИЕ К РАЗГОВОРУ)

ЧТО ТАКОЕ РЕСУРСЫ?

Выше упоминалось, что составной частью проекта, работа которого планируется в Windows, является файл определения ресурсов. Возникает вопрос: что же такое ресурсы, когда и в каких целях они используются?

У Windows, как уже говорилось, есть некоторые предопределенные данные (вспомним предопределенные курсоры, иконки и кисти). Точно так же, почти в каждой программе для Windows есть некоторые данные, которые определяются еще до начала работы программы, особым образом добавляются в выполняемый файл и используются при работе программы. Яркими примерами таких данных являются иконки и курсоры мыши. Кроме них, к числу ресурсов относятся:

- используемые в программе изображения;
- строки символов;
- меню;
- ускорители клавиатуры;
- диалоговые окна;
- шрифты;
- ресурсы, определяемые пользователем.

Следует отметить, что выполняемым файлом может быть файл программы .exe, файл динамической библиотеки .dll и другие бинарные файлы. Для удобства буду их называть bin-файлами.

Помимо того, что ресурсы определяются до начала работы программы и добавляются в bin-файл, у них есть еще одна характерная черта. При загрузке bin-файла в память, РЕСУРСЫ В ПАМЯТЬ НЕ ЗАГРУЖАЮТСЯ. Только в случае, если тот или иной ресурс требуется для работы программы, программа сама загружает ресурс в память.

Возможность использования того или иного атрибута в качестве ресурса не означает, что программист не может создавать эти атрибуты в программе. Яркий пример тому можно найти в работе старого доброго Program Manager'a. При перетаскивании иконки с места на место курсор меняет свою форму и принимает форму, подобную перетаскиваемой иконке. Естественно, что в этом случае курсоры определяются программой. Помимо этого, вспомним drag-and-drop в Explorer'е и изменение формы курсора при этом.

Еще одним примером являются динамические меню, т. е. меню, которые изменяют свои вид и предоставляемые возможности в зависимости от обстоятельств. Пример динамического меню будет приведен при изучении меню.

РЕСУРСЫ СТАНДАРТНЫЕ И НЕСТАНДАРТНЫЕ

Все ресурсы, заранее определенные в Win32 API, называются стандартными. Для работы с ними существуют специальные функции. Но именно эта стандартность и ограничивает возможности программиста. Стандарт, он и есть стандарт.

Для того чтобы можно было преодолеть эти ограничения, был создан особый тип ресурсов - определяемые пользователем ресурсы. Используя именно этот тип, мы можем предоставить в распоряжение программы практически любые данные. Но, как известно, бесплатным бывает только сыр в мышеловке. В данном случае платой за универсальность является усложнение программы, так как забота о манипулировании данными из ресурсов лежит уже не на системе, а на программе, использующей эти ресурсы. Программа может только получить указатель на данные ресурсов, загруженные в память средствами Windows. Дальнейшая работа с ними ложится ИСКЛЮЧИТЕЛЬНО на плечи программы!

ПОДКЛЮЧЕНИЕ РЕСУРСОВ К ИСПОЛНЯЕМОМУ ФАЙЛУ

Ресурсы создаются отдельно от файлов программы и добавляются в bin-файл при линковании программы. Подавляющее большинство ресурсов содержится в файлах ресурсов, имеющих расширение .RC. Имя файла ресурсов обычно совпадает с именем bin-файла программы. Так, если имя программы MYPROG.EXE, то имя файла ресурсов - MYPROG.RC.

Некоторые типы ресурсов (меню, например) можно описать на специальном языке и воспользоваться при этом обычным текстовым редактором, поддерживающим текст в формате ASCII. Другие ресурсы (иконки, курсоры, изображения) тоже описываются в текстовом виде, но частью их описания является последовательность шестнадцатиричных цифр, описывающих изображения. Можно, конечно, попробовать написать эту последовательность и в текстовом редакторе, но, наверное, в этом случае сложность создания ресурса приблизится к сложности написания программы, а возможно, и превысит ее. Обычно для создания ресурсов пользуются специальными средствами - редакторами ресурсов. Они позволяют создавать ресурсы, визуально контролировать правильность их создания, после чего сохранять их в файлах ресурсов.

Я часто использую «смешанный» способ редактирования ресурсов. Например, при визуальном редактировании диалоговых окон достаточно трудно точно установить элементы диалогового окна именно так, как хочется. Устанавливаю все элементы ПРИБЛИЗИТЕЛЬНО на те места, где они должны находиться, после чего сохраняю ресурсы в виде файла с расширением RC. Затем редактирую RC-файл как обычный текстовый файл, точно указывая при этом все размеры и позиции.

При создании RC-файлов программист может столкнуться с одной тонкостью. Некоторые ресурсы, такие, как иконки, курсоры, диалоговые окна, изображения (bitmap'ы) могут быть сохранены в отдельных файлах с расширениями .ico, .cur, .dlg, .bmp соответственно. В этом случае в RC-файлах делаются ссылки на упомянутые файлы.

Файл ресурсов создан - теперь его нужно откомпилировать. Компилируется он специальным компилятором ресурсов. Обычно имя компилятора ресурсов заканчивается на RC.EXE. В частности, в Borland 5.0 он называется BRC.EXE.

После компиляции файла ресурсов компилятором ресурсов создается новый файл, имеющий расширение .RES. Именно этот RES-файл используется линкером для добавления ресурсов в bin-файл. Следует отметить, что при необходимости RES-файлы могут создаваться и редакторами ресурсов. В каком формате создавать ресурсы и как присоединять их к исполняемому файлу, зависит от потребностей и привычек создающего ресурсы программиста.

Итак, в очередной раз постараемся подвести итог сказанному. Ресурсы создаются и включаются в bin-файл посредством выполнения следующих шагов (некоторые шаги могут быть опущены в зависимости от обстоятельств) (табл. 13).

Те программисты, которые при работе пользуются интегрированной средой, получают в некотором смысле преимущество. Во-первых, все эти шаги можно осуществить без выхода из интегрированной среды. Во-вторых, компиляция RC-файла и линкование полученного RES-файла можно выполнить автоматически.

После выполнения этих шагов в нашем bin-файле содержатся все необходимые данные нам данные, которые можно использовать (добавлять меню к окну, загружать курсоры, иконки, работать с диалоговыми окнами). Но все это - только описание порядка работы. В следующих разделах мы попробуем создать некоторые ресурсы. Не буду описывать работу с редакторами ресурсов. Во-первых, работа с ними достаточно проста, а во-вторых, описана в технических руководствах. Постараюсь описать синтаксис языка, который используется для создания сценариев (скриптов) ресурсов, после чего продемонстрировать, как работу с ресурсами можно

заменить и/или дополнить вызовами функций Win32 API. Рассмотрение мы построим следующим образом. Для каждого типа ресурсов сначала рассмотрим способ создания этого ресурса, подключения его к окну, а затем рассмотрим функции Win32, которые предназначены для работы с ресурсами.

В этом разделе мы будем рассматривать только те ресурсы, которые обеспечивают непосредственный диалог пользователя с программой. К их числу, прежде всего, относятся меню. Во-первых, именно с меню начинается знакомство с программой. Во-вторых, оценить функциональные возможности программы, можно просто взглянув на меню. То есть именно меню в большинстве случаев является визитной карточкой программы.

Кроме меню, наиболее часто для взаимодействия с пользователем используются диалоговые окна. Они, как правило, применяются для ввода данных и информирования программы о принятых решениях. При их рассмотрении, нам придется изучить элементы управления (controls) и общие элементы управления (common controls), научиться взаимодействовать с ними, устанавливать и считывать их состояние. Обращаю внимание читателя на следующее. Понимание работы меню и диалоговых окон очень важно. Зная принципы их работы, становится возможным написание программ для Windows, несущих какую-то полезную нагрузку.

Таблица 13. Последовательность создания файла ресурсов

Действие	Используемое средство
Создание RC-файла (при необходимости включающего ссылки на файлы с расширением .ico, .cur, .bmp, .dlg, .mnu и т. д.)	Редактор ресурсов (при необходимости может быть использован текстовый редактор и графические редакторы)
Редактирование RC-файла в текстовом виде	Текстовый редактор
Компиляция RC-файла - получение RES-файла	Компилятор ресурсов
Добавление ресурсов, содержащихся в RES-файле, в bin-файл	Линкер

ПОДКЛЮЧЕНИЕ МЕНЮ К ОКНУ

В предыдущей главе мы написали и исследовали программу, создающую окно. В этой главе мы должны научиться подключать меню к нашему окну и, естественно, научиться реагировать на сообщения, исходящие от меню. Мы узнаем о том, какие типы меню и элементов меню бывают, изучим различные способы построения меню.

Любой, кто хоть немного работал в Windows, знает, что меню располагаются сразу под заголовком окна и позволяют пользователю осуществить выбор возможностей, предоставляемых программой. Помимо этого, существуют также всплывающие меню, которые могут появляться в любой точке экрана. Обычно их содержание зависит от того, на каком окне щелкнули клавишей мыши. Их изучение начнем с характеристик меню.

Давайте представим себе главное меню программы как древовидную структуру. Основная часть - корень нашего дерева - это непосредственно главное меню. Само по себе оно представляет только структуру в памяти, не отображается на экране, не содержит ни одного элемента, но хранит указатель на список структур, описывающих подключаемые к нему элементы и всплывающие (popup) меню. Если читатель вспомнит свой опыт работы с Windows, он согласитсяс тем, что в окне в качестве основного меню отображается именно набор popup-меню. В свою очередь, popup-меню должно указать на список структур очередного, более низкого уровня и т. д. Конечные элементы меню никаких указателей на списки не имеют, но хранят некий идентификатор действия (назовем его идентификатором элемента меню), которое должна произвести программа при выборе данного элемента меню. Используя эти структуры, мы можем построить меню практически любой глубины и сложности.

Эта многоуровневая древовидная структура описывается в файле ресурсов. Описание меню имеет вид:

```
MenuName      MENU      [параметры]  // ← это - главное меню
  {
    Описание всех popup-меню и элементов меню второго уровня
  }
```

В данном случае MenuName - это имя создаваемого нами меню. Слово MENU обозначает начало определения меню. Параметры мы пока, до изучения работы с памятью, расматривать не будем.

В Win32 API для описания меню существуют два ключевых слова. Первое - POPUP - специфицирует всплывающее меню. Второе - MENUITEM - описывает обычный элемент меню.

Всплывающие меню описывается следующим образом:

```
POPUP      «Имя»      [,параметры]  // ← описание popup-меню
  {
    Описание всех popup-меню и элементов очередного уровня
  }
```

У конечного элемента меню в его описании есть еще одна характеристика - тот самый идентификатор действия:

```
MENUITEM    «Имя»,     MenuID     [,параметры]
```

В обоих случаях «Имя» - это тот текст, который будет выведен на экран при отображении меню (обратите внимание - при описании главного меню выводимого на экран текста нет!). В том случае, когда вместо имени окна записано слово SEPARATOR (без кавычек!), на ме сте элемента меню появляется горизонтальная линия. Обычно эти горизонтальные линии (сепараторы или разделители) используются для разделения элементов подменю на логические группы (логика определяется только программистом и никаких особенностей не содержит).

Если в имени меню встречается символ «&», то следующий за амперсандом символ на экране будет подчеркнут одинарной чертой. Этот элемент меню можно будет вызывать с клавиатуры посредством одновременного нажатия клавиши Alt и подчеркнутого символа.

Таблица 14. Параметры, описывающие элемент меню в файле ресурсов

Флаг	Значение
CHECKED	Рядом с именем элемента может отображаться небольшой значок, говорящий о том, что соответствующий флаг установлен
ENABLED	Элемент меню доступен
DISABLED	Элемент меню недоступен, но отображается как обычный
GRAYED	Элемент меню недоступен и отображается серым цветом
MENUBREAK	Горизонтальные меню размещают следующие элементы в новой строке, а вертикальные - в новом столбце
MENUBARBREAK	То же, что и предыдущее, но в случае вертикального меню столбцы разделяются вертикальной линией

MenuID - идентификатор действия. Он может быть передан функции окна, содержащего меню. Значение идентификатора определяется пользователем. Функция окна в зависимости от полученного MenuID производит определенные действия.

Параметры же описывают способ появления элемента на экране. Возможные значения параметров приведены в табл. 14.

Попробуем создать описание небольшого меню. Горизонтальное меню (menubar) позволит выбирать подменю «File», «Examples» и конечный элемент «Help». Подменю «File» будет содержать элементы «Open » и «Exit», разделенные горизонтальной линией, а подменю «Examples» - несколько конечных элементов.

Ниже приведен текст скрипта для этого меню:

```
MyMenu MENU
  {
     POPUP "&File"
       {
          MENUITEM "&Open", 101
          MENUITEM SEPARATOR
          MENUITEM "E&xit", 102
       }

     POPUP "&Examples"
       {
         POPUP "Example 1"
           {
             MENUITEM "1&1", 103
             MENUITEM "1&2", 104
           }
         POPUP "Example &2"
           {
             MENUITEM "2&1", 105
             MENUITEM "2&2", 106
           }
       }

     MENUITEM "&Help", 111
  }
```

Следует обратить внимание на то, что идентификаторы действия есть только у MENUITEM'ов. Popup-меню идентификаторов не содержат.

Теперь необходимо сделать так, чтобы меню стало доступным программе. В интегрированной среде это делается следующим образом:

- к проекту добавляется файл ресурсов (желательно, чтобы имя файла ресурсов совпадало с именем программы);

- в текст программы вносится изменение - при определении класса окна полю lpszMenuName структуры типа WNDCLASS присваивается указатель на строку, содержащую имя меню. В данном случае WndClass.lpszMenuName = «MyMenu»;
- производится перекомпиляция проекта.

Если читатель работает не в интегрированной среде, то ему необходимо до момента линкования откомпилировать ресурсы, а затем с помощью линкера присоединить их к исполняемому файлу. Попробуйте произвести эти действия с тем проектом, в котором вы создавали нашу первую программу. Если вы все сделали правильно, то у окна должно появиться меню, с которым можно немного поиграть. Попробуйте поэкспериментировать с описанием меню в файле ресурсов и видоизменить и непосредственно меню, и внешний вид popup-меню и элементов меню.

Таким образом, с помощью добавления к программе меню мы определили функциональность нашей программы. Конечно, тот пример, который здесь приведен, предназначен только для того, чтобы продемонстрировать возможности по управлению меню с помощью ресурсов. Более того, из сказанного можно сделать вывод, что возможности Win32 по управлению меню с помощью ресурсов достаточно скудны. Да, это так. Существует еще масса функций, позволяющих манипулировать меню. Мы приступим к их рассмотрению после того, как научимся реагировать на манипуляции, производимые с меню.

Реакция окна на сообщения от меню

Как уже было сказано выше, элементы в меню могут быть обычными, запрещенными и «серыми». Для пользователя обычные и запрещенные элементы выглядят одинаково, а текст в «серых» элементах напечатан серым шрифтом. Но только обычные элементы позволяют пользователю произвести выбор. Запрещенные и «серые» элементы меню могут быть только подсвечены, но с их помощью произвести выбор нельзя. Кроме этого, существуют отмечаемые элементы меню. В них слева от текста может находиться какой-либо значок. Если значок есть, то считают, что флажок, определяемый этим элементом, установлен. Если флажок сброшен, то значок отсутствует.

С другой стороны, в элементе меню может находиться как текст, так и картинка (bitmap). С точки зрения пользователя никакой разницы в применении меню с текстом или с картинкой нет.

Перед тем, как начать серьезный разговор о меню, напомню, что основному меню и всем всплывающим меню Windows присваивает хэндлы

(другими словами, все они являются структурами в памяти). Этот факт в дальнейшем будет играть важную роль.

Давайте попробуем поговорить о меню с точки зрения сообщений. При смене подсвеченного элемента меню (если, к примеру, пользователь «пробегает» по элементам меню с помощью клавиш со стрелками вверх и вниз) в оконную процедуру посылается сообщение WM_MENUSELECT. Это сообщение посылают все элементы меню. Когда же пользователь производит выбор (нажимает клавишу «Enter», к примеру), сообщение WM_COMMAND оконной процедуре посылают только обычные элементы меню. Запрещенные и «серые» элементы меню в этом случае никаких сообщений не посылают. В элементах wParam и lParam посылаемых сообщений хранится информация, достаточная для того, чтобы программа смогла определить, какие действия ей необходимо выполнить случае выбора пользователем того или иного элемента меню.

Вспомним, что помимо обычного меню у окна в большинстве случаев есть еще и системное меню. Сказанное относится и к системному меню. Отличие между обычным меню и системным состоит в том, что оконной процедуре посылаются сообщения WM_SYSMENUSELECT и WM_SYSCOMMAND. Кроме этого, сообщения WM_SYSCOMMAND оконная процедура получает и в случае нажатия кнопок минимизации, максимизации и закрытия окна, которые находятся не в системном меню, а в правом углу заголовка окна.

Рассмотрим параметры сообщения WM_MENUSELECT более подробно. В младшем слове wParam оконная процедура получает сведения о том, какой элемент стал подсвеченным. Если учесть, что макросы LOWORD() и HIWORD() выделяют соответственно младшее и старшее слово 32-битного аргумента, и назвать источник сообщения uItem, то можно записать:

```
uItem = (UINT) LOWORD(wParam);
```

В зависимости от обстоятельств смысл uItem различается:

- если подсвеченный элемент является конечным и не влечет за собой вызов popup-меню, то uItem содержит идентификатор элемента меню;
- если подсвеченный элемент при выборе влечет за собой вызов popup-меню, то uItem содержит номер (индекс) этого элемента в том меню, в котором оно находится;

В старшем слове wParam содержатся характеристики подсвеченного элемента. Аналогично предыдущему,

```
fuFlags = (UINT) HIWORD(wParam);
```

Возможные значения fuFlags приведены в табл. 15.

Таблица 15. Характеристики подсвеченного элемента меню

Флаг	Значение	Описание
MF_BITMAP	0x00000004L	Вместо строки в качестве элемента меню применяется bitmap
MF_CHECKED	0x00000008L	Элемент отмечаемый (со «значком»)
MF_DISABLED	0x00000002L	Элемент запрещен
MF_GRAYED	0x00000001L	Элемент запрещен и отображается серым цветом
MF_HILITE	0x00000080L	Элемент подсвечен
MF_MOUSESELECT	0x00008000L	Элемент выбран мышью
MF_OWNERDRAW	0x00000100L	За прорисовку элемента отвечает не система, а программа
MF_POPUP	0x00000010L	Элемент вызывает появление popup-меню более низкого уровня
MF_SYSMENU	0x00002000L	Элемент из системного меню

lParam содержит в себе хэндл того меню, которому принадлежит подсвеченный элемент. Обозначив хэндл меню как hMenu, получим:

```
hMenu = (HMENU) lParam;
```

Теперь пришла очередь рассмотрения сообщения WM_COMMAND. Как и в случае с WM_SELECTMENU, младшее слово wParam содержит сведения об источнике сообщения. Так как сообщение WM_COMMAND посылается только конечными элементами меню, то в младшем слове wParam содержится идентификатор выбранного элемента меню. На языке C

```
wID = LOWORD(wParam);
```

Старшее слово wParam указывает, от какого управляющего элемента пришло сообщение. Если сообщение пришло от меню, то это слово равно нулю, т. е.

```
wNotifyCode = HIWORD(wParam) = 0;
```

lParam в случае сообщения от меню ВСЕГДА равно NULL!

Теперь мы знаем вполне достаточно, чтобы написать программу, реагирующую на манипуляции с меню. Но я хотел бы, чтобы читатель набрался терпения и изучил еще одну тему. Ранее мы пришли к выводу о том, что возможности языка управления ресурсами достаточно скудны. Тем не менее, в Win32 существует множество функций, позволяющих манипулировать меню. Остановимся на этих функциях и выясним, каким образом можно реализовать меню без обращения к ресурсам.

Меню без использования ресурсов

Перед тем, как начать рассмотрение функций, предназначенных для работы с меню, я хотел бы сказать, что являюсь сторонником комбиниро-

ванного использования меню, определенного в виде ресурса, и функций Win32. Решение об использовании того или иного способа должен принимать программист в соответствии с задачами, которые должна решать разрабатываемая программа. Если в примере я придерживаюсь какого-то одного способа, то читатель должен понимать, что пример - это всего-навсего демонстрация возможностей Win32, а не призыв всегда и везде делать именно так, как сделано в предлагаемом примере. Еще раз повторю - программист свободен в выборе применяемых технологий.

Как было сказано выше, меню имеет строгую древовидную структуру, которая начинается с меню первого уровня (оно обычно называется главным меню программы или menubar'ом и располагается сразу под заголовком окна). К этому меню первого уровня могут быть присоединены как конечные элементы меню, так и элементы, выбор которых приводит к появлению так называемых всплывающих (popup) меню, к которым, в свою очередь, присоединяются элементы очередного уровня и т. д. Перед началом создания меню вся его структура должна быть тщательно продумана. Неплохо, если бы программист имел перед глазами графическое представление этого меню. Если все предварительные вопросы решены, то мы готовы приступить к созданию меню.

Итак, для создания меню необходимо выполнить следующие действия:

- выбрать подменю самого низкого уровня, которые содержат только конечные элементы меню, и создать их с помощью функций CreateMenu() или CreatePopupMenu() в зависимости от потребностей. Эти функции возвращают хэндл созданного меню. Меню создается пустым;
- посредством функции AppendMenu() добавляем в них требуемые элементы;
- создаем меню следующего, более высокого уровня, и добавляем в них требуемые элементы и меню, созданные нами на предыдущем шаге;
- повторяем эти шаги до тех пор, пока создание всех подменю не будет закончено;
- создаем главное меню программы посредством использования функции CreateMenu();
- присоединяем созданные подменю самого высокого уровня к главному меню программы с помощью функции AppendMenu();
- присоединяем меню к окну посредством использования функции SetMenu();
- прорисовываем меню с помощью функции DrawMenuBar().

Если в ходе программы сложилась такая ситуация, что меню оказалось не присоединенным к окну, перед выходом из программы обяза-

тельно уничтожаем его, вызывая функцию DestroyMenu() (присоединенное к окну меню уничтожается автоматически при уничтожении окна).

Для того чтобы проиллюстрировать сказанное, давайте разберем небольшую программу. Я постарался написать эту программу так, чтобы в ней имелись основные типы элементов меню, и была бы возможность обработать выдаваемые меню сообщения. Чтобы придать программе функциональность, информация о получении сообщения будет выдаваться в строку состояния - небольшую область внизу окна (возможности строки состояния мы будем изучать позже). При запуске у окна возникает меню. Внизу экрана появляется строка состояния, в которой будет отображаться информация о выбранном элементе меню. Основное меню окна состоит из двух всплывающих меню, «File» и «Help», и элемента меню, в котором вместо строки отображается bitmap. В первом всплывающем меню находятся два элемента, «Enable exit» и «Exit», во втором - один элемент, «About», который остается запрещенным в течение всего периода существования окна. Кроме этого, элемент «Exit» при запуске объявляется «серым», т. е. из программы можно выйти только через системное меню. Однако в случае выбора элемента «Enable exit» «Exit» становится обычным, а вместо «Enable exit» возникает «Disable exit». При выборе элемента с bitmap'ом отображается окно сообщений с текстом о том, что выбран именно этот элемент. На этом возможности программы исчерпываются.

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

const IDM_Enable_Disable = 0;
const IDM_Exit = 1;
const IDM_About = 2;
const IDP_File = 3;
const IDP_Help = 4;

char* pMessages[]
 {
  "Enable or disable exit",
  "Exit from the program",
  "About this program",
  "File operations",
  "Help operations",
  "Menu example",
  "System menu"
 };
```

```
LRESULT CALLBACK MenuDemoWndProc ( HWND, UINT, UINT, LONG );

HWND hStatusWindow;
UINT wId;
HMENU hMenu,hFileMenu,hHelpMenu;
HINSTANCE hInst;

int APIENTRY WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                      LPSTR lpszCmdParam, int nCmdShow )

 {
  HWND hWnd ;
  WNDCLASS WndClass ;
  MSG Msg;

  hInst = hInstance;
/* Registering our window class */
/* Fill WNDCLASS structure */
  WndClass.style = CS_HREDRAW | CS_VREDRAW;
  WndClass.lpfnWndProc = (WNDPROC) MenuDemoWndProc;
  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 = "MenuExample";

  if ( !RegisterClass(&WndClass) )
    {
     MessageBox(NULL,"Cannot register class","Error",MB_OK);
     return 0;
    }

  hWnd = CreateWindow("MenuExample", "Program No 2",
                      WS_OVERLAPPEDWINDOW,
                      CW_USEDEFAULT,
                      CW_USEDEFAULT,
                      CW_USEDEFAULT,
                      CW_USEDEFAULT,
                      NULL, NULL,
                      hInstance,NULL);
  if(!hWnd)
    {
     MessageBox(NULL, "Cannot create window", "Error", MB_OK);
     return 0;
    }
  InitCommonControls();
  hStatusWindow = CreateStatusWindow(WS_CHILD | WS_VISIBLE,
```

```
                                        "Menu sample", hWnd,wId);
  if(!hStatusWindow)
    {
     MessageBox(NULL, "Cannot create status window", "Error", MB_OK);
     return 0;
    }

/* Try to create menu */
  AppendMenu( (hFileMenu=CreatePopupMenu()), MF_ENABLED, MF_STRING,
                IDM_Enable_Disable, "&Enable exit");
  AppendMenu(hFileMenu, MF_GRAYED | MF_STRING, IDM_Exit,"E&xit");
  AppendMenu((hHelpMenu=CreatePopupMenu()),MF_DISABLED MF_STRING,
                IDM_About, "&About");
  hMenu = CreateMenu();
  AppendMenu(hMenu, MF_ENABLED | MF_POPUP, (UINT) hFileMenu,
                "&File");
  AppendMenu(hMenu, MF_ENABLED | MF_POPUP, (UINT) hHelpMenu,
                "&Help");
  SetMenu(hWnd, hMenu);
/* Show our window */
  ShowWindow(hWnd, nCmdShow);
  UpdateWindow(hWnd);
  DrawMenuBar(hWnd);

/* Beginning of messages cycle */
  while(GetMessage(&Msg, NULL, 0, 0))
   {
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
   }
  return Msg.wParam;
 }

LRESULT CALLBACK MenuDemoWndProc (HWND hWnd, UINT Message,
                                        UINT wParam, LONG lParam )
 {
  RECT Rect;
  static UINT nFlag = MF_ENABLED;
  char* pContent[]
   {
    "Enable exit",
    "Disable exit"
   };
  static UINT nIndex = 0;
  static HBITMAP hBitmap;
  int nDimension;

  switch(Message)
   {
    case WM_CREATE:
```

```
        nDimension = GetSystemMetrics(SM_CYMENU);
        hBitmap = LoadImage(hInst, "msdogs.bmp", IMAGE_BITMAP,
                                nDimension * 2, nDimension, LR_LOADFROMFILE);
        AppendMenu(GetMenu(hWnd), MF_BITMAP, IDM_Bitmap, hBitmap);
        break;
      case WM_COMMAND:
        switch (wParam)
         {
          case IDM_Enable_Disable:
           EnableMenuItem(hFileMenu, IDM_Exit, MF_BYCOMMAND | nFlag);
           nFlag = ( nFlag == MF_ENABLED ) ? MF_GRAYED : MF_ENABLED;
           nIndex = ( nIndex == 0) ? 1 : 0;
           ModifyMenu(hFileMenu, IDM_Enable_Disable, MF_BYCOMMAND |
                        MF_STRING, IDM_Enable_Disable, pContent[nIndex]);
           break;
          case IDM_Exit:
           SendMessage(hWnd, WM_CLOSE, NULL, NULL);
           break;
         }
      case WM_SIZE:
        SendMessage(hStatusWindow, WM_SIZE, wParam, lParam);
        GetClientRect(hWnd, &Rect);
        return 0;
      case WM_MENUSELECT:
// Selection is losted
        if ( ((UINT) HIWORD(wParam) == 0xffff) & ((HMENU) lParam == 0) )
          {
           SendMessage(hStatusWindow ,SB_SETTEXT, (WPARAM) 0,
                        (LPARAM) pMessages[5]);
           return 0;
          }
        if ((UINT) HIWORD (wParam) & MF_SYSMENU)
          {
           SendMessage(hStatusWindow, SB_SETTEXT, (WPARAM) 0,
                        (LPARAM) pMessages[6]);
           return 0;
          }
        if ((UINT) HIWORD(wParam) & MF_POPUP)
          {
           SendMessage(hStatusWindow, SB_SETTEXT, (WPARAM) 0,
                        (LPARAM) pMessages[3 + LOWORD(wParam)]);
           return 0;
          }
        SendMessage(hStatusWindow, SB_SETTEXT, (WPARAM) 0, (LPARAM)
                    pMessages[LOWORD(wParam)]);
        return 0;
      case WM_DESTROY:
        DeleteObject(hBitmap);
        PostQuitMessage(0);
        return 0;
```

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

Листинг № 3. Программа, демонстрирующая возможности по манипулированию меню.

Вид окна, создаваемого программой, показан на рис. 7.

Как и в случае нашей первой программы для Windows, давайте рассмотрим эту программу. Естественно, включаем файлы заголовков Win32. Включение файла «commctrl.h» обусловлено вызовом функций для работы со строкой состояния, заголовки которых находятся в этом файле. Далее идут многочисленные описания и определения, объяснять которые я не буду, они сразу же становятся понятными при разборе программы. В функции WinMain() все ясно до момента вызова функции InitCommonControls(). Но и здесь нет ничего страшного. Эта функция вызывается всегда перед использованием библиотеки общих элементов управления, к которым относится и строка состояния. К меню и нашей задаче эта функция имеет весьма далекое отношение. Интерес вызывает фрагмент, который начинается после строки комментария /* Try to create menu */.

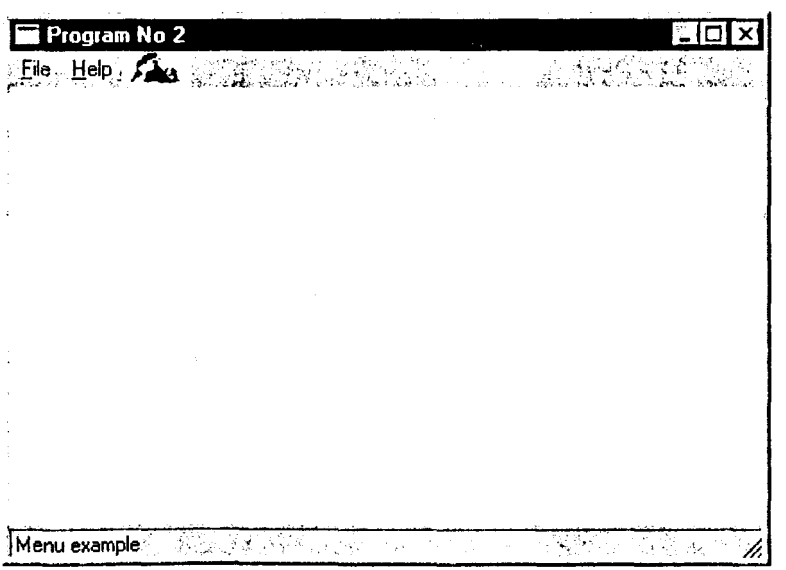


Рис. 7. Окно с меню, содержащим bitmap

Таблица 16. Битовые флаги, определяющие поведение и вид элемента меню

Флаг	Назначение
MF_BITMAP	Вместо строки символов в качестве элемента меню используется изображение (bitmap)
MF_CHECKED	Отмечаемый элемент меню
MF_DISABLED	Запрещенный, но не «серый» элемент меню
MF_ENABLED	Разрешенный элемент меню
MF_GRAYED	Запрещенный «серый» элемент меню
MF_MENUBARBREAK	То же, что и следующее, но вертикальные столбцы отделяются вертикальной чертой
MF_MENUBREAK	Очередной элемент меню размещают в новой строке (menubar) или в новом столбце
MF_OWNERDRAW	За прорисовку элемента отвечает программа, а не Windows
MF_POPUP	Выбор элемента влечет за собой появление меню более низкого уровня
MF_SEPARATOR	Горизонтальная черта
MF_STRING	В качестве элемента меню используется строка символов
MF_UNCHECKED	Неотмечаемый элемент меню

Как уже было сказано, сначала мы создаем меню с помощью вызова функции CreatePopupMenu(), которая возвращает нам хэндл созданного меню hFileMenu. В это меню мы с помощью функции AppendMenu() добавляем два элемента, «Enable exit» и «Exit». Функция AppendMenu() заслуживает того, чтобы поговорить о ней подробнее.

При вызове функции AppendMenu() она получает четыре аргумента. Аргумент первый - хэндл того меню, в которое добавляется элемент. Ничего сложного или интересного в этом вызове нет. Второй аргумент - комбинация битовых флагов, определяющих внешний вид и поведение добавляемого элемента меню. Перечень и назначение флагов приведены в табл. 16.

Заметим, что некоторые флаги не могут быть установлены одновременно. Например, MF_BITMAP, MF_STRING и MF_POPUP, MF_ENABLED, MF_DISABLED и MF_GRAYED.

Интерпретация третьего параметра зависит от того, установлен ли во втором параметре флаг MF_POPUP, т. е. является ли добавляемый элемент меню конечным элементом или влечет за собой вызов очередного меню. В первом случае - конечный элемент - параметр содержит идентификатор этого элемента. Если же добавляется меню, то параметр содержит хэндл добавляемого меню.

И последний параметр тоже интерпретируется в зависимости от установленных флагов. Установлен флаг MF_BITMAP - параметр содержит

хэндл bitmap'а. Установлен флаг MF_STRING - параметр содержит указатель на строку символов. Установлен MF_OWNERDRAW - параметр содержит информацию, используемую программой при прорисовке элемента.

Разобравшись с функцией AppendMenu(), мы можем не останавливаться на последующих ее вызовах и остановиться на вызове функции SetMenu().

Что означает «создать меню»? Это, как и в случае создания окна, означает всего лишь создание и последующее заполнение некоторых структур в памяти. После создания меню не принадлежит никакому окну и до поры до времени бесполезно блуждает в глубинах памяти. Для того чтобы меню могло выполнять свои функции, оно должно быть «закреплено» за одним из окон. Функция SetMenu() и привязывает меню к конкретному окну. Аргументы этой функции очевидны - хэндл закрепляемого меню и хэндл окна, к которому меню прикрепляется. После вызова этой функции указатель на меню включается в структуру окна и может нормально использоваться.

После отображения и прорисовки окна вызывается функция DrawMenuBar() для прорисовки меню. И у этой функции очевидно наличие одного аргумента - хэндла окна, которому принадлежит прорисовываемое меню.

К этому моменту мы произвели все действия, требующиеся для создания меню. Далее программа запускает цикл обработки сообщений и начинает посылать сообщения функции окна. Перейдем к рассмотрению оконной функции. Перед этим я прошу читателя не пугаться, увидев обращение к функции SendMessage(). Она только посылает окну (хэндл окна-адресата - первый аргумент) сообщение (второй аргумент) с заданными wParam (третий аргумент) и lParam (четвертый аргумент). В данном случае посылаются сообщения строке состояния для того, чтобы в ней отобразился текст, на который указывает lParam.

При этом оконная функция самостоятельно обрабатывает только четыре сообщения - WM_COMMAND, WM_MENUSELECT, WM_SIZE и WM_DESTROY. WM_DESTROY мы уже рассматривали. WM_SIZE в этом примере используется для того, чтобы при изменении размеров окна строка состояния могла бы нормально перерисоваться, т. е. сейчас это сообщение нас не интересует. В настоящий момент интерес представляют только сообщения WM_MENUSELECT и WM_COMMAND.

Начнем с рассмотрения WM_MENUSELECT. У этого сообщения есть одна особенность. Если поле fuFlags, т. е. старшее слово wParam, равно 0xffff и при этом поле hMenu (т. е. lParam) равно NULL, то это означает, что меню закрылось (не стало выделенных элементов), так как пользова-

тель нажал Escape или щелкнул мышкой где-нибудь вне меню. В этом случае в строке состояния появляется текст, описывающий назначение программы в целом («Menu example»). При получении сообщения от системного меню в строке состояния возникает «System menu». Во всех остальных случаях текст в строке состояния описывает назначение подсвеченного элемента меню. Непосредственно ход обработки этого сообщения ясен из листинга и особых пояснений не требует.

Но обработка WM_MENUSELECT - это всего лишь прелюдия к настоящей работе, которая происходит тогда, когда пользователь выбирает конечный элемент меню посредством нажатия клавиши «Enter» или щелкнет левой клавишей мышки на элементе. В этом случае оконная процедура получает сообщение WM_COMMAND. Для того чтобы определить, как реагировать на сообщение, мы должны проанализировать младшее слово wParam, которое хранит идентификатор элемента меню, и в зависимости от его значения предпринимать те или иные действия. В данном случае оконная функция может получать WM_COMMAND только с двумя идентификаторами - IDM_Enable_Disable и IDM_Exit. При получении последнего мы осуществляем выход из программы. При обработке первого я демонстрирую использование двух функций - EnableMenuItem() и ModifyMenu().

При получении WM_COMMAND, младшее слово wParam которого равно IDM_Enable_Disable, производятся следующие действия:

- с помощью функции EnableMenuItem() запрещается или делается доступным элемент «Exit»;
- с помощью функции ModifyMenu() изменяется текст элемента, выбор которого приводит к состоянию элемента «Exit».

Эти функции достаточно показательны и их разбор поможет читателю еще глубже понять функции, работающие с меню.

Функция EnableMenuItem() позволяет программисту изменять состояние элемента (разрешенный, запрещенный, «серый») меню по своему усмотрению. При вызове функции, ей передаются три аргумента. Первый аргумент - хэндл того меню, которому принадлежит элемент. В нашем случае меняется состояние элемента, находящегося в меню «File», хэндл которого hFileMenu. Второй аргумент определяет тот элемент, состояние которого изменяется, но каким способом происходит определение, а также в какое состояние переходит элемент, зависит от третьего аргумента, который в очередной раз представляет комбинацию битовых флагов.

Возможные флаги приведены в табл. 17.

Таблица 17. Флаги, используемые при вызове функции EnableMenuItem()

Флаг	Значение
MF_BYCOMMAND	*Изменяемый элемент меню определяется по его идентификатору*
MF_BYPOSITION	Изменяемый элемент меню определяется по его номеру (индексу) в меню
MF_ENABLED	После вызова функции элемент становится разрешенным
MF_DISABLED	После вызова функции элемент становится запрещенным
MF_GRAYED	После вызова функции элемент становится «серым»

После изменения состояния элемента «Exit» с разрешенного на серое и наоборот, необходимо изменить текст в элементе, от которого зависит это состояние. Это изменение производится посредством вызова функции ModifyMenu(), которой передаются пять аргументов. Первые два аргумента функционально подобны аргументам EnableMenuItem(), т. е. первый аргумент - хэндл меню, которому принадлежит изменяемый элемент, а второй аргумент определяет непосредственно изменяемый элемент. Можно было бы сказать, что и третий аргумент функционально подобен, но он представляет собой комбинацию флагов, во-первых, определяющих элемент, подлежащий изменению (MF_BYCOMMAND или MF_BYPOSITION), а во-вторых, определяющих состояние элемента после изменения (перечень этих флагов в точности соответствует приведенному в табл. 16). Четвертый аргумент указывает или идентификатор измененного элемента, или хэндл нового меню (если, конечно, в третьем аргументе установлен флаг MF_POPUP). И наконец, последний аргумент - новое содержание измененного элемента. В зависимости от того, какой флаг установлен в третьем аргументе (MF_BITMAP, MF_STRING или MF_OWNERDRAW), последний аргумент содержит хэндл bitmap'a, указатель на строку или информацию, используемую при прорисовке элемента.

Таким образом, с помощью только функций Win32 мы создали меню и изменили его состояние.

Надеюсь, что при чтении этого раздела и разборе приведенного примера читатель понял технические приемы работы с меню, и теперь может применять полученные знания при разработке собственных программ. Описанными функциями отнюдь не исчерпываются возможности Win32 по управлению меню. Например, в меню можно не добавлять, а вставлять элементы посредством функции InsertMenu(), функция DeleteMenu() удаляет элемент из меню, информацию о меню можно получить с помощью функций GetMenu(), GetMenuString(), GetMenuItemCount() и других.

В рамках этой книги нет возможности описать все функции работы с меню. Надеюсь, что читатель, получив начальные знания, проявит любознательность и сам найдет в системе помощи Win32 сведения о функциях, работающих с меню. Тем не менее, в примере, который приводится в разделе, посвященном созданию диалоговых окон, можно будет найти еще две функции, работающие с меню.

Изучение работы с меню на этом не заканчивается. Нераскрытым остался еще один вопрос - подключение акселераторов меню, который будет рассмотрен ниже.

АКСЕЛЕРАТОРЫ

Итак, мы научились создавать и манипулировать элементами меню. Но, к большому разочарованию тех, кто привык работать без мышки, у наших меню есть один серьезный недостаток. Выбор элементов мы можем производить только последовательно, входя в главное меню, подменю, подменю... и так до тех пор, пока не дойдем до нужного элемента. А у многих программ есть возможность обращаться к элементам меню напрямую посредством использования некоторых комбинаций клавиш. Возникает закономерный вопрос - как сделать так, чтобы и в наших программах была такая возможность?

Комбинации клавиш, которые при нажатии автоматически выбирают соответствующий им элемент меню (даже в тех случаях, когда оно не активно и не отображается), называются акселераторами. Это название (в переводе с английского акселератор означает ускоритель) выбрано достаточно удачно, ибо в тех случаях, когда пользователь запомнил их и привык к их использованию, ввод команд осуществляется намного быстрее, чем активизация меню и выбор этих команд.

Акселераторы являются одним из типов ресурсов, т. е. для того, чтобы использовать акселераторы, нам необходимо в файле ресурсов создать таблицу акселераторов. Она имеет следующий формат:

```
TableName   ACCELERATORS
 {
   Key1, MenuID1 [,тип] [,параметр]
   ....
   Keyn, MenuIDn [,тип] [,параметр]
 }
```

TableName - это определяемое пользователем имя таблицы акселераторов. *Key* определяет клавишу или комбинацию клавиш, при нажатии

которой происходит ввод команды. *Тип* определяет, является ли клавиша стандартной (это значение применяется по умолчанию) или виртуальной. *Параметр* может принимать одно из следующих значений: NOINVERT, ALT, CONTROL и SHIFT. Обычно при использовании акселераторных комбинаций меню отображается так, словно мы выбрали команду обычным способом. NOINVERT означает, что при использовании акселератора внешне меню на ввод команды никак не отреагирует, даже если будет активно и отображено. Присутствие ALT указывает, что для получения акселераторной комбинации одновременно с указанной клавишей необходимо нажать клавишу Alt. CONTROL говорит о том, что одновременно с клавишей должна нажиматься клавиша Control, а SHIFT требует одновременного с клавишей нажатия Shift.

В качестве клавиши можно указать либо ее символ в кавычках, либо код ASCII-символа, либо код виртуальной клавиши, определенной в файлах заголовков. При использовании ASCII-кода в качестве типа должно быть указано ASCII, а в случае применения виртуальной клавиши тип должен быть VIRTKEY. Виртуальная клавиша - это системно-независимый код, определенный для основного набора служебных клавиш. Этот набор включает клавиши F1-F12, стрелки и т. д. Коды виртуальных клавиш определены в заголовочных файлах. Все их идентификаторы начинаются с букв VK (Virtual Key). Разница между виртуальной клавишей и ASCII-символом с точки зрения пользователя состоит в том, что виртуальные клавиши не различают прописных и строчных букв, в отличие от ASCII-символов.

При определении акселераторов можно пойти на небольшую хитрость. Представим себе, что в качестве акселератора мы указали заглавную букву и, скажем, ALT. В этом случае нам придется одновременно нажимать три клавиши - букву, клавишу SHIFT (необходимо сделать символ заглавным!) и клавишу Alt. Таким образом, при указании в качестве основной клавиши заглавной буквы, можно определять трехклавишные акселераторы. Кстати, если мы хотим, чтобы для вызова команды использовалась клавиша Control, то можно символ в кавычках предварить знаком ^.

Примерами акселераторов в файле ресурсов могут служить следующие записи:

```
«a», IDM_The_First_Item, ALT     // определяется комбинация Alt-a
«A», IDM_The_Second_Item, ALT // определяется комбинация
                                  Shift-Alt-a
```

Таблица акселераторов должна быть загружена в память после создания окна до начала работы с меню. Поэтому желательно вызов функции

LoadAccelerator(), осуществляющей загрузку таблицы акселераторов, вставить в текст программы сразу же после создания окна.

Функция LoadAccelerator() при вызове должна получить два аргумента. Первый аргумент - это хэндл экземпляра программы, а второй - имя таблицы акселераторов. В результате вызова мы получаем хэндл нового объекта - таблицы акселераторов в памяти.

Но и это еще не все. Если рассуждать логически, то каждое нажатие акселераторной комбинации должно генерировать сообщение WM_COMMAND. Для этого акселераторы и создавались. Поэтому, даже после загрузки таблицы в память программа не сможет на них правильно реагировать, если мы не будем использовать функцию TranslateAccelerator(), которая преобразует сообщения от клавиатуры в сообщения WM_COMMAND. Описание этой функции можно найти в заголовочном файле winuser.h:

```
WINUSERAPI int WINAPI TranslateAcceleratorA(HWND hWnd,
                                            HACCEL hAccTable,
                                            LPMSG lpMsg);
WINUSERAPI int WINAPI TranslateAcceleratorW(HWND hWnd,
                                            HACCEL hAccTable,
                                            LPMSG lpMsg);
#ifdef UNICODE
#define TranslateAccelerator  TranslateAcceleratorW
#else
#define TranslateAccelerator  TranslateAcceleratorA
#endif // !UNICODE
```

Аргументы этой функции в достаточной степени очевидны. Первый аргумент - хэндл окна, которому принадлежит меню с акселераторами, второй - хэндл таблицы акселераторов, с помощью которой производится генерация сообщения WM_COMMAND, третий - указатель на сообщение. TranslateAccelerator() возвращает ненулевое значение, если нажата акселераторная комбинация и нуль в противном случае. Поэтому с учетом вызова этой функции цикл обработки сообщений должен выглядеть следующим образом:

```
while(GetMessage(&Msg, NULL, 0, 0))
    {
     if( !TranslateAccelerator(hWnd, hAccel, &Msg) )
        {
          TranslateMessage(&Msg);
          DispatchMessage(&Msg);
        }
    }
  return Msg.wParam;
```

Итак, с созданием таблиц акселераторов мы разобрались. Дело за малым - рассмотреть небольшой пример. В данном случае я не стал изобретать велосипед, и сделал следующее:

в программе, взятой из предыдущего примера, создал меню не в программе, а в файле ресурсов;

определил в файле ресурсов акселераторные комбинации;

добавил в цикл сообщений обработку акселераторных комбинаций.

В результате получились файлы, которые приведены ниже:

```
#define IDM_Enable_Disable 0
#define IDM_Exit 1
#define IDM_About 2
#define IDP_File 3
#define IDP_Help 4
```

Листинг № 4. Файл определений:

```
#include «menu.h»

MyMenu MENU
  {
    POPUP "&File"
      {
        MENUITEM "Enable exit\te", IDM_Enable_Disable, GRAYED
        MENUITEM "E&xit", IDM_Exit
      }

    POPUP "&Help"
      {
        MENUITEM "About\ta", IDM_About, DISABLED
      }
  }

MyMenu ACCELERATORS
  {
    «x», IDM_Exit,           ASCII
    «a», IDM_About,          ASCII
    «e», IDM_Enable_Disable, ASCII
    «d», IDM_Enable_Disable, ASCII
  }
```

Листинг № 5. Файл ресурсов:

```
#include <windows.h>
#include <commctrl.h>
#include "menu.h"
```

```
char* pMessages[] = {"Enable or disable exit", "Exit from the program",
                     "About this program", "File operations",
                     "Help operations", "Menu example", "System menu"};

long WINAPI HelloWorldWndProc ( HWND, UINT, UINT, LONG );

HWND hStatusWindow;
UINT wId;
HMENU hMenu,hFileMenu,hHelpMenu;
HINSTANCE hInst;

int APIENTRY WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                      LPSTR lpszCmdParam, int nCmdShow )
 {
  HWND hWnd ;
  WNDCLASS WndClass ;
  MSG Msg;
  HACCEL hAccel;
  hInst = hInstance;
/* Registering our window class */
/* Fill WNDCLASS structure */
  WndClass.style = CS_HREDRAW | CS_VREDRAW;
  WndClass.lpfnWndProc = (WNDPROC) HelloWorldWndProc;
  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 = "MyMenu";
  WndClass.lpszClassName = "MenuExample";

  if ( !RegisterClass(&WndClass) )
    {
     MessageBox(NULL,"Cannot register class","Error",MB_OK);
     return 0;
    }
  hWnd = CreateWindow("MenuExample", "Program No 2",
                      WS_OVERLAPPEDWINDOW,
                      CW_USEDEFAULT,
                      CW_USEDEFAULT,
                      CW_USEDEFAULT,
                      CW_USEDEFAULT,
                      NULL, NULL,
                      hInstance,NULL);
  if(!hWnd)
    {
     MessageBox(NULL,"Cannot create window","Error",MB_OK);
     return 0;
    }
```

```
  InitCommonControls();
  hStatusWindow = CreateStatusWindow(WS_CHILD | WS_VISIBLE,
                                     "Menu sample",
                                     hWnd,wId);
  if(!hStatusWindow)
    {
     MessageBox(NULL,"Cannot create status window","Error",MB_OK);
     return 0;
    }

/* Load the accelerators table */
  hAccel = LoadAccelerators(hInst,"MyAccelerators");
/* Show our window */
  ShowWindow(hWnd,nCmdShow);
  UpdateWindow(hWnd);
  hFileMenu = GetSubMenu(GetMenu(hWnd),0);

/* Beginning of messages cycle */
  while(GetMessage(&Msg, NULL, 0, 0))
   {
    if( !TranslateAccelerator(hWnd,hAccel,&Msg) )
      {
       TranslateMessage(&Msg);
       DispatchMessage(&Msg);
      }
   }
  return Msg.wParam;
 }

long WINAPI HelloWorldWndProc (HWND hWnd, UINT Message,
                               UINT wParam, LONG lParam )
 {
  RECT Rect;
  static UINT nFlag = MF_ENABLED;
  char* pContent[]
   {
    "&Enable exit\te",
    "&Disable exit\td"
   };
  static UINT nIndex = 0;

  switch(Message)
   {
    case WM_COMMAND:
     switch (wParam)
      {
       case IDM_Enable_Disable:
        EnableMenuItem(hFileMenu, IDM_Exit, MF_BYCOMMAND | nFlag);
        nFlag = ( nFlag == MF_ENABLED ) ? MF_GRAYED : MF_ENABLED;
        nIndex = ( nIndex == 0) ? 1 : 0;
```

```
            ModifyMenu(hFileMenu, IDM_Enable_Disable, MF_BYCOMMAND |
                        MF_STRING, IDM_Enable_Disable, pContent[nIndex]);
            break;
           case IDM_Exit:
            SendMessage(hWnd, WM_CLOSE, NULL, NULL);
            break;
          }
        case WM_SIZE:
         SendMessage(hStatusWindow,WM_SIZE,wParam,lParam);
         GetClientRect(hWnd,&Rect);
         return 0;
        case WM_MENUSELECT:
// Selection is losted
         if ( ((UINT) HIWORD(wParam) == 0xffff) & ((HMENU) lParam == 0) )
           {
            SendMessage(hStatusWindow, SB_SETTEXT, (WPARAM) 0,
                        (LPARAM) pMessages[5]);
            return 0;
           }
         if ((UINT) HIWORD(wParam) & MF_SYSMENU)
           {
            SendMessage(hStatusWindow, SB_SETTEXT, (WPARAM) 0,
                        (LPARAM) pMessages[6]);
            return 0;
           }
         if ((UINT) HIWORD(wParam) & MF_POPUP)
           {
            SendMessage(hStatusWindow, SB_SETTEXT, (WPARAM) 0,
                        (LPARAM) pMessages[3 + LOWORD(wParam)]);
            return 0;
           }
         SendMessage(hStatusWindow,SB_SETTEXT, (WPARAM) 0, (LPARAM)
                     pMessages[LOWORD(wParam)]);
         return 0;
        case WM_DESTROY:
         PostQuitMessage(0);
         return 0;
       }
      return DefWindowProc(hWnd,Message,wParam, lParam);
     }
```

Листинг № 5. Программа, демонстрирующая возможности акселераторов меню.

Мне бы хотелось, чтобы читатель поэкспериментировал с этой программой, попробовал переопределить акселераторы, и вновь провел несколько экспериментов.

Завершим рассмотрение темы о меню ответом на вопрос о том, как можно создать акселераторы без использования ресурсов.

Таблица 18. Возможные значения флагов поля fVirt структуры типа ACCEL

Флаг	Значение
FALT	При нажатии акселераторной комбинации должна быть нажата клавиша Alt
FCONTROL	При нажатии акселераторной комбинации должна быть нажата клавиша Control
FNOINVERT	Внешне меню не реагирует на нажатие акселераторной комбинации
FSHIFT	При нажатии акселераторной комбинации должна быть нажата клавиша Shift
FVIRTKEY	Поле key определяет виртуальную клавишу. Если это поле не установлено, то считается, что поле key содержим символ ASCII

Для создания таблицы акселераторов применяется функция CreateAcceleratorTable(), которой в качестве аргументов передаются адрес массива структур типа ACCEL и число структур в этом массиве.

Назначение полей структуры ACCEL должно быть понятно читателю, внимательно прочитавшему текущий раздел. В файле winuser.h эта структура описана следующим образом:

```
typedef struct tagACCEL {
  BYTE  fVirt;          /* Also called the flags field */
  WORD  key;
  WORD  cmd;
} ACCEL, *LPACCEL;
```

Если мы вспомним формат описания акселератора в файле ресурсов, то сразу можно догадаться о том, что поле cmd - это аналог поля MenuId, key соответствует Key, а значения поля fVirt являются комбинациями флагов (табл. 18), которые однозначно соответствуют полям Тип и Параметр.

И наконец, чтобы завершить тему об акселераторах, замечу, что при уничтожении окна автоматически из памяти удаляются только акселераторы, созданные с помощью функции LoadAccelerator(). В случае, если использовалась функция CreateAcceleratorTable(), программа должна сама заботиться об удалении таблицы из памяти. Для этого применяется функция DestroyAcceleratorTable(), в качестве аргумента которой передается хэндл таблицы акселераторов.

Мы завершили рассмотрение темы, связанной с меню и акселераторами. К этому моменту читатель должен быть готов к тому, чтобы самостоятельно использовать ПОЧТИ все возможности по управлению меню,

предоставляемые Win32. За пределами нашего внимания остался один пункт. Он связан с созданием в памяти структуры типа MENUITEMTEMPLATE и использования ее для создания меню посредством вызова функции LoadMenuIndirect(). В книге Чарльза Петцольда есть одна фраза, которая меня не только развеселила, но и подвигла на изучение этого совершенно бесполезного (это моя личная точка зрения, но тогда-то я еще не знал этого!) вопроса. Вот эта фраза: «If you're brave, you can try using it yourself (*Если ты смелый, ты можешь самостоятельно* попробовать использовать ее (функцию LoadMenuIndirect()).» Больше времени на разбор этой функции я терять не стану. Уважаемый читатель! Если вы не желаете прислушаться к моему совету, изучите, пожалуйста, третий способ создания меню самостоятельно.

Очередное ура! Мы прошли еще одну тему! Теперь мы умеем полностью определять порядок взаимодействия программы с пользователем через меню. Для того чтобы завершить раздел, посвященный непосредственному взаимодействию пользователя с программой, осталось всего ничего - начать и завершить тему о диалоговых окнах и обо всем, что с ними связано.

ДИАЛОГОВЫЕ ОКНА И ИХ ЭЛЕМЕНТЫ

В предыдущей главе мы разобрались с порядком создания меню и акселераторных комбинаций. Но любой работавший с Windows знает, что возможности программы, обеспечивающие взаимодействие с пользователем, отнюдь не ограничиваются рамками меню. Основным средством «общения» пользователя с программой являются диалоговые окна (их также называют диалогами). В этом разделе мы рассмотрим работу диалоговых окон и их взаимодействие не только с пользователем и программой, но и окнами более низкого уровня, называемыми элементами управления, которые выполняют большую часть черновой работы, незаметной не только пользователям, но и программистам.

Диалоговые окна можно классифицировать по двум критериям. Первый критерий - это модальность диалогового окна. Второй критерий, не всегда заметный, заслуживает особого внимания. Второй критерий фактически определяет, с одной стороны, возможности диалогового окна, а с другой стороны, ответственного за обработку сообщений посылаемых диалоговому окну. В подавляющем большинстве случаев обработку сообщений, адресованных диалоговому окну, производит функция диалогового окна (о ней речь впереди), но иногда диалоговое окно своей функции не имеет (она запрятана в «глубинах» системы) и всю обработку производит система. Возможности таких окон очень ограничены, в

основном они предназначены для выдачи сообщений пользователю. Об этом говорит даже их название - окна сообщений. Сейчас мы определим, что такое модальное и немодальное окно, рассмотрим процесс создания диалоговых модальных и немодальных диалоговых окон, остановимся на некоторых элементах управления диалоговыми окнами, а потом поговорим об окнах сообщений. Надеюсь, что даже после такого краткого экскурса в область диалоговых окон читатель сможет спокойно манипулировать окнами и элементами управления.

Модальные и немодальные диалоги

Диалоговые окна бывают модальными и немодальными.

Наиболее часто используются модальные окна. Эти окна не дают пользователю возможности работать с другими окнами, созданными приложением, породившим диалоговое окно, но разрешают переключаться на работу с другими приложениями. Для того чтобы пользователь мог продолжить работу с другими окнами своего приложения, необходимо завершить работу с диалоговым окном.

В особых случаях, представляющих угрозу системе и требующих немедленной реакции оператора, могут использоваться системные модальные окна. Эти окна не позволяют переключаться ни на какое другое окно. Естественно, что и применять системное модальное окно нужно с умом.

Немодальные диалоговые окна не требуют своего завершения для продолжения работы, и пользователь может во время работы с ними свободно переключаться на любое приложение.

РАБОТА С ДИАЛОГОВЫМИ ОКНАМИ

Диалоговое окно позволяет вводить и получать информацию, которую сложно или вовсе невозможно ввести через меню. Я уже говорил о том, что диалоговое окно имеет в своем составе некие элементы, окна более низкого уровня. Их называют элементами управления. Примером одного из таких элементов могу служить кнопки, которые, наверное, видел любой, хоть чуть-чуть поработавший с Windows. Так как без элементов управления диалоговые окна теряют всякий смысл, то рассмотрим, что такое

КНОПКИ, СПИСКИ И ПРОЧЕЕ...

Как уже было сказано, элементы управления - это ОКНА более низкого по отношению к диалоговому окну уровня. Предлагаю отметить то, что элементы управления никогда не могут использоваться как самостоятельные окна. Они всегда используются на фоне какого-то окна, которое

является для них родительским окном. Элементы управления, таким образом, всегда являются дочерними окнами, другими словами, у них всегда присутствует стиль WM_CHILD.

Как и любые другие окна, элементы управления могут получать и выдавать сообщения. Правда, это относится не ко всем элементам управления, но... Стоп! Давайте прервемся на секунду.

Мне бы хотелось обратить внимание читателя на один интересный момент. Для посылки сообщения обычно используют функции SendMessage() и SendDlgItemMessage(). Дело в том, что значение, которое возвращают эти функции, зависит только от того сообщения, которое они отправили. Таким образом, если вам необходимо узнать по возвращенному значению, что произошло в результате обработки того или иного сообщения, ищите описание возвращаемых значений не в описаниях функций, а в описаниях сообщений.

По умолчанию подавляющее большинство сообщений от элементов управления получает диалоговое окно, которому они принадлежат. Диалоговое окно должно каким-то образом их обрабатывать. Отсюда очередной вывод - у диалогового окна должна быть собственная оконная функция.

Каждому элементу управления присваивается идентификатор. При каком-либо воздействии на этот орган управления со стороны пользователя диалоговое окно получает сообщение, содержащее идентификаторы элемента и типа производимого пользователем действия. Диалоговая функция обрабатывает эти сообщения и выполняет соответствующие действия. Этот процесс происходит параллельно с обработкой сообщений в оконной функции. При этом нужно заметить, что в отличие от обычного окна, «нормальное» диалоговое окно не имеет своего цикла обработки сообщений. Цикл обработки сообщений запускается один раз при запуске программы.

Элементами управления могут быть кнопки (buttons), которые мы уже использовали в окнах сообщений, переключатели (check boxes), селекторы (radio buttons), списки (list boxes), комбинированные списки (combo boxes), линейки прокрутки (scroll bars) и статические элементы (statics). Все элементы в этом перечне относятся к категории базовых, и все они присутствовали и в Windows 3.x. На их основе Microsoft разработала серию новых элементов управления (common controls), которые позволили расширить возможности интерфейса с пользователем и улучшить внешний вид приложений. Мы рассмотрим как базовые, так и новые общие (как еще можно перевести на русский язык название «common controls»?) элементы управления.

Я уже упоминал. что основную часть работы диалогового окна выполняют элементы управления. Поэтому рассмотрим сначала вопрос о том, как может быть создано диалоговое окно, а потом на примерах - работу каждого типа элементов управления.

СОЗДАНИЕ ДИАЛОГОВОГО ОКНА

Диалоговое окно, как и меню, может быть создано несколькими способами: во-первых, с помощью описания его в файле ресурсов и, во-вторых, во время выполнения программы. Наиболее часто используется описание диалога в файле ресурсов. Лучше всего при создании диалога воспользоваться редактором ресурсов, с помощью которого может быть создан текстовый файл, содержащий описание диалогового окна. Ресурсы диалога в этом текстовом файле задаются оператором DIALOG, который имеет следующий формат:

```
DialogName  DIALOG [DISCARDABLE]        X, Y, Width, Height
CAPTION «Заголовок окна»
STYLE <Стили диалогового окна>
FONT n, <имя шрифта>

 {
  Описание элементов диалога
 }
```

В данном случае DialogName - это имя диалогового окна. Опция DISCARDABLE станет совершенно ясной при рассмотрении вопроса об организации памяти в Windows. Параметры X и Y - это координаты верхнего левого угла диалогового окна, Width и Height - ширина и высота диалога. STYLE описывает стили окна. Здесь могут использоваться как стили, применяемые для описания обычных окон (об этих стилях мы говорили при создании первой программы для Windows), так и стили, применяемые только в диалоговых окнах. Эти новые стили приведены в табл. 19.

Приведенных выше сведений вполне достаточно, чтобы написать заготовку диалогового окна в файле ресурсов. Но какой смысл описывать диалоговое окно, если в нем нет ни одного из элементов управления? Ведь даже закрыть такое диалоговое окно (если в нем, конечно, нет системного меню) невозможно! Значит, нам необходимо срочно научиться описывать эти элементы!

Я уже упоминал о том, что в «недрах» Win32 есть масса предопределенных объектов. В частности, там находятся и некоторые предопределенные классы окон. К таким классам относятся кнопки (класс «button»), списки (класс «listbox»), комбинированные списки (класс «combobox»), окна редактирования (класс «edit»), полосы прокрутки класс «scrollbar»), статические элементы (класс «static»). У каждого класса есть свой определенный набор стилей, которые определяют внешний вид и поведение элементов управления, относящихся к данному классу.

Управление окном каждого класса, а также получение информации от него производится с помощью обмена управляющими сообщениями. О действиях пользователей с ними элементы управления оповещают свои родительские окна через нотификационные сообщения. Предлагаю читателю запомнить это, так как мы еще неоднократно вспомним о предопределенных сообщениях.

Приступим к изучению элементов управления. Вспомним, что мы уже неоднократно встречались с кнопками. Давайте и начнем с описания обычных кнопок (buttons).

Таблица 19. Стили диалоговых окон

Стиль	Значение	Эффект
DS_ABSALIGN	0x0001L	Положение диалогового окна исчисляется в экранных координатах
DS_SYSMODAL	0x0002L	Создается системное модальное диалоговое окно
DS_3DLOOK	0x0004L	Создается диалоговое окно, имеющее зрительную иллюзию трехмерности
DS_FIXEDSYS	0x0008L	Вместо SYSTEM_FONT используется SYSTEM_FIXED_FONT
DS_NOFAILCREATE	0x0010L	Диалоговое окно создается, несмотря на то, что при его создании произошли ошибки
DS_LOCALEDIT	0x0020L	В 32-битных приложениях не используется
DS_SETFONT	0x0040L	Определяется шрифт, который будет применяться в диалоговом окне
DS_MODALFRAME	0x0080L	Создается модальное диалоговое окно
DS_NOIDLEMSG	0x0100L	
DS_SETFOREGROUND	0x0200L	Поместить диалоговое окно на передний план
DS_CONTROL	0x0400L	
DS_CENTER	0x0800L	Диалоговое окно помещается в центр рабочей области
DS_CENTERMOUSE	0x1000L	
DS_CONTEXTHELP	0x2000L	

Кнопки

Перед тем, как начать рассказ о кнопках, хочу предостеречь читателя. Дело в том, что можно использовать кнопки и в обычных окнах. Но они, как и большинство элементов управления, проектировались для использования именно в диалоговых окнах. Использование кнопок в обычных окнах не рекомендуется, ибо это увеличивает риск того, что программа будет работать неправильно.

Кнопка - это имитация на экране обычной кнопки или переключателя. В этом разделе под кнопками я также подразумеваю не только PushButtons (обычные нажимаемые кнопки), но и Check Boxes (обычно это небольшие квадратики, в которых можно установить или не установить пометку) и Radio Buttons (небольшие кружочки, В ОДНОМ из которых стоит точка). Пользователь может установить курсор мыши на кнопку, щелкнуть клавишей мыши - и кнопка пошлет диалоговому окну сообщение WM_COMMAND. То же произойдет и в случае, если пользователь сначала выберет кнопку клавишей Tab, а потом нажмет Enter.

В параметрах сообщения WM_COMMAND содержится информация, которой достаточно, чтобы диалоговое окно узнало, от какой кнопки пришло сообщение, какое действие требуется выполнить, и каким образом пользователь инициировал выдачу сообщения.

При этом необходимо отметить, что обычная кнопка (её называют PushButton) не помнит того, что с ней делали, т. е. она на короткое время становится нажатой, а затем возвращается в исходное состояние. Можно нажать кнопку десять раз подряд, и все десять раз она пошлет диалоговому окну одно и то же сообщение, если, конечно, кнопка не сделана запрещенной. CheckBox помнит о том, что он находится в одном из двух состояний - установленном или не установленном, некоторые CheckBox'ы могут находиться еще и в неопределенном состоянии.

Говорить о состоянии одной RadioButton бессмысленно. Дело в том, что RadioButton'ы предназначены для осуществления выбора одного из нескольких взаимоисключающих вариантов, поэтому можно говорить о группе (иногда говорят кластере) RadioButton'ов. В частности, для объединения RadioButton'ов в кластеры служит такой элемент управления, как группа (GroupBox). Обычно группы используются для группирования органов управления только для улучшения дизайна диалоговых окон. Что же касается RadioButton'ов, то без обрамляющей группы существование кластера не имеет смысла.

Формат описания кнопок в окне диалога достаточно прост:

CONTROL «Заголовок», ButtonID, class, styles, X, Y, Width, Height

X, Y, Width, Height - это все ясно. Все то же самое, что и при описании непосредственно диалогового окна. «Заголовок» - надпись на кнопке или рядом с кнопкой. ButtonID - идентификатор кнопки, т. е. значение, которое посылается диалоговому окну при нажатии кнопки в качестве LOWORD (wParam). Через HIWORD(wParam) диалоговое окно получает код нотификации, т. е. код того действия, которое произвел пользователь. Примерами действия пользователя могут служить нажатие клавиши Enter, двойной щелчок правой или левой клавишей мыши и так далее. А источник, т. е. хэндл инициировавшего сообщение окна, сообщения содержится в lParam (я напомню, что если сообщение приходит от меню, то lParam всегда равен 0, а если от акселератора - 1). Все легко и просто. Сложности начинаются при рассмотрении класса, определяющегося полем class, типа кнопки и стиля кнопки, которые определяется параметром style.

Для кнопок, вне зависимости от того, PushButton ли это, RadioButton или CheckBox, класс всегда определяется как «button».

Читатель, наверное, уже привык к тому, что ответы на большинство вопросов можно найти в файлах заголовков и файлах помощи Win32. Поэтому и сейчас, как всегда, смотрим в заголовочный файл winuser.h, выбираем оттуда стили кнопок, которые начинаются с букв BS_, и сводим их в табл. 20. Надоело изучать эти таблицы? Ничего, тяжело в учении - легко в бою! © А.В. Суворов, «Наука побеждать».

А теперь, когда мы изучили весь вопрос теоретически, попробуем разобраться со всем этим разнообразием кнопок и стилей. Напишем небольшую программу, в которой будут присутствовать все виды кнопок (но отнюдь не все виды стилей). От кнопок будем получать сообщения и обрабатывать их. Мы также научимся устанавливать кнопки в те или иные состояния. В предлагаемой программе ничего не делается, за исключением того, что демонстрируется, как устанавливать кнопки в то или иное состояние и считывать состояние, в котором кнопки находятся.

В программе создается диалоговое окно, имеющее кластер RadioButtons, состоящий из трех кнопок, три CheckBox'а, а также две PushButton. Состояния RadioButtons и CheckBoxes до отображения диалогового окна могут быть определены через меню. Затем проявляется обратная связь - состояния RadioButtons и CheckButtons определяют состояния элементов меню после закрытия диалога. PushButton с надписью «Cancel» приводит к закрытию диалогового окна.

Таблица 20. Стили кнопок

Флаг	Значение	Эффект
BS_PUSHBUTTON	0x00000000L	Создается обычная кнопка
BS_DEFPUSHBUTTON	0x00000001L	Создается обычная кнопка, которая срабатывает при нажатии «Enter» даже тогда, когда не выбрана
BS_CHECKBOX	0x00000002L	Создается CheckBox, при нажатии состояние автоматически не изменяется, забота об этом ложится на программу
BS_AUTOCHECKBOX	0x00000003L	Создается CheckBox, который автоматически меняет свое состояние при нажатии
BS_RADIOBUTTON	0x00000004L	Создается Radio Button, автоматически состояние не меняется
BS_3STATE	0x00000005L	То же, что и BS_CHECKBOX, но имеет три состояния - включенное, отключенное и неопределенное, автоматически состояние не меняет
BS_AUTO3STATE	0x00000006L	То же, что и предыдущее, но состояние меняется автоматически
BS_GROUPBOX	0x00000007L	Группа
BS_USERBUTTON	0x00000008L	Устаревший стиль, необходимо использовать BS_OWNERDRAW
BS_AUTORADIOBUTTON	0x00000009L	То же, что и RadioButton, но при нажатии состояние меняется автоматически
BS_OWNERDRAW	0x0000000BL	За прорисовку кнопки отвечает программа, а не система
BS_LEFTTEXT	0x00000020L	Текст помещается слева от RadioButton'а или CheckBox'а, то же, что и BS_RIGHTBUTTON
BS_TEXT	0x00000000L	Внутри или рядом с кнопкой отображается текст
BS_ICON	0x00000040L	Внутри кнопки или рядом с кнопкой отображается иконка
BS_BITMAP	0x00000080L	Внутри кнопки или рядом с кнопкой отображается bitmap
BS_LEFT	0x00000100L	Размещает текст у левого края прямоугольника, выделенного для размещения кнопки
BS_RIGHT	0x00000200L	Размещает текст у правого края прямоугольника, выделенного для размещения текста
BS_CENTER	0x00000300L	Размещает текст по горизонтали в центре прямоугольника, выделенного для размещения кнопки

Флаг	Значение	Эффект
BS_TOP	0x00000400L	Размещает текст у верхнего края прямоугольника, выделенного для размещения кнопки
BS_BOTTOM	0x00000800L	Размещает текст у нижнего края прямоугольника, выделенного для размещения кнопки
BS_VCENTER	0x00000C00L	Размещает текст по вертикали в центре прямоугольника, выделенного для размещения кнопки
BS_PUSHLIKE	0x00001000L	Делает CheckBox или RadioButton внешне похожими на PushButton
BS_MULTILINE	0x00002000L	При необходимости текст разбивается на несколько строк
BS_NOTIFY	0x00003000L	Разрешает посылку родительскому окну нотификационных сообщений BN_DBLCLK, BN_KILLFOCUS и BN_SETFOCUS
BS_FLAT	0x00008000L	Не добавляется имитация трёхмерности изображения элемента управления
BS_RIGHTBUTTON	0x00000020L	RadioButton или CheckBox размещаются справа от надписи (то же, что и BS_LEFTTEXT)

При работе программы любое изменение состояния кнопок приводит к выдаче сообщения в строке состояния диалогового окна.

Ниже приводен файл описаний, использующийся при работе демонстрационной программы:

```
#define IDM_EXIT              101
#define IDM_RadioButton1      102
#define IDM_RadioButton2      103
#define IDM_RadioButton3      104
#define IDM_CheckButton1      105
#define IDM_CheckButton2      106
#define IDM_CheckButton3      107
#define IDM_DisplayDialog     108
#define IDC_BUTTON1           201
#define IDC_BUTTON2           202
#define IDC_GROUPBOX1         203
#define IDC_RADIOBUTTON1      204
#define IDC_RADIOBUTTON2      205
```

```
#define IDC_RADIOBUTTON3        206
#define IDC_GROUPBOX2           207
#define IDC_CHECKBOX1           208
#define IDC_CHECKBOX2           209
#define IDC_CHECKBOX3           210
#define IDC_STATUSBAR           301
```

А теперь - основной файл программы:

```
#include <windows.h>
#include <commctrl.h>
#include "buttons.h"

HINSTANCE hInst;
HWND hWnd ;
int nRadioButtonId;
UINT uCheckBoxesState[3] = {MF_UNCHECKED, MF_UNCHECKED,
                            MF_UNCHECKED};

long WINAPI ButtonsExampleWndProc ( HWND, UINT, UINT, LONG );
BOOL CALLBACK ButtonsExampleDialogProc(HWND, UINT, WPARAM,
                                       LPARAM);

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow )
 {
  WNDCLASS WndClass ;
  MSG Msg;
  char szClassName[] = "ButtonsExample";
  hInst = hInstance;
/* Registering our window class */
/* Fill WNDCLASS structure */
  WndClass.style = CS_HREDRAW | CS_VREDRAW;
  WndClass.lpfnWndProc = ButtonsExampleWndProc;
  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 = "ButtonsExampleMenu";
  WndClass.lpszClassName = szClassName;

  if ( !RegisterClass(&WndClass) )
     {
      MessageBox(NULL,"Cannot register class","Error",MB_OK);
      return 0;
     }
  hWnd = CreateWindow(szClassName, "Button Use Example",
```

```
                        WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, NULL, NULL,
                        hInstance,NULL);
  if(!hWnd)
    {
     MessageBox(NULL,"Cannot create window","Error",MB_OK);
     return 0;
    }
  InitCommonControls();
/* Show our window */
  ShowWindow(hWnd,nCmdShow);
  UpdateWindow(hWnd);
/* Beginning of messages cycle */
  while(GetMessage(&Msg, NULL, 0, 0))
   {
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
   }
  return Msg.wParam;
 }

LRESULT CALLBACK ButtonsExampleWndProc (HWND hWnd, UINT Message,
                                        UINT wParam, LONG lParam )
 {
  static BOOL bFlag = FALSE;
  static HMENU hMenu1, hMenu2;
  int i;

  switch(Message)
   {
   case WM_CREATE:
    hMenu1 = GetSubMenu(GetSubMenu(GetMenu(hWnd), 1), 0);
    hMenu2 = GetSubMenu(GetSubMenu(GetMenu(hWnd), 1), 1);
    CheckMenuRadioItem(hMenu1, IDM_RadioButton1, IDM_RadioButton3,
                       IDM_RadioButton1, MF_BYCOMMAND);
    nRadioButtonId = IDM_RadioButton1 + 102;
    for(i = 0; i < 3; i++)
     CheckMenuItem(hMenu2, IDM_CheckButton1 + i, MF_UNCHECKED);
    break;
   case WM_COMMAND:
    switch(LOWORD(wParam))
     {
      case IDM_EXIT:
       SendMessage(hWnd, WM_CLOSE, 0,0);
       break;
      case IDM_RadioButton1:
      case IDM_RadioButton2:
      case IDM_RadioButton3:
       nRadioButtonId = LOWORD(wParam) + 102;
```

```
        CheckMenuRadioItem(hMenu1, IDM_RadioButton1, IDM_RadioButton3,
                           LOWORD(wParam), MF_BYCOMMAND);
        break;
      case IDM_CheckButton1:
      case IDM_CheckButton2:
      case IDM_CheckButton3:
        i = LOWORD(wParam) - 105;
        uCheckBoxesState[i] = uCheckBoxesState[i] == MF_CHECKED ?
                                                    MF_UNCHECKED :
                                                    MF_CHECKED;
        CheckMenuItem(hMenu2, LOWORD(wParam), MF_BYCOMMAND |
                      uCheckBoxesState[i]);
        break;
      case IDM_DisplayDialog:
        DialogBox(hInst, "ButtonsExample", hWnd, ButtonsExampleDialogProc);
        break;
     }
    break;
   case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
  }
 return DefWindowProc(hWnd,Message,wParam, lParam);
}

BOOL CALLBACK ButtonsExampleDialogProc(HWND hDlg, UINT Message,
                                       WPARAM wParam,
                                       LPARAM lParam)
{
 int i;
 char cMyMessage[80];

 switch(Message)
  {
   case WM_INITDIALOG:
// Set states of controls
    SendDlgItemMessage(hDlg, nRadioButtonId, BM_SETCHECK,
                       BST_CHECKED, 0);
    for(i = IDC_CHECKBOX1; i <= IDC_CHECKBOX3; i++)
     if(uCheckBoxesState[i - 208])
       SendDlgItemMessage(hDlg, i, BM_SETCHECK, BST_CHECKED, 0);
    return TRUE;
   case WM_COMMAND:
    switch(LOWORD(wParam))
     {
      case IDC_RADIOBUTTON1:
      case IDC_RADIOBUTTON2:
      case IDC_RADIOBUTTON3:
       sprintf(cMyMessage,"Message from RadioButton%d",
               LOWORD(wParam) - 203);
```

```
            SendDlgItemMessage(hDlg,IDC_STATUSBAR,SB_SETTEXT,
                               (WPARAM) 0, (LPARAM) cMyMessage);
            CheckMenuRadioItem(GetSubMenu(GetSubMenu(GetMenu(hWnd),1), 0),
                               IDM_RadioButton1, IDM_RadioButton3,
                               LOWORD(wParam)- 102, MF_BYCOMMAND);
            return FALSE;
          case IDC_CHECKBOX1:
          case IDC_CHECKBOX2:
          case IDC_CHECKBOX3:
            sprintf(cMyMessage,"Message from CheckBox%d",
                  LOWORD(wParam) - 207);
            SendDlgItemMessage(hDlg,IDC_STATUSBAR,SB_SETTEXT,
                               (WPARAM) 0, (LPARAM) cMyMessage);
            i = LOWORD(wParam) - 208;
            uCheckBoxesState[i] = uCheckBoxesState[i] == MF_CHECKED ?
                                                  MF_UNCHECKED :
                                                  MF_CHECKED;
            CheckMenuItem(GetSubMenu(GetSubMenu(GetMenu(hWnd), 1), 1),
                          LOWORD(wParam) - 103, uCheckBoxesState[i]);
            return FALSE;
          case IDC_BUTTON1:
            SendDlgItemMessage(hDlg,IDC_STATUSBAR,SB_SETTEXT,
                               (WPARAM) 0,
                               (LPARAM) "Message from PushButton");
            return TRUE;
          case IDC_BUTTON2:
// Save the state of RadioButtons
            i = IDC_RADIOBUTTON1;
            while(!SendDlgItemMessage(hDlg, i, BM_GETCHECK, 0, 0))
              i++;
            nRadioButtonId = i;
// Save the state of CheckButtons
            for(i = IDC_CHECKBOX1; i <= IDC_CHECKBOX3; i++)
              uCheckBoxesState[i - 208] = SendDlgItemMessage(hDlg, i,
                                          BM_GETCHECK, 0, 0) == 0 ?
                                          MF_UNCHECKED : MF_CHECKED;
            EndDialog(hDlg,0);
            return TRUE;
        }
      break;
    }
  return FALSE;
 }
```

При линковании программы необходимо использовать файл ресурсов:

```
#include "buttons.h"

ButtonsExample DIALOG 50, 50, 154, 108
```

```
STYLE DS_MODALFRAME | DS_3DLOOK | DS_CONTEXTHELP |
        WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Buttons Example"
FONT 8, "MS Sans Serif"
{
 CONTROL "PushButton", IDC_BUTTON1, "button", BS_PUSHBUTTON |
              BS_CENTER | BS_NOTIFY | WS_CHILD | WS_VISIBLE |
              WS_TABSTOP, 8, 72, 64, 16
 CONTROL "RadioButton1", IDC_RADIOBUTTON1, "button",
              BS_AUTORADIOBUTTON | WS_CHILD | WS_VISIBLE |
              WS_TABSTOP, 8, 12, 64, 16
 CONTROL "RadioButton2", IDC_RADIOBUTTON2, "button",
              BS_AUTORADIOBUTTON | BS_FLAT | WS_CHILD |
              WS_VISIBLE | WS_TABSTOP, 8, 28, 64, 16
 CONTROL "RadioButton3", IDC_RADIOBUTTON3, "button",
              BS_AUTORADIOBUTTON | BS_LEFTTEXT | WS_CHILD |
              WS_VISIBLE | WS_TABSTOP, 8, 44, 64, 16
 CONTROL "Group1", IDC_GROUPBOX1, "button", BS_GROUPBOX |
              WS_CHILD | WS_VISIBLE | WS_GROUP, 4, 4, 72, 60
 CONTROL "CheckBox1", IDC_CHECKBOX1, "button", BS_AUTOCHECKBOX |
              WS_CHILD | WS_VISIBLE | WS_TABSTOP, 82, 12, 64, 16
 CONTROL "CheckBox2", IDC_CHECKBOX2, "button", BS_AUTOCHECKBOX |
              WS_CHILD | WS_VISIBLE | WS_TABSTOP, 82, 28, 64, 16
 CONTROL "CheckBox3", IDC_CHECKBOX3, "button", BS_AUTOCHECKBOX |
              WS_CHILD | WS_VISIBLE | WS_TABSTOP, 82, 43, 64, 16
 CONTROL "Group2", IDC_GROUPBOX2, "button", BS_GROUPBOX |
              WS_CHILD | WS_VISIBLE | WS_GROUP, 78, 4, 72, 60
 CONTROL "Cancel", IDC_BUTTON2, "button", BS_PUSHBUTTON |
              BS_CENTER | WS_CHILD | WS_VISIBLE | WS_TABSTOP, 82, 72,
              64, 16
 CONTROL "StatusWindow1", IDC_STATUSBAR, "msctls_statusbar32", 3 |
              WS_CHILD | WS_VISIBLE, 0, 116, 154, 12
}
ButtonsExampleMenu MENU
{
 POPUP "&File"
 {
  MENUITEM "E&xit", IDM_EXIT
 }

 POPUP "&Dialog"
 {
  POPUP "Initialize &RadioButtons"
  {
   MENUITEM "Set RadioButton&1", IDM_RadioButton1
   MENUITEM "Set RadioButton&2", IDM_RadioButton2
   MENUITEM "Set RadioButton&3", IDM_RadioButton3
  }

  POPUP "Initialize &CheckButtons"
```

```
{
MENUITEM "Set CheckButton&1", IDM_CheckButton1
MENUITEM "Set CheckButton&2", IDM_CheckButton2
MENUITEM "Set CheckButton&3", IDM_CheckButton3
}

MENUITEM SEPARATOR
MENUITEM "Displa&y Dialog", IDM_DisplayDialog
}
}
```

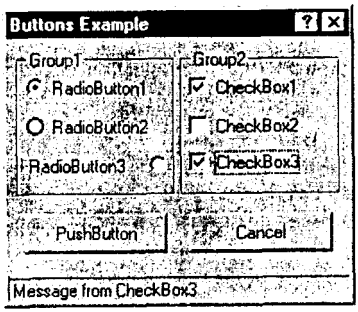


Рис. 8. Диалоговое окно с кнопками различных стилей

На рис. 8 показано диалоговое окна, которое создается данной программой. Функция WinMain() полностью стандартна и ничего нового не содержат.

При обработке сообщения WM_CREATE мы узнаем о новой возможности, связанной с меню. С помощью функции CheckRadioMenuItem() можно заставить группу элементов меню работать как кластер RadioButtons. В этом случае установка отметки у одного элемента меню приводит к сбросу отметки у всех других элементов меню, входящих в состав группы. Характерно, что при определении группы мы должны указать минимальный и максимальный идентификаторы элементов меню, включаемых в группу. Элементы, включаемые в меню, должны иметь идентификаторы, попадающие в определенный интервал, а не произвольно определенные. Этим мы будем пользоваться при определении и отображении в меню состояния кластера RadioButtons.

При обработке того же сообщения мы встречаем еще одну функцию, позволяющую установить отметку у элемента меню, - CheckMenuItem(). Эта функция позволяет изменять состояние только одного элемента меню. С помощью этой функции мы будем устанавливать и отображать состояние CheckButtons.

Наверное, читателю небезынтересно узнать, что эти две функции для отметки элементов по умолчанию используют различные значки. Рекомендую читателю обратить внимание на то, какие значки используются для отметки в обеих функциях.

Но эта программа написана отнюдь не для того, чтобы рассказывать о новых возможностях меню. При обработке сообщения от элемента меню с надписью «Display Dialog» создается диалоговое окно, в котором и содержатся те кнопки, о которых мы говорили. В зависимости от того, какое диалоговое окно должно быть создано, могут быть использованы функции DialogBox() и CreateDialog(). Функция DialogBox() создает модальное диалоговое окно. Немодальное диалоговое окно создается с помощью функции CreateDialog(). Мы используем функцию DialogBox().

В файле winuser.h эта функция описана следующим образом:

```
WINUSERAPI int WINAPI DialogBoxParamA(HINSTANCE hInstance,
                                      LPCSTR lpTemplateName,
                                      HWND hWndParent,
                                      DLGPROC lpDialogFunc,
                                      LPARAM dwInitParam);
WINUSERAPI int WINAPI DialogBoxParamW(HINSTANCE hInstance,
                                      LPCWSTR lpTemplateName,
                                      HWND hWndParent,
                                      DLGPROC lpDialogFunc,
                                      LPARAM dwInitParam);
#ifdef UNICODE
#define DialogBoxParam  DialogBoxParamW
#else
#define DialogBoxParam  DialogBoxParamA
#endif // !UNICODE
#define DialogBoxA(hInstance, lpTemplate, hWndParent, lpDialogFunc) \
DialogBoxParamA(hInstance, lpTemplate, hWndParent, lpDialogFunc, 0L)
#define DialogBoxW(hInstance, lpTemplate, hWndParent, lpDialogFunc) \
DialogBoxParamW(hInstance, lpTemplate, hWndParent, lpDialogFunc, 0L)
#ifdef UNICODE
#define DialogBox  DialogBoxW
#else
#define DialogBox  DialogBoxA
#endif // !UNICODE
```

Видно, что функция DialogBox() фактически является частным случаем функции DialogBoxParam(). На их различии мы остановимся чуть позже, а сейчас рассмотрим аргументы DialogBox().

Первый аргумент понятен, мы его используем в каждой программе. Второй аргумент - указатель на имя шаблона, использующегося при

построении диалога. В нашем случае диалог сохранен в виде ресурса, поэтому мы указываем имя ресурса.

Третий аргумент - хэндл родительского окна. О том, что такое родительское окно, нужно сказать особо.

Во-первых, если мы представим окна, начиная с Desktop'а, как располагающиеся друг над другом (в так называемом Z-порядке), то дочернее окно обязательно будет над родительским, от этого зависит порядок обработки сообщений. Во-вторых, сообщения о действиях с диалоговым окном (нотификационые сообщения) система будет посылать именно родительскому окну. В-третьих, при закрытии родительского окна дочернее окно закрывается автоматически. Возможно, конечно, создание диалоговых окон без родителя, но тогда придется писать огромное количество кода, обеспечивающее нормальное функционирование окна.

Обычно родительским окном является то, оконная функция которого создает диалоговое окно. В программе основное окно программы является родительским по отношению к диалоговому окну.

И наконец, последний аргумент - указатель на функцию диалогового окна, т. е. на функцию, которая будет обрабатывать получаемые от элементов управления сообщения.

В дополнение к этому заметим, что последний аргумент функции DialogBoxParam() - это какой-то параметр, определяемый программистом, который может быть передан функции диалогового окна.

Итак, мы рассмотрели функцию основного окна и остановились на функции окна диалога. На очереди - ее рассмотрение.

Диалоговая функция очень напоминает функцию окна, но имеет ряд отличий. Во-первых, обычная оконная функция возвращает значение типа LRESULT. Диалоговая функция возвращает значение типа BOOL. Во-вторых, обычная функция окна передает сообщения, обрабатывать которые не нужно, процедуре обработки по умолчанию (DefWindowProc()). Диалоговая функция в том случае, если она обработала сообщение, возвращает TRUE, а в противном случае FALSE. Другими словами, диалоговая функция должна вернуть FALSE в том случае, если ей необходимо передать сообщение для дальнейшей обработки в «недрах» Windows.

Для функции диалогового окна аналогом сообщения WM_CREATE является сообщение WM_INITDIALOG. Диалоговая функция получает его после создания диалогового окна в памяти, но до его отображения. Обычно именно при обработке этого сообщения производится инициализация диалога. Программа, которую мы сейчас разбираем, не является исключением. При обработке WM_INITDIALOG мы встречаемся с одним исключением из стройной системы правил Win32. Если программе необ-

ходимо, чтобы система продолжила обработку сообщения WM_INITDIALOG, то после обработки этого сообщения программа должна вернуть TRUE. Рекомендую читателю немного поэкспериментировать с программой и попробовать после обработки WM_INITDIALOG вернуть вместо TRUE значение FALSE. Думаю, разница будет заметна сразу (по моему мнению, она бросается в глаза).

При обработке сообщения WM_DIALOG в программе производится установка состояний RadioButtons и CheckButtons. Здесь мы встречаемся еще с одной интересной функцией. Давайте вспомним, что любой элемент управления является окном. Для управления окнами используются сообщения, а для того, чтобы послать сообщение окну, нужно знать хэндл окна - адресата. При создании диалога мы определяли идентификатор элемента управления. Win32 позволяет определить хэндл окна - элемента управления. Для этого предназначена функция GetDlgItem(), возвращающая искомый хэндл. Теперь мы можем послать сообщение окну с помощью функции SendMessage(). Таким образом, нам нужно написать что-то типа SendMessage(GetDlgItem(...));

Для того чтобы облегчить жизнь программистам, в Win32 включена функция SendDlgItemMessage(), объединяющая две упомянутые выше функции в одну. Прототип этой функции можно найти в файле winuser.h:

```
WINUSERAPI LONG WINAPI SendDlgItemMessageA(HWND hDlg,
                                           int nIDDlgItem,
                                           UINT Msg,
                                           WPARAM wParam,
                                           LPARAM lParam);
WINUSERAPI LONG WINAPI SendDlgItemMessageW(HWND hDlg,
                                           int nIDDlgItem,
                                           UINT Msg,
                                           WPARAM wParam,
                                           LPARAM lParam);
#ifdef UNICODE
#define SendDlgItemMessage  SendDlgItemMessageW
#else
#define SendDlgItemMessage  SendDlgItemMessageA
#endif // !UNICODE
```

Первый аргумент - хэндл диалогового окна, функция диалога получает его от системы при вызове. Второй аргумент - идентификатор элемента управления. Третий аргумент - посылаемое элементу сообщение. Для управления кнопками служат сообщения, идентификаторы которых начинаются с BM_. Все определенные в winuser.h идентификаторы приведены в табл 21.

Таблица 21. Сообщения, посылаемые кнопкам

Идентификатор	Значение	Описание
BM_GETCHECK	0x00F0	Получить состояние отметки CheckBox'a или RadioButton'a
BM_SETCHECK	0x00F1	Установить отметку в CheckBox'a или RadioButton'a
BM_GETSTATE	0x00F2	Получить состояние кнопки
BM_SETSTATE	0x00F3	Установить состояние подсветки кнопки (имитация удержания нажатой клавиши мыши на кнопке)
BM_SETSTYLE	0x00F4	Изменить стиль кнопки
BM_CLICK	0x00F5	Симуляция нажатия кнопки мыши
BM_GETIMAGE	0x00F6	Получить хэндл изображения (иконки или bitmap'a), связанного с кнопкой
BM_SETIMAGE	0x00F7	Связать изображение с кнопкой

Таблица 22. Состояния кнопок и их описание

Идентификатор	Значение	Описание
BST_UNCHECKED	0x0000	CheckBox или RadioButton делается неотмеченной (может устанавливаться)
BST_CHECKED	0x0001	CheckBox или RadioButton делается отмеченной (может устанавливаться)
BST_INDETERMINATE	0x0002	Состояние CheckBox не определено (может устанавливаться)
BST_PUSHED	0x0004	Кнопка нажата (только в ответ на запрос о состоянии кнопки)
BST_BST_FOCUS	0x0008	Кнопка имеет клавиатурный фокус (только в ответ на запрос о состоянии кнопки)

Четвертый и пятый аргументы - это wParam и lParam посылаемого сообщения. Для каждого сообщения они определяются отдельно. Надеюсь, что читатель разобереться с параметрами этих сообщений самостоятельно по файлам помощи. Состояния кнопок имеют идентификаторы, начинающиеся с BST_ (табл. 22).

После того, как мы разобрали функцию SendDlgItemMessage(), все остальное в программе вызывать каких-либо трудностей не должно.

Окно сообщений

Но возникает вопрос: неужели же даже для простейших действий, например, для запроса подтверждения о необходимости удаления файла, необходимо писать функцию диалогового окна? Ответ очевиден: нет, не нужно. Для решения этих вопросов вполне достаточно так называемого окна сообщений.

Окно сообщений, которое мы неоднократно использовали, является простейшим типом диалогового окна. Его назначение определяется его названием. Ни для чего другого, кроме вывода сообщения пользователю и предложения нажать одну из имеющихся кнопок, эти окна не предназначены. Тем не менее, работу с этим типом диалогов я хотел бы рассмотреть очень подробно, так как она очень часто используется не только для выдачи сообщений пользователю, но и для отладки. Функция, с помощью которой создается окно сообщений, называется MessageBox().

В файле winuser.h эта функция описана следующим образом:

```
WINUSERAPI int WINAPI MessageBoxA(HWND hWnd, LPCSTR lpText,
                                  LPCSTR lpCaption, UINT uType);
WINUSERAPI int WINAPI MessageBoxW(HWND hWnd , LPCWSTR lpText,
                                  LPCWSTR lpCaption, UINT uType);
#ifdef UNICODE
#define MessageBox  MessageBoxW
#else
#define MessageBox  MessageBoxA
#endif // !UNICODE
```

Первый аргумент этой функции - hWnd - хэндл родительского окна, т. е. того окна, которому будут посылаться сообщения от окна сообщений (извините меня, уважаемый читатель, за тавтологию. В данном случае я прошу не путать окно сообщений ДЛЯ ПОЛЬЗОВАТЕЛЯ с сообщениями ДЛЯ ОКОН). Второй аргумент - lpText - указатель на строку, содержащую отображаемый внутри окна текст. Перед отображением этот текст может быть отформатирован с помощью функции sprintf(). Третий аргумент - lpCaption -заголовок окна сообщений. (Мне, например, использующему окна сообщений в основном для вывода отладочных сообщений, нравится давать окну заголовки типа «Hurrau!» или «At last...».)

Четвертый аргумент - uType - определяет тип окна сообщений, т. е.:

перечень кнопок, отображаемых в окне сообщений;

иконку, отображаемую в окне сообщений;

кнопку, считающуюся кнопкой по умолчанию;

модальность окна сообщений.

Наверное, вы неоднократно видели на экране окно сообщений с различным набором кнопок. Этот набор состоит из кнопок «OK», «Retry», «Abort» и др. Наличие каждой из таких кнопок определяется флагами, установленными в четвертом аргументе. Возможные значения флагов uType можно найти в файле winuser.h. Все они начинаются с букв MB (табл. 23).

Таблица 23. Типы окон сообщений

Флаг	Значение	Эффект
MB_OK	0x00000000L	Кнопка «OK»
MB_OKCANCEL	0x00000001L	Кнопки «OK» и «Cancel»
MB_ABORTRETRYIGNORE	0x00000002L	Кнопки «Abort», «Retry», «Ignore»
MB_YESNOCANCEL	0x00000003L	Кнопки «Yes», «No», «Cancel»
MB_YESNO	0x00000004L	Кнопки «Yes», «No»
MB_RETRYCANCEL	0x00000005L	Кнопки «Retry», «Cancel»

Таблица 24. Идентификаторы иконки, появляющиеся в окне сообщений

Флаг	Значение	Эффект
MB_ICONHAND	0x00000010L	Иконка с изображением знака «Stop»
MB_ICONQUESTION	0x00000020L	Иконка с изображением вопросительного знака
MB_ICONEXCLAMATION	0x00000030L	Иконка с изображением восклицательного знака
MB_ICONASTERISK	0x00000040L	Иконка с изображением буквы i
MB_ICONINFORMATION	0x00000040L	Иконка с изображением буквы i
MB_ICONSTOP	0x00000010L	Иконка с изображением знака «Stop»

Таблица 25. Идентификаторы кнопки по умолчанию

Флаг	Значение	Эффект
MB_DEFBUTTON1	0x00000000L	Первая кнопка работает по умолчанию
MB_DEFBUTTON2	0x00000100L	Вторая кнопка работает по умолчанию
MB_DEFBUTTON3	0x00000200L	Третья кнопка работает по умолчанию

Таблица 26. Идентификаторы, определяющие модальность окна сообщений

Флаг	Значение	Эффект
MB_APPLMODAL	0x00000000L	Разрешаются переключения на другие приложения
MB_SYSTEMMODAL	0x00001000	Не разрешаются переключения на другие приложения
MB_TASKMODAL	0x00002000	Применяется в случае отсутствия родительского окна для запрещения ввода в другие окна

Таблица 27. Значения, возвращаемые функцией MessageBox()

Нажатая клавиша	Числовое значение	Возвращаемое функцией значение
OK	1	IDOK
Cancel	2	IDCANCEL
Abort	3	IDABORT
Retry	4	IDRETRY
Ignore	5	IDIGNORE
Yes	6	IDYES
No	7	IDNO
	8	IDCLOSE
	9	IDHELP

Кроме этого набора кнопок, uType определяет также и иконку (одну из предопределенных в Win32), которая будет отображаться в окне сообщений. Таблица 24 содержит флаги, определяющие иконку, появляющуюся в окне сообщений.

Следующие флаги определяют, какая из кнопок будет считаться кнопкой по умолчанию (табл. 25). Модальность окна сообщений определяют флаги, приведенные в табл. 26.

Ну, вот, кажется и все. Мне бы хотелось обратить внимание читателя на то, что комбинировать с помощью логических операций можно только величины из разных таблиц. Это заметно даже при просмотре численных значений. Что произойдет при комбинировании этом, известно только фирме Microsoft. Например, указав в порядке эксперимента одновременно флаги MB_RETRYCANCEL и MB_ABORTRETRYIGNORE, я вообще не получил никакой кнопки в окне сообщений. Пришлось завершать процесс аварийно.

Итак, выдавать сообщения мы научились. Не хватает самой малости, выяснить, как приложение узнает о том, какую кнопку нажал пользователь. Но здесь дело обстоит просто. Возвращаемое функцией MessageBox() значение напрямую определяется тем, какую кнопку нажал пользователь (табл. 27).

К сожалению, два последних значения, которые я нашел в заголовочном файле winuser.h, в файлах помощи фирмы Microsoft не описаны. Об их назначении можно только догадываться по их названиям.

Мы закончим рассмотрение темы об окнах сообщений очередной демонстрационной программой. Для того чтобы увидеть, что возвращает функция MessageBox(), я воспользовался обычным окном сообщений из интегрированной среды Borland C++ 5.0. В демонстрационной программе я не стал мудрствовать лукаво и выдал на отображение одно-

единственное окно сообщений с одной иконкой и тремя кнопками, «Abort», «Retry» и «Ignore». При нажатии клавиш «Retry» и «Ignore» в окне Message появляются сообщения о том, какая клавиша нажата. При нажатии клавиши «Abort» работа программы прекращается:

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance, LPSTR
lpszCmdLine, int nCmdShow)
  {
   int nResult = IDRETRY;
   int nIndex;
   char* pcMessage[]
    {
     "Retry key pressed",
     "Ignore key pressed"
    };

   while( (nResult = MessageBox(NULL, "And what do you want to see?",
          "See in Message window, please", MB_ABORTRETRYIGNORE |
          MB_ICONASTERISK)) != IDABORT)
   {
    switch(nResult)
     {
      case IDRETRY:
       nIndex = 0;
       break;
      case IDIGNORE:
       nIndex = 1;
       break;
     }
    OutputDebugString(pcMessage[nIndex]);
   }
  return 1;
 }
```

На рис. 9 показано окно сообщений, создаваемое программой.

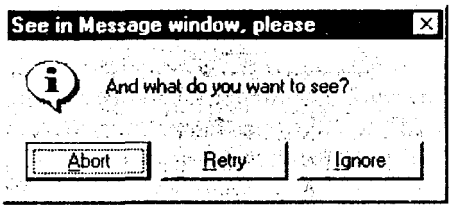


Рис. 9. Окно сообщений с тремя кнопками

Единственное, что осталось неясным в этой программе - это функция OutputDebugString(). Ее единственным аргументом является указатель на строку, которая должна быть передана отладчику. В случае отсутствия отладчика, содержимое строки появляется в окне сообщений интегрированной среды.

Мы разобрали вопросы о создании окна сообщений и о работе кнопок. Теперь я расскажу о таком мощном элементе, как окно списка. Наверное, это наиболее часто используемый элемент управления (не считая, конечно, кнопок), а возможности, которые список предоставляет программисту, иногда просто поражают. И это без учета комбинированных списков, всевозможных производных типа окон просмотра деревьев и т.д.

СПИСКИ

Окно списка - это совокупность строк и/или картинок, которые отображаются в скроллируемом окне. Список позволяет добавить элемент в список, удалить элемент из списка, найти элемент, получить общее число элементов и т. д. Образно говоря, список - это некое уменьшенное подобие базы данных, позволяющее выполнять большинство стандартных функций по управлению базой данных.

Все списки делятся на две большие группы. В первую группу входят списки, которые позволяют выбрать только один элемент из всех имеющихся. Вторую группу составляют списки, позволяющие выбрать одновременно несколько элементов, удовлетворяющих определенному критерию.

У списков есть еще одно очень важное свойство. С каждым элементом списка мы можем связать некоторый объект в памяти. Другими словами, то, что мы видим в списке на экране, может быть только вершиной айсберга. Сам айсберг, сиречь информация, связанная с элементом, может храниться глубоко в недрах Win32. Список не может быть очень большим (список, как и любая динамическая структура данных, хранится в памяти), но он может быть весьма удобным инструментом для создания и хранения небольших объемов данных. Кстати, для хранения данных можно создать окно списка, но на отображение его не выводить. В памяти будет создана динамическая структура, с которой удобно работать.

Список может быть создан вместе с диалоговым окном в качестве ресурса, а также посредством использования функции CreateWindow(). В последнем случае в качестве имени класса необходимо указывать «LISTBOX». В подавляющем большинстве случаев списки создаются в ресурсах, поэтому мы остановимся именно на этом способе. При работе в редакторе ресурсов ничего сложного в создании списка нет. Некоторые

сложности возникают при создании ресурса в текстовом редакторе, но все эти сложности преодолимы. Формат описания окна списка в файле ресурсов ничем не отличается от описания, которое мы использовали для описания кнопок. Дабы читателю не пришлось разыскивать это описание, я приведу его еще раз:

CONTROL «Заголовок», ListboxID, «listbox», styles, X, Y, Width, Height

Здесь я заменил класс «button» на класс «listbox», ибо именно к этому классу относятся списки. Возможные стили списков мы получим, обратившись к файлу winuser.h (табл. 28).

Все сообщения, с которыми работают списки, можно разделить на несколько логических групп. Постараюсь описать три группы, в которые я включил сообщения, представляющиеся мне наиболее важными.

Сообщения, обеспечивающие добавление и удаление элемента

Для того чтобы добавить элемент в список, необходимо просто послать списку сообщение LB_ADDSTRING. При этом wParam должен быть равным нулю, а lParam должен указывать на добавляемый к списку объект. Этот элемент совсем необязательно должен быть строкой. Если у списка не установлен стиль LBS_HASSTRING, то lParam, указывает на объект, связанный с элементом. Для того чтобы получить или изменить эти данные, можно воспользоваться сообщениями LB_GETITEMDATA и LB_SETITEMDATA.

Если у списка установлены стили LBS_SORT и LBS_HASSTRING, то строка добавляется в список, после чего происходит сортировка. Если стиль LBS_SORT не указан, строка добавляется в конец списка. Если, наоборот, указан стиль LBS_SORT, но не указан LBS_HASSTRING, то список посылает родительскому окну одно или несколько сообщений WM_COMPAREITEM, которые позволяют определить, где должен быть расположен включаемый элемент. Возвращает это сообщение либо номер, под которым элемент включен в список, либо одно из двух значений, говорящих об ошибке: LB_ERR - встретилась ошибка; LB_ERRSPACE - не хватило памяти для размещения элемента.

Элемент может быть добавлен в список и другим способом. Отличие сообщения LB_INSERTSTRING от предыдущего состоит в том, что wParam этого сообщения содержит номер (считается от нуля) элемента, ПОСЛЕ которого нужно включить данный элемент. Кроме этого, сортировка элементов в этом случае не производится. Возвращаемые значения точно такие же, как и в предыдущем случае.

Особо нужно рассмотреть случай, когда необходимо создать список файлов в текущей директории. Для того чтобы облегчить жизнь программисту, в систему было включено сообщение LB_DIR. В качестве wParam этого сообщения записываются атрибуты файлов, имена которых необходимо добавить в список. Возможные значения этого параметра приведены в табл. 29.

Таблица 28. Стили окон списков

Флаг	Значение	Описание
LBS_NOTIFY	0x0001L	Посылает сообщение родительскому окну о щелчке или двойном щелчке клавишей мыши
LBS_SORT	0x0002L	Строки сортируются по алфавиту
LBS_NOREDRAW	0x0004L	Внешний вид списка не изменяется даже тогда, когда производятся изменения
LBS_MULTIPLESEL	0x0008L	Список позволяет множественный выбор
LBS_OWNERDRAWFIXED	0x0010L	Родительское окно ответственно за прорисовку элементов, все элементы списка одинаковой высоты
LBS_OWNERDRAWVARIABLE	0x0020L	То же, что и предыдущее, но элементы списка могут быть разной высоты
LBS_HASSTRING	0x0040L	Элементы списка - строки
LBS_USETABSTOPS	0x0080L	Разрешает расширять символы табуляции, встречающиеся в строках
LBS_NOINTEGRALHEIGHT	0x0100L	Список создается точно такого же размера, который указан в программе, выравнивание не производится
LBS_MULTICOLUMN	0x0200L	В списке создается несколько колонок, он скроллируется по горизонтали
LBS_WANTKEYBOARDINPUT	0x0400L	Позволяет приложению обрабатывать ввод с клавиатуры тогда, когда список удерживает фокус ввода
LBS_EXTENDEDSEL	0x0800L	Позволяет списку с множественным выбором использовать для выделения клавишу Shift совместно с мышью или другие клавиатурные комбинации
LBS_DISABLENOSCROLL	0x1000L	Показывать запрещенную линейку прокрутки тогда, когда в списке недостаточно элементов для прокрутки
LBS_NODATA	0x2000L	Устаревший стиль
LBS_NOSEL	0x4000L	Элементы списка видны, но выделение запрещено
LBS_STANDARD		LBS_NOTIFY | LBS_SORT | WS_VSCROLL | WS_BORDER

Таблица 29. Атрибуты файлов, добавляемых в окно списка

Параметр	Значение	Описание
DDL_READWRITE	0x0000	Включить только файлы, доступные для чтения и записи, без дополнительных атрибутов
DDL_READONLY	0x0001	Включить в список только файлы, доступные для чтения
DDL_HIDDEN	0x0002	Включить в список скрытые файлы
DDL_SYSTEM	0x0004	Включить в список системные файлы
DDL_DIRECTORY	0x0010	Включить в список поддиректории
DDL_ARCHIVE	0x0020	Включить в список архивные файлы
DDL_POSTMSG	0x2000	
DDL_DRIVES	0x4000	Включить в список имена дисководов
DDL_EXCLUSIVE	0x8000	Включать в список файлы только с указанными атрибутами

lParam сообщения LB_DIR указывает на строку, которая определяет, какие файлы необходимо добавить в список. Строка формируется по правилам, принятым ещё в MS DOS, то есть, к примеру, для того, чтобы отобразить все файлы в директории MyDir на диске C: необходимо записать «c:\\MyDir*.*»

Удаление элемента из списка производится посредством посылки списку сообщения LB_DELETESTRING. В wParam этого сообщения необходимо указать номер удаляемого элемента. При анализе возвращаемого значения необходимо учесть, что при нормальном удалении возвращается число оставшихся элементов списка. Значение LB_ERR должно указать программисту на то, что он неверно указал номер удаляемого элемента.

Вторая большая группа сообщений - это

Сообщения, обеспечивающие навигацию в списке

Под навигацией в списке я понимаю возможность программы определить, где находится указатель (выделенный элемент) списка и/или установить указатель на тот элемент, который в данный момент требуется программе. Возможности здесь достаточно обширные.

Наверное, наиболее часто для определения места выделенного элемента в списке будет использоваться сообщение LB_GETCURSEL. Никаких параметров это сообщение не использует, и wParam, и lParam должны быть равны 0. Если возвращаемое значение равно LB_ERR, в списке нет выделенных элементов.

Сделать элемент выделенным позволяет сообщение LB_SETCURSEL, wParam которого должен содержать номер элемента, который должен

стать текущим. Это сообщение имеет дело только со списками, позволяющими одиночный выбор.

Узнать, какая строка или какие данные хранятся в элементе списка, можно с помощью сообщения LB_GETTEXT. wParam должно хранить индекс интересующего нас элемента, а lParam должно указывать на буфер, в который будут записаны строка или указатель на ассоциированные данные.

Число элементов в списке может быть определено посредством сообщения LB_GETCOUNT. Параметры этого сообщения не используются и должны быть установлены в 0, а возвращает оно число элементов в списке. Одна тонкость - число элементов всегда на 1 больше индекса последнего элемента списка. Например, в списке один элемент. Его номер будет равным нулю, но LB_GETCOUNT вернет 1.

И последней группой сообщений, на которых мы остановимся, являются

Нотификационные сообщения

Если у списка установлен стиль LBS_NOTIFY, то список будет оповещать родительское окно о том, какие события с ним произошли посредством нотификационных сообщений. Нотификационные сообщения в случае списка - это сообщения WM_COMMAND, у которых младшее слово wParam содержит идентификатор окна списка, старшее слово wParam - код нотификации, а lParam - хэндл окна списка.

Кодов нотификации всего шесть (табл. 30).

Таблица 30. Коды нотификационных сообщений, посылаемых окнами списков

Код нотификации	Описание
LBN_ERRSPACE	Не хватает памяти
LBN_SELCHANGE	Выделенным стал другой элемент
LBN_DBLCLK	Пользователь сделал двойной щелчок клавишей мыши
LBN_SELCANCEL	Пользователь снял выделение
LBN_SETFOCUS	Список получил клавиатурный фокус
LBN_KILLFOCUS	Список потерял клавиатурный фокус

Для того чтобы проиллюстрировать все то, о чем говорилось выше, приведем небольшую программу. Эта программа демонстрирует добавление строк в список и выборку информации о строках. К сожалению, мы еще не изучили окно редактирования, поэтому я не мог написать программу, осуществляющую ввод информации с экрана, но, надеюсь, мы сделаем это в последующих разделах.

Для нормальной компиляции программы требуется файл ресурсов:

```
#include "list.h"

ListBox DIALOG 50, 50, 150, 140
STYLE DS_MODALFRAME | DS_3DLOOK | DS_CONTEXTHELP |
        WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "ListBox Example"
FONT 8, "MS Sans Serif"
{
 PUSHBUTTON "Display", ID_OK, 15, 100, 50, 14
 PUSHBUTTON "Cancel", ID_Cancel, 85, 100, 50, 14
 CONTROL "Families", ID_MyListBox, "listbox", LBS_STANDARD |
            WS_CHILD | WS_VISIBLE | WS_TABSTOP, 15, 16, 120, 65
 CONTROL "Families", -1, "static", SS_LEFT | WS_CHILD | WS_VISIBLE, 16, 6,
            120, 10
 CONTROL "StatusBar", ID_StatusBar, "msctls_statusbar32", 3 | WS_CHILD |
            WS_VISIBLE, 0, 129, 150, 12
}

ListBoxMenu MENU
 {
  POPUP "&File"
   {
    MENUITEM "E&xit", IDM_Exit
   }
 MENUITEM "&Display Dialog", IDM_DisplayDialog
}
```

Далее следует файл заголовков, также используемый в программе:

```
#define IDM_Exit            101
#define IDM_Cancel          102
#define IDM_DisplayDialog   103
#define ID_OK               201
#define ID_Cancel           202
#define ID_MyListBox        203
#define ID_StatusBar        204
```

И, наконец, основной файл программы:

```
#include <windows.h>
#include "list.h"
#include <commctrl.h>

HINSTANCE hInst;

LRESULT CALLBACK ListBoxExampleWndProc(HWND, UINT, UINT, LONG);
```

```
BOOL CALLBACK ListBoxExampleDialogProc(HWND, UINT, WPARAM,
                                       LPARAM);

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow )
 {
  HWND hWnd ;
  WNDCLASS WndClass ;
  MSG Msg;
  char szClassName[] = "ListExample";

  hInst = hInstance;
  InitCommonControls();
/* Registering our window class */
/* Fill WNDCLASS structure */
  WndClass.style = CS_HREDRAW | CS_VREDRAW;
  WndClass.lpfnWndProc = ListBoxExampleWndProc;
  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 = "ListBoxMenu";
  WndClass.lpszClassName = szClassName;

  if ( !RegisterClass(&WndClass) )
    {
     MessageBox(NULL,"Cannot register class","Error",MB_OK);
     return 0;
    }
  hWnd = CreateWindow(szClassName, "ListBox Example Program",
                      WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                      CW_USEDEFAULT, CW_USEDEFAULT,
                      CW_USEDEFAULT, NULL, NULL,
                      hInstance,NULL);
  if(!hWnd)
    {
     MessageBox(NULL,"Cannot create window","Error",MB_OK);
     return 0;
    }

  /* Show our window */
  ShowWindow(hWnd,nCmdShow);
  UpdateWindow(hWnd);

/* Beginning of messages cycle */
  while(GetMessage(&Msg, NULL, 0, 0))
   {
    TranslateMessage(&Msg);
```

```
    DispatchMessage(&Msg);
   }
  return Msg.wParam;
 }

LRESULT CALLBACK ListBoxExampleWndProc(HWND hWnd, UINT Message,
                                       UINT wParam, LONG lParam )
 {
  switch(Message)
   {
    case WM_COMMAND:
     switch(LOWORD(wParam))
      {
       case IDM_DisplayDialog:
        DialogBox(hInst, "ListBox", hWnd, ListBoxExampleDialogProc);
        break;
       case IDM_Exit:
        SendMessage(hWnd, WM_CLOSE, 0, 0);
        break;
      }
     break;
    case WM_DESTROY:
     PostQuitMessage(0);
     return 0;
   }
  return DefWindowProc(hWnd,Message,wParam, lParam);
 }

BOOL CALLBACK ListBoxExampleDialogProc(HWND hDlg, UINT Message,
                                       WPARAM wParam,
                                       LPARAM lParam)
 {
  int i;
// I like «Ivanov, Petrov, Sidorov ...» TV-program. ☺
  LPSTR pszItems[12] = {"Berdiev", "Vasilkov", "Ivanov", "Petrov",
                "Sidorov", "Johnson", "Jackson", "Tompson",
                "Pererepenko", "Khabibullin", "Novozhenov",
                "Mamedov"};
  char cMessage[36] = "Message about ";
  char cItem[12];
  static HWND hListBox;

  switch(Message)
   {
    case WM_INITDIALOG:
     hListBox = GetDlgItem(hDlg, ID_MyListBox);
     for(i = 0; i < 12; i++)
      SendMessage(hListBox, LB_ADDSTRING, (WPARAM) 0, (LPARAM)
                  pszItems[i]);
     return TRUE;
```

```
    case WM_COMMAND:
     switch(LOWORD(wParam))
      {
       case ID_MyListBox:
        if(HIWORD(wParam) == LBN_SELCHANGE)
          {
           SendMessage(hListBox, LB_GETTEXT, SendMessage(hListBox,
                       LB_GETCURSEL, 0, 0), (LPARAM) cItem);
           strcpy(cMessage + 14, cItem);
           SendDlgItemMessage(hDlg, ID_StatusBar, SB_SETTEXT,
                              (WPARAM) 0, (LPARAM) cMessage);
          }
        break;
       case ID_Cancel:
        EndDialog(hDlg, 0);
        break;
      }
     break;
   }
  return FALSE;
}
```

В этой программе основное меню окна предлагает отобразить диалог. При запуске диалога (при обработке сообщения WM_INITDIALOG) производится заполнение списка фамилиями. При переносе выделения с выбранного элемента окна списка на другой элемент в строке состояния отображается выбранная фамилия. Таким образом, я продемонстрировал заполнение списка и выборку информации из него, а также работу по обработке нотификационного сообщения. Вид создаваемого программой окна со списком показан на рис. 10.

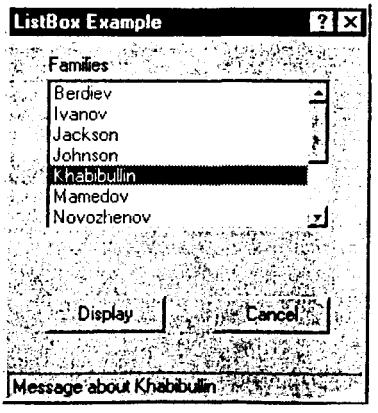


Рис. 10. Диалоговое окно, содержащее окно списка

Следующим шагом в изучении органов управления, применяющихся в диалоговых окнах, является изучение достаточно интересного элемента, который называется

ОКНО РЕДАКТИРОВАНИЯ

Окно редактирования - это один из наиболее сложных (с точки зрения реализации, а не использования) и наиболее интересных элементов управления. Фактически этот элемент представляет собой небольшой текстовый редактор, который позволяет вводить текст, редактировать его, копировать в буфер, вставлять из буфера и т. д. Окна редактирования могу быть однострочными и многострочными. Однострочные окна редактирования обычно используются для ввода небольших элементов текста. Например, в Program Manager'е для запуска программы пользователю необходимо ввести имя и командную строку этой программы. Примером редактора, большую часть функциональности которого обеспечивается за счет многострочного окна редактирования, является Notepad, который поставляется со всеми версиями Windows.

Окно редактирования можно создать как в файле ресурсов, так и как отдельное дочернее окно, указав при этом предопределенный класс «edit». В этом разделе мы разберем оба случая использования окна редактирования.

Перед тем как начать изучение, давайте вспомним, что все поведение элемента управления зависит от того, какие стили мы укажем при его создании. Все стили окна упомянуты в winuser.h. Все они начинаются с букв ES_. Эти стили приведены в табл. 31.

Таблица 31. Стили окна редактирования

Стиль	Значение	Описание
ES_LEFT	0x0000L	Текст в окне редактирования выравнивается по левому краю
ES_CENTER	0x0001L	Текст в окне редактирования выравнивается по правому краю
ES_RIGHT	0x0002L	Текст в окне редактирования выравнивается по центру
ES_MULTILINE	0x0004L	Создается многострочное окно редактирования
ES_UPPERCASE	0x0008L	Вводимый текст преобразуется в прописные буквы
ES_LOWERCASE	0x0010L	Вводимый текст преобразуется в строчные буквы
ES_PASSWORD	0x0020L	Все вводимые символы отображаются в виде звездочек

Стиль	Значение	Описание
ES_AUTOVSCROLL	0x0040L	При необходимости текст в многострочном окне редактирования скроллируется по вертикали
ES_AUTOHSCROLL	0x0080L	При необходимости текст в окне редактирования скроллируется по горизонтали
ES_NOHIDESEL	0x0100L	При потере окном редактирования фокуса ввода выделение с текста не снимается
ES_OEMCONVERT	0x0400L	Вводимые символы из одного набора преобразуются в символы из другого набора
ES_READONLY	0x0800L	Текст в окне редактирования можно только просматривать, но не редактировать
ES_WANTRETURN	0x1000L	При нажатии клавиши Enter в многострочном окне система вставляет в текст символ возврата каретки
ES_NUMBER	0x2000L	Разрешается осуществлять ввод только цифр

Рассмотрим случай создания окна редактирования обычным образом, без использования ресурсов. Надеюсь, что создание окна редактирования (как элемента диалогового окна в файле ресурсов) никаких проблем не вызовет. Для того чтобы увидеть возможности окна редактирования, разберем небольшую демонстрационную программу:

```
#include <windows.h>
#define ID_Edit 101

HINSTANCE hInst;
LRESULT CALLBACK EditDemoWndProc ( HWND, UINT, UINT, LONG );

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow )
 {
  HWND hWnd ;
  WNDCLASS WndClass ;
  MSG Msg;
  char szClassName[] = "EditDemo";

  hInst = hInstance;
/* Registering our window class */
/* Fill WNDCLASS structure */
  WndClass.style = CS_HREDRAW | CS_VREDRAW;
  WndClass.lpfnWndProc = EditDemoWndProc;
  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 = szClassName;

    if ( !RegisterClass(&WndClass) )
      {
       MessageBox(NULL,"Cannot register class","Error",MB_OK);
       return 0;
      }
    hWnd = CreateWindow(szClassName, "EditDemo",
                        WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, NULL, NULL,
                        hInstance,NULL);
    if(!hWnd)
      {
       MessageBox(NULL,"Cannot create window","Error",MB_OK);
       return 0;
      }

/* Show our window */
    ShowWindow(hWnd,nCmdShow);
    UpdateWindow(hWnd);
/* Beginning of messages cycle */
    while(GetMessage(&Msg, NULL, 0, 0))
      {
       TranslateMessage(&Msg);
       DispatchMessage(&Msg);
      }
    return Msg.wParam;
  }

LRESULT CALLBACK EditDemoWndProc (HWND hWnd, UINT Message,
                                  UINT wParam, LONG lParam )
  {
    static HWND hEditWnd;
    RECT Rect;

    switch(Message)
      {
       case WM_CREATE:
         GetClientRect(hWnd, &Rect);
         hEditWnd = CreateWindow("edit", NULL,
                                 WS_CHILD | WS_VISIBLE |
                                 WS_HSCROLL | WS_VSCROLL |
                                 WS_BORDER | ES_LEFT |
                                 ES_MULTILINE | ES_AUTOHSCROLL |
                                 ES_AUTOVSCROLL,
                                 0, 0, 0, 0,
```

```
                                    hWnd,
                                    (HMENU) ID_Edit,
                                    hInst,
                                    NULL);
      return 0;
    case WM_SIZE:
      MoveWindow(hEditWnd, 0, 0, LOWORD(lParam), HIWORD(lParam),
                 TRUE);
      return 0;
    case WM_SETFOCUS:
      SetFocus(hEditWnd);
      return 0;
    case WM_DESTROY:
       PostQuitMessage(0);
       return 0;
  }
  return DefWindowProc(hWnd,Message,wParam, lParam);
}
```

Вид окна, создаваемого программой, показан на рис. 11.

Эта программа создает окно редактирования, которое располагается поверх основного окна программы. Фактически в программе создается текстовый редактор, позволяющий осуществлять набор и редактирование текста, выделять части текста. Выделенные части текста могут быть перемещены в Clipboard посредством нажатия клавиш Shift-Delete, а после нажатия клавиш Shift-Insert текст из Clipboard'а может быть вставлен в окно.

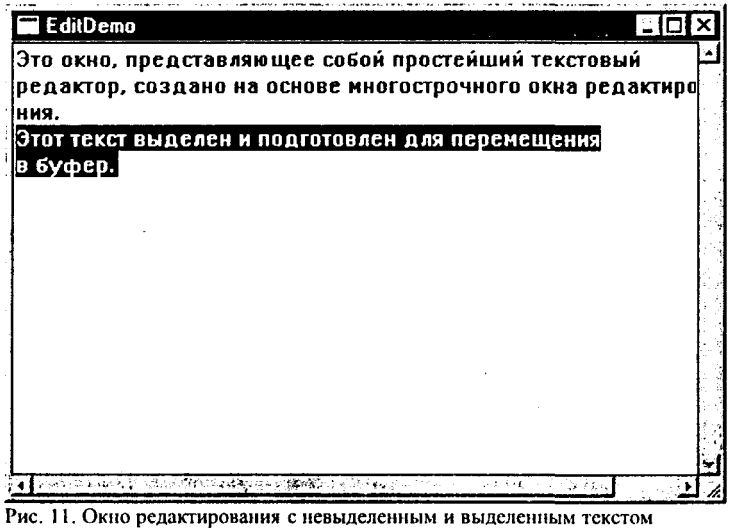


Рис. 11. Окно редактирования с невыделенным и выделенным текстом

Управление окном редактирования, как и всех остальных элементов управления, осуществляется посредством посылки окну сообщений.

Перед тем, как начать рассмотрение сообщений необходимо отметить, что для нормальной работы часть текста в окне редактирования должна быть выделена. Операции производятся именно с выделенной частью текста. Копирование текста в окно редактирования осуществляется от места размещения текстового курсора.

Следующие пять сообщений не имеют параметров, а их действия очевидны из их названия:

WM_COPY - выделенная часть текста копируется в Clipboard;

WM_PASTE - содержимое Clipboard'а копируется в окно редактирования (данные вставляются только в том случае, если в Clipboard'е находится текст);

WM_CUT - выделенная часть текста удаляется из окна редактирования и помещается в Clipboard;

WM_CLEAR - выделенная часть текста удаляется из окна редактирования и не помещается в Clipboard;

WM_UNDO - отменяется последняя операция.

Для того чтобы получить границы выделения текста, необходимо использовать сообщение EM_GETSEL. Младшее слово возвращаемого значения содержит начальную, а старшее слово - конечную позицию выделения плюс 1. Другими словами,

```
DWORD dwPosition = SendMessage(hEditWnd, EM_GETSEL,
                                (WPARAM) 0, (LPARAM) 0);
WORD wBeginPosition = LOWORD(dwPosition);
WORD wEndPosition = HIWORD(dwPosition) - 1;
```

Если программе необходимо выделить часть текста, то она может использовать сообщение EM_SETCEL, в lParam которого необходимо указать начальную и конечную позиции выделения:

```
SendMessage(hEditWnd, EM_SETSEL, 0, MAKELONG(wBeginPosition,
        wEndPosition));
```

С помощью сообщения EM_REPLACESEL программа может заменить выделенный текст на другой:

```
SendMessage(hEditWnd, EM_REPLACESEL, 0, (LPARAM) pszNewText);
```

И наконец, посылка сообщения, с помощью которого приложение может скопировать в собственный буфер набранный пользователем текст, выглядит следующим образом:

```
SendMessage(hEditWnd, EM_GETLINE, (WPARAM) nLine,
            (LPARAM) (LPCSTR) pBuffer);
```

В этом сообщении в качестве wParam указывается номер строки (в случае однострочного окна номер строки игнорируется), а в качестве lParam - указатель на буфер, в который будет записана строка из окна редактирования.

На этом завершается рассмотрение стандартных элементов управления.

Должен заметить, что в Win32 включены новые, так называемые общие элементы управления (common controls). Многие из них уже были реализованы в приложениях, работающих в среде Windows 3.x, но до их документирования в Windows 3.x дело не дошло. Встречались попытки описать их реализацию в частности, в MSDN были описаны строка состояния, панель инструментов (toolbar), окно просмотра деревьев. Поэтому, наверное, можно сказать, что появление общих элементов управления ожидалось. И теперь мы приступаем к их изучению.

ОБЩИЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ

Перед тем, как мы начнем изучение работы непосредственно элементов управления, мы должны научиться подключать библиотеку, реализующую эти элементы. Для нормальной работы программы с общими элементами управления необходимо выполнить два шага. Первым шагом является подключение к программе файла заголовков commctrl.h. Вторым шагом является подключение при линковании непосредственно библиотеки. Она называется comctl32.dll.

Перед использованием любого из общих элементов управления необходимо загрузить эту библиотеку. Это делается с помощью функции InitCommonControls(), которая описана в файле commctrl.h как

```
void InitCommonContorls(void);
```

Эта функция не только загружает библиотеку общих элементов управления, но и инициализирует их подсистему. Обычно эта функция вызывается перед первым использованием одного из элементов.

Характерно, что все общие элементы управления являются дочерними окнами, т. е. они не могут создаваться и использоваться в качестве главного окна программы. Управление ими, как и обычными элементами управления, осуществляется посредством посылки им сообщений. Элементы управления информируют родительское окно о событиях, произошедших с ними, посредством передачи нотификационных сообщений.

Первым элементом управления, работу с которым мы рассмотрим, будет строка состояния (status bar). Ранее мы использовали ее в наших демонстрационных программах, но работу с ней не изучали.

РАБОТА СО СТРОКОЙ СОСТОЯНИЯ

Для того чтобы отобразить информацию о текущем состоянии программы, выполняемых операциях и режимах, в программе может использоваться элемент управления, который называют окном или линейкой состояния (status bar). Мне кажется более удачным термин «строка состояния». Включение окна состояния в программу в значительной степени изменяет внешний вид окна и позволяет создать более понятный и удобный для пользователя интерфейс.

Строка состояния может быть включена в описание диалогового окна в файле ресурсов. Но так как окно состояния является стандартным окном, то, естественно, что для его создания могут быть использованы функции и стандартные функции CreateWindow() и CreateWindowEx(). При этом в качестве имени класса окна необходимо задать макро «STATUSCLASSNAME». В зависимости от того, какую систему кодировки использует программа, можно воспользоваться также и «истинным» именем класса (msctls_statusbar32). В commctrl.h эти макро и имена описаны следующим образом:

```
#ifdef _WIN32
#define STATUSCLASSNAMEW     L"msctls_statusbar32"
#define STATUSCLASSNAMEA     "msctls_statusbar32"

#ifdef UNICODE
#define STATUSCLASSNAME      STATUSCLASSNAMEW
#else
#define STATUSCLASSNAME      STATUSCLASSNAMEA
#endif

#else
#define STATUSCLASSNAME      "msctls_statusbar"
#endif
```

Тем не менее, для создания окна состояния предусмотрена и отдельная функция CreateStatusWindow(). В файле commctrl.h эта функция определяется так:

```
WINCOMMCTRLAPI HWND WINAPI CreateStatusWindowA(LONG style,
                                               LPCSTR lpszText,
                                               HWND hwndParent,
                                               UINT wID);
WINCOMMCTRLAPI HWND WINAPI CreateStatusWindowW(LONG style,
                                               LPCWSTR lpszText,
                                               HWND hwndParent,
                                               UINT wID);

#ifdef UNICODE
#define CreateStatusWindow    CreateStatusWindowW
#else
#define CreateStatusWindow    CreateStatusWindowA
#endif
```

Как можно узнать из описания функции, она требует передачи ей четырех аргументов. Первый аргумент, style, должен определять стиль создаваемого окна. У строки состояния есть единственный собственный стиль, SBARS_SIZEGRIP, который позволяет в правый угол строки состояния добавить «ручку» (внешне, честно говоря, на ручку это совсем не похоже) для изменения ее размеров. Но наличие у строки состояния единственного стиля не мешает комбинировать его со стандартными стилями окон, например с WS_CHILD и WS_VISIBLE.

Второй аргумент - lpszText - является указателем на строку, которая будет отображена в строке состояния сразу после ее создания. Ничего особенного здесь нет.

Третий аргумент - hwndParent - тоже не требует особых объяснений. Он является хэндлом родительского окна строки состояния.

Наконец, четвертый аргумент - uID - идентификатор окна состояния.

Попробуйте вызвать эту функцию (не забудьте про InitCommonControls()!) - и вы увидите, что в окне появилась строка состояния с текстом, определенным во втором аргументе этой функции.

После того, как строка состояния прорисована, иногда бывает необходимо разделить ее на несколько панелей для того, чтобы в каждой панели отображать информацию, логически не связанную с отображаемой в других панелях. Для того чтобы сделать это, необходимо послать строке состояния сообщение SB_SETPARTS. При этом wParam этого сообщения должен определять число панелей, а lParam должен содержать указатель

на массив целых чисел (число элементов массива должно быть равно wParam). Каждый элемент в этом массиве должен определять позицию (в координатах родительского окна) правой границы соответствующей части. Если элемент равен -1, то границей панели считается правая граница строки состояния. В случае успешного завершения операции функция, с помощью которой послано сообщение, возвращает TRUE. Значение FALSE должно заставить программиста поискать ошибку в собственной программе.

Приложение может определить число панелей, на которые разделена строка состояния, и их координаты с помощью сообщения SB_GETPARTS. Если lParam этого сообщения равен нулю, то функция, пославшая сообщение, возвращает число панелей строки состояния. При этом значение wParam роли не играет. Для того чтобы получить координаты панелей, wParam должен определять число панелей, для которых нужно получить координаты, а lParam должен указывать на массив целых чисел, в который будут записаны эти координаты. В этом случае функция также возвращает число панелей. Если при обработке сообщения произошла какая-то ошибка, то функция возвращает нулевое значение.

Итак, строка состояния сформирована. Но зачем она нужна без отображенной информации? Для того чтобы отобразить определенный текст в строке состояния, нужно послать ей сообщение SB_SETTEXT. В качестве wParam этого сообщения используется результат логического сложения двух величин. Первая (iPart) - номер (считая от нуля) панели, в которой необходимо отобразить текст. Вторая (uType) определяет, как будет выглядеть текст. В качестве uType могут быть использованы значения, приведенные в табл. 32.

Таблица 32. Возможные типы строки состояния

Тип	Значение	Описание
	0x0000	Текст кажется вдавленным в панель
SBT_NOBORDERS	0x0100	Панель прорисовывается без ограничительных линий
SBT_POPOUT	0x0200	Панель прорисовывается выпуклой
SBT_RTLREADING	0x0400	Используется для языков, в которых чтение идет справа налево, как, например, в арабском
SBT_OWNERDRAW	0x1000	За прорисовку панели отвечает родительское окно

lParam сообщения должен содержать указатель на строку, которую необходимо отобразить в панели строки состояния.

А теперь, для того, чтобы прочитать текст в панели, необходимо строке состояния послать сообщение WM_GETTEXT. wParam этого сообщения должен содержать номер панели, а lParam - указатель на строку, в которую будет записан текст, содержащийся в панели.

Это основные сообщения, используемые при работе со строкой состояния.

Для иллюстрации некоторых возможностей строки состояния, о которых я только что рассказал, рассмотрим несколько измененную программу из раздела о кнопках. Из-за мелких отличий не буду приводить текст программы целиком, а приведу только текст диалоговой функции:

```
BOOL CALLBACK ButtonsExampleDialogProc(HWND hDlg,
                                       UINT Message,
                                       WPARAM wParam,
                                       LPARAM lParam)
 {
  int i;
  char cMyMessage[80];
  RECT Rect;
  int nBorders[3];
  switch(Message)
   {
    case WM_INITDIALOG:
// Set states of controls
     SendDlgItemMessage(hDlg, nRadioButtonId, BM_SETCHECK,
                        BST_CHECKED, 0);
     for(i = IDC_CHECKBOX1; i <= IDC_CHECKBOX3; i++)
      if(uCheckBoxesState[i - 208])
        SendDlgItemMessage(hDlg, i, BM_SETCHECK, BST_CHECKED, 0);
     GetClientRect(hDlg, &Rect);
     nBorders[0] = Rect.right / 3;
     nBorders[1] = Rect.right / 3 * 2;
     nBorders[2] = -1;
     SendDlgItemMessage(hDlg, IDC_STATUSBAR, SB_SETPARTS, 3,
                        (LPARAM) nBorders);
     return TRUE;
    case WM_COMMAND:
     switch(LOWORD(wParam))
      {
       case IDC_RADIOBUTTON1:
       case IDC_RADIOBUTTON2:
       case IDC_RADIOBUTTON3:
        sprintf(cMyMessage,"RadioButton%d", LOWORD(wParam) - 203);
        SendDlgItemMessage(hDlg,IDC_STATUSBAR,SB_SETTEXT,
                           (WPARAM) 0, (LPARAM) cMyMessage);
```

```
            CheckMenuRadioItem(GetSubMenu(GetSubMenu(GetMenu(hWnd), 1),  0),
                               IDM_RadioButton1, IDM_RadioButton3,
                               LOWORD(wParam) - 102,
                               MF_BYCOMMAND);
            return FALSE;
           case IDC_CHECKBOX1:
           case IDC_CHECKBOX2:
           case IDC_CHECKBOX3:
            sprintf(cMyMessage,"CheckBox%d", LOWORD(wParam) - 207);
            SendDlgItemMessage(hDlg,IDC_STATUSBAR,SB_SETTEXT,
                               (WPARAM) 1, (LPARAM) cMyMessage);
            i = LOWORD(wParam) - 208;
            uCheckBoxesState[i] = uCheckBoxesState[i] == MF_CHECKED ?
                  MF_UNCHECKED : MF_CHECKED;
            CheckMenuItem(GetSubMenu(GetSubMenu(GetMenu(hWnd), 1), 1),
                          LOWORD(wParam) - 103, uCheckBoxesState[i]);
            return FALSE;
           case IDC_BUTTON1:
            SendDlgItemMessage(hDlg, IDC_STATUSBAR, SB_SETTEXT,
                               (WPARAM) 2, (LPARAM) "PushButton");
            return TRUE;
           case IDC_BUTTON2:
// Save the state of RadioButtons
            i = IDC_RADIOBUTTON1;
            while(!SendDlgItemMessage(hDlg, i, BM_GETCHECK, 0, 0))
              i++;
            nRadioButtonId = i;
// Save the state of CheckButtons
            for(i = IDC_CHECKBOX1; i <= IDC_CHECKBOX3; i++)
              uCheckBoxesState[i - 208] = SendDlgItemMessage(hDlg, i,
                              BM_GETCHECK, 0, 0) == 0 ?
                              MF_UNCHECKED :
                              MF_CHECKED;
            EndDialog(hDlg,0);
            return TRUE;
         }
        break;
     }
    return FALSE;
  }
```

Отличия, внесенные в эту функцию, весьма невелики. Во-первых, строка состояния делится на три панели, и, во-вторых, каждый кластер объектов управления отображает сообщения в своей панели. Как видите, уважаемый читатель, ничего сложного здесь нет. Вид диалогового окна, создаваемого программой, показан на рис. 12.

Сравните внешний вид строки состояния, приведенной на рисунке, и строки состояния из раздела о кнопках.

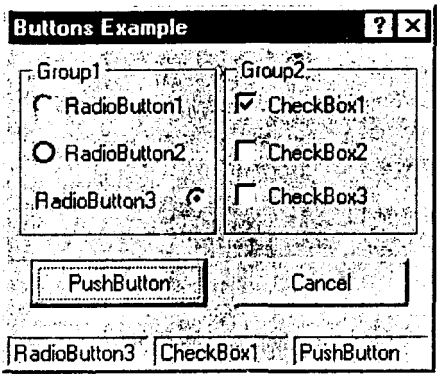


Рис. 12. Диалоговое окно со строкой состояния

РАБОТА СО СПИНОМ

Иногда в приложениях встречаются ситуации, в которых полосы прокрутки (скроллирования) не нужны, достаточно только кнопок «вверх» и «вниз». Ярким примером такой ситуации может служить окно, открываемое в WinWord'е для Windows'95 при необходимости определить параметры страницы. Естественно, полоса прокрутки для того, чтобы чуть увеличить значение размера бумаги или сделать поля поменьше, не нужна. Для подобных случаев в Win32 предусмотрен новый элемент управления, называемый спином. Спин является особым видом линейки прокрутки и состоит только из кнопок со стрелками, которые находятся на концах линейки, и не включает линейки прокрутки. Обычно спин используется в одном из двух вариантов. Во-первых, он может применяться а la маленькая линейка прокрутки. В этом случае его называют up-down control'ом. Во-вторых, часто он используется в сочетании с другим элементом управления, называемым в этом случае buddy window (приятельским окном). Как правило, этим приятельским окном оказывается окно редактирования. Кстати, в приведенном ниже примере именно окна редактирования и оказываются приятельскими окнами. В этом случае элемент управления называется spin'ом. В данном разделе под словом «спин» будем понимать как up-down control, так и собственно спин.

Спин может создаваться и в составе диалогового окна в файле ресурсов, и как обычное окно посредством использования функций CreateWindow() или CreateWindowEx(). При этом в качестве имени класса необходимо указывать «UPDOWN_CLASS». Соответствующее описание можно встретить в comctl.h:

```
#ifdef _WIN32

#define UPDOWN_CLASSA       "msctls_updown32"
#define UPDOWN_CLASSW       L"msctls_updown32"

#ifdef UNICODE
#define  UPDOWN_CLASS       UPDOWN_CLASSW
#else
#define  UPDOWN_CLASS       UPDOWN_CLASSA
#endif

#else
#define UPDOWN_CLASS        "msctls_updown"
#endif
```

Тем не менее, для создания спина создана специальная функция CreateUpDownControl(), описание которого мы можем найти в файле commctrl.h:

```
WINCOMMCTRLAPI HWND WINAPI CreateUpDownControl(
                                DWORD dwStyle,
                                int x, int y, int cx, int cy,
                                HWND hParent, int nID,
                                HINSTANCE hInst, HWND hBuddy,
                                 int nUpper, int nLower, int nPos);
```

Таблица 33. Стили спина

Стиль	Значение	Описание
UDS_WRAP	0x0001	При достижении максимальной позиции отсчет начинается вновь с минимальной позиции и наоборот.
UDS_SETBUDDYINT	0x0002	У спина есть приятельское окно
UDS_ALIGNRIGHT	0x0004	Спин размещается справа от приятельского окна
UDS_ALIGNLEFT	0x0008	Спин размещается слева от приятельского окна
UDS_AUTOBUDDY	0x0010	При изменении позиции спина текст в приятельском окне меняется автоматически
UDS_ARROWKEYS	0x0020	Разрешается использование клавиатуры для изменения текущего состояния спина
UDDS_HORZ	0x0040	Спин располагается горизонтально
UDS_NOTHOUSANDS	0x0080	Не отображать запятую для разделения классов в числах, отображаемых в приятельском окне

Таблица 34. Сообщения, посылаемые спину

Сообщение	Значение	Описание
UDM_SETRANGE	WM_USER+101	Установка диапазона прокрутки, wParam = 0, min значение - в старшем слове lParam, max значение - в младшем слове lParam
UDM_GETRANGE	WM_USER+102	Получение диапазона прокрутки, wParam и lParam должны быть равны 0, возвращает max значение в младшем, а min значение - в старшем слове возвращаемого значения
UDM_SETPOS	WM_USER+103	Установка текущей позиции спина, wParam=0, lParam - значение новой позиции
UDM_GETPOS	WM_USER+104	Получение текущей позиции спина, wParam и lParam должны быть равны 0, возвращает текущую позицию в младшем слове возвращаемого значения
UDM_SETBUDDY	WM_USER+105	Определение приятельского окна, wParam хэндлу приятельского окна, lParam = 0, возвращает хэндл бывшего приятельского окна
UDM_GETBUDDY	WM_USER+106	Получение хэндла приятельского окна, wParam и lParam должны быть равны 0, возвращает хэндл приятельского окна в младшем слове возвращаемого значения

Параметр dwStyle определяет стиль окна. Все стили, разработанные специально для спина, начинаются с UDS_. Они приведены в табл. 33.

Как и всегда, для управления спином используются сообщения. Обычно при нажатии одной из стрелок спина родительскому окну посылается сообщение WM_VSCROLL. При этом lParam этого сообщения содержит хэндл спина. При обработке этого сообщения необходимо быть аккуратным и не забывать, что сообщения WM_VSCROLL могут поступать и от других элементов управления.

Остальные сообщения, используемые спином, приведены в табл. 34.

Для того чтобы проиллюстрировать возможности спина, рассмотрим небольшую демонстрационную программу. Для работы этой программы вам потребуется файл описаний, приведенный ниже:

```
#define IDM_Exit      101
#define IDM_Dialog    102
#define IDM_About     103
#define ID_OK         104
#define ID_Edit       105
#define ID_Spin       106
```

Кроме этого, ниже приведен файл ресурсов, используемый программой:

```
#include "spin.h"

SpinDemoMenu MENU
 {
  POPUP "&File"
   {
    MENUITEM "E&xit", IDM_Exit
   }

  MENUITEM "&Dialog", IDM_Dialog

  POPUP "&Help"
   {
    MENUITEM "&About", IDM_About
   }

 }
SpinDemoDialog DIALOG 0, 0, 100, 100
STYLE DS_MODALFRAME | DS_3DLOOK | WS_POPUP | WS_VISIBLE |
        WS_CAPTION | WS_SYSMENU
CAPTION "Spin Demo Dialog"
FONT 8, "MS Sans Serif"
 {
  DEFPUSHBUTTON "OK", ID_OK, 25, 60, 50, 14
  CONTROL "", ID_Edit, "edit", ES_LEFT | WS_CHILD | WS_VISIBLE |
              WS_BORDER | WS_TABSTOP | ES_NUMBER, 25, 20, 50, 12
 }
```

И наконец, непосредственно текст программы:

```
#include <windows.h>
#include <commctrl.h>
#include "spin.h"

HINSTANCE hInst;

LRESULT CALLBACK SpinDemoWndProc(HWND, UINT, UINT, LONG );
BOOL CALLBACK SpinDemoDialogProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow )
 {
  HWND hWnd ;
  WNDCLASS WndClass ;
  MSG Msg;
  char szClassName[] = "SpinDemo";
  hInst = hInstance;
```

```
/* Registering our window class */
/* Fill WNDCLASS structure */
   WndClass.style = CS_HREDRAW | CS_VREDRAW;
   WndClass.lpfnWndProc = SpinDemoWndProc;
   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 = "SpinDemoMenu";
   WndClass.lpszClassName = szClassName;

   if ( !RegisterClass(&WndClass) )
      {
       MessageBox(NULL,"Cannot register class","Error",MB_OK);
       return 0;
      }

   hWnd = CreateWindow(szClassName, "Spin Demo",
                       WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                       CW_USEDEFAULT, NULL, NULL,
                       hInstance,NULL);
   if(!hWnd)
      {
       MessageBox(NULL,"Cannot create window","Error",MB_OK);
       return 0;
      }

   InitCommonControls();
   /* Show our window */
   ShowWindow(hWnd,nCmdShow);
   UpdateWindow(hWnd);

/* Beginning of messages cycle */
   while(GetMessage(&Msg, NULL, 0, 0))
     {
      TranslateMessage(&Msg);
      DispatchMessage(&Msg);
     }
   return Msg.wParam;

  }

LRESULT CALLBACK SpinDemoWndProc (HWND hWnd, UINT Message,
                                  UINT wParam, LONG lParam )
  {
   switch(Message)
```

```
    {
     case WM_COMMAND:
      switch(wParam)
       {
        case IDM_Exit:
         SendMessage(hWnd, WM_CLOSE, 0, 0);
         break;
        case IDM_Dialog:
         DialogBox(hInst, "SpinDemoDialog", hWnd, SpinDemoDialogProc);
         break;
       }
      return 0;
     case WM_DESTROY:
      PostQuitMessage(0);
      return 0;
    }
  return DefWindowProc(hWnd,Message,wParam, lParam);
}

BOOL CALLBACK SpinDemoDialogProc(HWND hDlg, UINT Message,
                                 WPARAM wParam, LPARAM lParam)
{
  static HWND hEditWnd;
  static HWND hSpinWnd;
  switch(Message)
    {
     case WM_INITDIALOG:
      hEditWnd = GetDlgItem(hDlg,ID_Edit);
      hSpinWnd = CreateUpDownControl(WS_CHILD | WS_BORDER |
                                 WS_VISIBLE |
                                 UDS_SETBUDDYINT |
                                 UDS_ALIGNRIGHT,
                                  0, 12, 50, 50,
                                 hDlg, ID_Spin,
                                 hInst,
                                 hEditWnd,
                                 100, 0, 50);
      return TRUE;
     case WM_COMMAND:
      switch(LOWORD(wParam))
       {
         case ID_OK:
          EndDialog(hDlg,0);
          return TRUE;
       }
      break;
    }
  return FALSE;
}
```

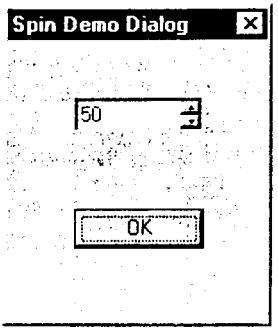


Рис. 13. Диалоговое окно со спином

При выборе элемента «Dialog» из меню программы производится отображение диалогового окна, вид которого показан на рис. 13.

Действия, приводящие к отображению спина, локализованы в той части программы, которая обрабатывает сообщение WM_INITDIALOG. Спин создается посредством вызова функции CreateUpDownControl(), при этом в качестве приятельского окна указывается окно редактирования, созданное как часть ресурса с описанием диалогового окна. Все остальные действия производятся автоматически. Читатель может попробовать изменить стили спина и посмотреть, к чему это приведет.

Таблица 35. Стили трекбара

Стиль	Значение	Описание
TBS_HORZ	0x0000	Определяет горизонтальную ориентацию трекбара
TBS_BOTTOM	0x0000	Шкала расположена под ползунком (для горизонтального трекбара)
TBS_RIGHT	0x0000	Шкала расположена справа от трекбара (для вертикального трекбара)
TBS_AUTOTICKS	0x0001	Шкала трекбара создается с делениями
TBS_VERT	0x0002	Определяет вертикальную ориентацию трекбара
TBS_TOP	0x0004	Шкала расположена над ползунком (для горизонтального трекбара)
TBS_LEFT	0x0004	Шкала расположена слева от трекбара (для вертикального трекбара)
TBS_BOTH	0x0008	Шкала расположена с двух сторон трекбара
TBS_NOTICKS	0x0010	Шкала трекбара создается без делений
TBS_ENABLESELRANGE	0x0020	Разрешается отображение диапазона
TBS_FIXEDLENGTH	0x0040	При изменении диапазона длина трекбара не изменяется
TBS_NOTHUMB	0x0080	У трекбара нет слайдера

РАБОТА С ТРЕКБАРОМ

Очередным клоном линейки прокрутки является ползунок (trackbar или slider). Его внешний вид достаточно эффектен и интересен. Он напоминает регулятор, используемый в аппаратуре, скажем, в качестве регулятора громкости. Небольшим отличием трекбара от линейки прокрутки является то, что у ползунка есть шкала, вдоль которой он движется. Честно говоря, мне очень не нравится переводить на русский слова, к которым я уже привык и которые обычно используются в качестве программистского сленга. Поэтому давайте будем в данном случае под словом «трекбар» понимать весь элемент управления, а под словом «слайдер» - указатель, движущийся вдоль шкалы.

К сожалению, для создания трекбара не предусмотрено специальной функции, поэтому создавать его необходимо посредством вызова функции CreateWindow() или CreateWindowEx(). При этом в качестве имени класса следует указать макрос TRACKBAR_CLASS, который описан в commctrl.h:

```
#ifdef _WIN32

#define TRACKBAR_CLASSA      "msctls_trackbar32"
#define TRACKBAR_CLASSW      L"msctls_trackbar32"

#ifdef UNICODE
#define  TRACKBAR_CLASS      TRACKBAR_CLASSW
#else
#define  TRACKBAR_CLASS      TRACKBAR_CLASSA
#endif

#else
#define TRACKBAR_CLASS       "msctls_trackbar"
#endif
```

При создании трекбара могут использоваться стили окна, идентификаторы которых начинаются с TBS_ (табл. 35).

Что еще можно сказать об этих стилях? По-моему, здесь все ясно. Даже понятно, что стили TBS_HORZ, TBS_BOTTOM и TBS_RIGHT являются стилями, принимаемыми по умолчанию.

А теперь настало время рассмотреть сообщения, посредством которых осуществляется управление трекбаром. Все эти сообщения приведены в табл. 36.

Таблица 36. Сообщения, посылаемые трекбару

Сообщение	*Значение*	*Описание*
TBM_GETPOS	WM_USER	wParam и lParam = 0, возвращает текущую позицию слайдера
TBM_GETRANGEMIN	WM_USER + 1	wParam и lParam = 0, возвращает нижнюю границу диапазона прокрутки
TBM_GETRANGEMAX	WM_USER + 2	wParam и lParam = 0, возвращает верхнюю границу диапазона прокрутки
TBM_GETTIC	WM_USER + 3	wParam = номер (от нуля) пометки, lParam = 0, возвращает позицию в диапазоне, соответствующую указанной пометке
TBM_SETTIC	WM_USER + 4	wParam = 0, lParam = позиции, устанавливает пометку в указанной позиции
TBM_SETPOS	WM_USER + 5	wParam = TRUE, lParam = новой позиции слайдера, слайдер устанавливается в новую позицию
TBM_SETRANGE	WM_USER + 6	wParam = флаг перерисовки (BOOL), lParam = MAKELONG(wMinimum, wMaximum), устанавливает диапазон для слайдера трекбара
TBM_SETRANGEMIN	WM_USER + 7	wParam = флаг перерисовки (BOOL), lParam = wMinimum, установка нижней границы диапазона для слайдера
TBM_SETRANGEMAX	WM_USER + 8	wParam = флаг перерисовки (BOOL), lParam = wMaximum, установка верхней границы диапазона для слайдера
TBM_CLEARTICS	WM_USER + 9	wParam = флаг перерисовки (BOOL), lParam = 0, удаляет текущую пометку
TBM_SETSEL	WM_USER + 10	wParam = флаг перерисовки (BOOL), lParam = MAKELONG(wMinimum, wMaximum, установка диапазона выделения в трекбаре
TBM_SETSELSTART	WM_USER + 11	wParam = флаг перерисовки (BOOL), lParam = wStart, установка нижней границы выделения
TBM_SETSELEND	WM_USER + 12	wParam = флаг перерисовки (BOOL), lParam = wEnd, установка верхней границы выделения
TBM_GETPTICS	WM_USER + 14	wParam и lParam = 0, возвращает указатель на массив, содержащий позиции пометок
TBM_GETTICPOS	WM_USER + 15	wParam = номер (от нуля) пометки, lParam = 0, возвращает позицию пометки в оконных координатах

Сообщение	Значение	Описание
TBM_GETNUMTICS	WM_USER + 16	wParam = 0, lParam = 0, возвращает число поместок
TBM_GETSELSTART	WM_USER + 17	wParam = 0, lParam = 0, возвращает нижнюю границу выделения
TBM_GETSELEND	WM_USER + 18	wParam = 0, lParam = 0, возвращает верхнюю границу выделения
TBM_CLEARSEL	WM_USER + 19	wParam = флаг перерисовки (BOOL), lParam = 0, сбрасывает текущее выделение
TBM_SETTICFREQ	WM_USER + 20	wParam = частоте следования пометок, lParam = номеру позиции, начиная с которой расставляются пометки, устанавливает шаг расстановки пометок
TBM_SETPAGESIZE	WM_USER + 21	wParam = 0, lParam = wPageSize, определяет, на сколько позиций необходимо передвинуть слайдер при получении сообщений TB_PAGEDOWN и TB_PAGEUP, возвращает предыдущее значение этого параметра
TBM_GETPAGESIZE	WM_USER + 22	wParam = 0, lParam = 0, возвращает значение параметра, о котором говорится в предыдущей строке таблицы
TBM_SETLINESIZE	WM_USER + 23	wParam = 0, lParam = wLineSize, определяет, на сколько позиций необходимо передвинуть слайдер при получении сообщений TB_LINEDOWN и TB_LINEUP, возвращает предыдущее значение этого параметра
TBM_GETLINESIZE	WM_USER + 24	wParam = 0, lParam = 0, возвращает значение параметра, о котором говорится в предыдущей строке таблицы
TBM_GETTHUMBRECT	WM_USER + 25	
TBM_GETCHANNELRECT	WM_USER + 26	
TBM_SETTHUMBRECT	WM_USER + 27	
TBM_GETTHUMBRECT	WM_USER + 28	

Таблица 37. Коды нотификации, посылаемые трекбаром

Сообщение	Значение	Описание
TB_LINEUP	0	Нажата VK_LEFT (стрелка влево) или VK_UP (стрелка вверх)
TB_LINEDOWN	1	Нажата VK_RIGHT (стрелка вправо) или VK_DOWN (стрелка вниз)
TB_PAGEUP	2	Нажата VK_NEXT (PageUp) или щелчок мышью перед слайдером
TB_PAGEDOWN	3	Нажата VK_PRIOR (PageDown) или щелчок мышью после слайдера
TB_THUMBPOSITION	4	Слайдер зафиксирован после протяжки с помощью мыши
TB_THUMBTRACK	5	Слайдер протягивается с помощью мыши
TB_TOP	6	Нажата VK_HOME (Home), слайдер устанавливается в положение, соответствующее верхней границе диапазона
TB_BOTTOM	7	Нажата VK_END (End), слайдер устанавливается в положение, соответствующее нижней границе диапазона
TB_ENDTRACK	8	Слайдер зафиксирован после перемещения с помощью клавиатуры

При манипуляциях с трекбаром последний посылает родительскому окну сообщения WM_HSCROLL. В младшем слове wParam содержится код нотификации, который определяет характер действия, произведенного с трекбаром. Старшее слово wParam содержит позицию слайдера в момент возникновения сообщения. Дескриптор окна трекбара записывается в lParam. В табл. 37 приведены нотификационные сообщения, используемые при работе с трекбаром.

Сообщениия WM_TOP, WM_BOTTOM, WM_LINEDOWN и WM_LINEUP посылаются только в том случае, если пользователь воздействует на трекбар с помощью клавиатуры, TB_THUMBPOSITION и TB_THUMBTRACK посылаются в случае работы с мышью, остальные сообщения могут посылаться в обоих случаях.

Теперь, изучив теоретически работу трекбара, мы можем рассмотреть программу, в которой демонстрируются возможности этого элемента управления. Для нормальной работы программы помимо основного файла с текстом программы необходимы еще два файла: описаний и ресурсов. Файл описаний приведен ниже:

```
#define IDM_Exit    101
#define IDM_Dialog  102
#define IDM_About   103
```

```
#define ID_OK          104
#define ID_Edit        105
#define ID_Spin        106
#define ID_Trackbar    107
```

А теперь - файл ресурсов.

```
#include "trackbar.h"
TrackbarMenu MENU
 {
  POPUP "&File"
   {
    MENUITEM "E&xit", IDM_Exit
   }
  MENUITEM "&Dialog", IDM_Dialog
  POPUP "&Help"
   {
    MENUITEM "&About", IDM_About
   }
 }
TrackbarDialog DIALOG 0, 0, 100, 100
STYLE DS_MODALFRAME | DS_3DLOOK | WS_POPUP | WS_VISIBLE |
        WS_CAPTION | WS_SYSMENU
CAPTION "Trackbar Demo Dialog"
FONT 8, "MS Sans Serif"
{
 DEFPUSHBUTTON "OK", ID_OK, 25, 73, 50, 14
 CONTROL "", ID_Edit, "edit", ES_LEFT | WS_CHILD | WS_VISIBLE |
              WS_BORDER | WS_TABSTOP | ES_NUMBER, 25, 14, 50, 12
}
```

А теперь - очередь основного файла программы.

```
#include <windows.h>
#include <commctrl.h>
#include "trackbar.h"

HINSTANCE hInst;

LRESULT CALLBACK TrackbarWndProc(HWND, UINT, UINT, LONG );
BOOL CALLBACK TrackbarDialogProc(HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow )
 {
  HWND hWnd ;
  WNDCLASS WndClass ;
  MSG Msg;
  char szClassName[] = "TrackbarDemo";
  hInst = hInstance;
/* Registering our window class */
/* Fill WNDCLASS structure */
  WndClass.style = CS_HREDRAW | CS_VREDRAW;
```

```
  WndClass.lpfnWndProc = TrackbarWndProc;
  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 = "TrackbarMenu";
  WndClass.lpszClassName = szClassName;
  if ( !RegisterClass(&WndClass) )
    {
     MessageBox(NULL,"Cannot register class","Error",MB_OK);
     return 0;
    }

  hWnd = CreateWindow(szClassName, "Trackbar Demo Program",
                      WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                      CW_USEDEFAULT, CW_USEDEFAULT,
                      CW_USEDEFAULT, NULL, NULL,
                      hInstance,NULL);
  if(!hWnd)
    {
     MessageBox(NULL,"Cannot create window","Error",MB_OK);
     return 0;
    }

  InitCommonControls();
  /* Show our window */
  ShowWindow(hWnd,nCmdShow);
  UpdateWindow(hWnd);
/* Beginning of messages cycle */
  while(GetMessage(&Msg, NULL, 0, 0))
   {
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
   }
  return Msg.wParam;

 }

LRESULT CALLBACK TrackbarWndProc (HWND hWnd, UINT Message,
                                  UINT wParam, LONG lParam )
 {
  switch(Message)
   {
    case WM_COMMAND:
     switch(wParam)
      {
       case IDM_Exit:
        SendMessage(hWnd, WM_CLOSE, 0, 0);
```

```
            break;
          case IDM_Dialog:
            DialogBox(hInst, "TrackbarDialog", hWnd, TrackbarDialogProc);
            break;
        }
       return 0;
      case WM_DESTROY:
       PostQuitMessage(0);
       return 0;
    }
   return DefWindowProc(hWnd,Message,wParam, lParam);
}

BOOL CALLBACK TrackbarDialogProc(HWND hDlg, UINT Message,
                                 WPARAM wParam, LPARAM lParam)
{
  static HWND hEditWnd;
  static HWND hSpinWnd;
  static HWND hTrackbarWnd;
  switch(Message)
   {
     case WM_INITDIALOG:
      hEditWnd = GetDlgItem(hDlg, ID_Edit);
      hTrackbarWnd = GetDlgItem(hDlg, ID_Trackbar);
      hSpinWnd = CreateUpDownControl(WS_CHILD | WS_BORDER |
                                     WS_VISIBLE | UDS_SETBUDDYINT |
                                     UDS_ALIGNRIGHT,
                                      0, 12, 50, 50,
                                      hDlg, ID_Spin, hInst, hEditWnd,
                                      10, 0, 5);
      hTrackbarWnd = CreateWindow(TRACKBAR_CLASS,"Trackbar Demo",
                                  WS_CHILD | WS_VISIBLE | WS_TABSTOP |
                                  TBS_AUTOTICKS,
                                  4, 75, 142, 40,
                                  hDlg, NULL, hInst, NULL);
      SendMessage(hTrackbarWnd, TBM_SETRANGE, TRUE,
                  MAKELONG(0,10));
      SendMessage(hTrackbarWnd, TBM_SETPOS, TRUE, 5);
      return TRUE;
     case WM_VSCROLL:
      SendMessage(hTrackbarWnd, TBM_SETPOS, TRUE,
                  GetDlgItemInt(hDlg, ID_Edit, NULL,1));
      return TRUE;
     case WM_HSCROLL:
      SetDlgItemInt(hDlg, ID_Edit, SendMessage(hTrackbarWnd,
                    TBM_GETPOS, 0,0), TRUE);
     case WM_COMMAND:
      switch(LOWORD(wParam))
       {
         case ID_OK:
```

```
            EndDialog(hDlg,0);
            return TRUE;
          }
        break;
    }
  return FALSE;
}
```

После того, как эта программа будет запущена и в основном меню программы будет выбран элемент «Dialog», на экране появится диалоговое окно, вид которого показан на рис. 14.

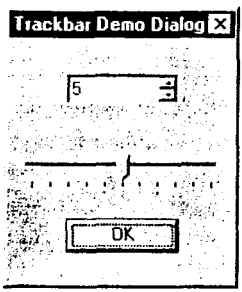


Рис. 14. Диалоговое окно со спином и трекбаром

Попробуйте изменить положение спина. При этом изменится положение слайдера на трекбаре. Аналогично, если изменить положение слайдера с помощью мыши, то изменится значение в окне редактирования, которое является приятельским окном спина.

При написании этой программы использовались две новые функции. Первая, GetDlgItemInt(), описана в файле winuser.h следующим образом:

```
WINUSERAPI UINT WINAPI GetDlgItemInt(HWND hDlg,    int nIDDlgItem,
                                     BOOL *lpTranslated,
                                     BOOL bSigned);
```

Эта функция берет число, записанное в окне редактирования (оно представлено в виде строки), преобразует его в числовой вид и возвращает числовое значение. В программе мы используем эту функцию для того чтобы считать значение спина и соответствующим образом изменить положение слайдера трекбара.

Вторая функция, описанная в том же файле, имеет следующий прототип:

```
WINUSERAPI BOOL WINAPI SetDlgItemTextA(HWND hDlg, int nIDDlgItem,
                                       LPCSTR lpString);
WINUSERAPI BOOL WINAPI SetDlgItemTextW(HWND hDlg, int nIDDlgItem,
```

```
                                        LPCWSTR lpString);
#ifdef UNICODE
#define SetDlgItemText  SetDlgItemTextW
#else
#define SetDlgItemText  SetDlgItemTextA
#endif // !UNICODE
```

Она производит действие, обратное GetDlgItemInt(), т. е. получает в качестве аргумента целое число и возвращает его представление в виде строки. В программе она используется для того, чтобы в окне редактирования отобразить номер позиции слайдера. Все просто, не так ли?

РАБОТА С ИНДИКАТОРОМ (PROGRESS BAR'OM)

Надеюсь, что читатель уже имеет опыт инсталляции программных продуктов для Windows. Там степень завершенности задачи отражается синей полосой, которая постоянно растет. По достижении ею отметки, соответствующей 100%, процесс инсталляции оказывается завершенным. Вот эта синяя полоса и является индикатором, который индицирует степень завершенности достаточно длительной задачи.

По своему внешнему виду это, наверное, самый простой из общих элементов управления. То же самое можно сказать и о работе с ним. У него только один стиль, и всего пять сообщений применяются для управления им.

Как и в случае трекбара, специальной функции для создания индикатора нет. Для создания индикатора необходимо использовать функцию CreateWindow() или CreateWindowEx(). Для указания имени класса необходимо использовать макрос PROGRESS_CLASS, который в файле commctrl.h описан следующим образом:

```
#ifdef _WIN32

#define PROGRESS_CLASSA     "msctls_progress32"
#define PROGRESS_CLASSW     L"msctls_progress32"

#ifdef UNICODE
#define  PROGRESS_CLASS     PROGRESS_CLASSW
#else
#define  PROGRESS_CLASS     PROGRESS_CLASSA
#endif

#else
#define PROGRESS_CLASS      "msctls_progress"
#endif
```

Для управления индикатором используются сообщения. Поговорим о них.

Наверное, для того чтобы использовать индикатор, необходимо определить для него минимальное и максимальное значения (в приведенном примере, когда я говорил об инсталляционных программах, минимальное и максимальное значения равны 0 и 100 соответственно). С этой целью используется сообщение PBM_SETRANGE (WM_USER+1). wParam его должен быть равно 0, а lParam должно определяться как MAKELONG(wMinRange, wMaxRange), где wMinRange и wMaxRange - минимальное и максимальное значения. Функция, пославшая это сообщение, возвращает значения старого диапазона. Если обозначить возвращаемое значение как lReturn, то LOWORD(lReturn) будет определять нижнюю границу диапазона, а HIWORD(lReturn) будет содержать верхнюю границу диапазона.

Для установки индикатора в определенную позицию (другими словами, для определения необходимой длины индикатора) используется сообщение PBM_SETPOS (WM_USER + 2). При этом wParam должен содержать позицию, в которую необходимо установить индикатор. lParam должен быть равным 0. Предыдущая позиция индикатора возвращается функцией, пославшей сообщение.

Сообщение PBM_DELTAPOS применяется для определения значения, на которое будет увеличена длина индикатора. wParam этого сообщения определяет приращение, а lParam должен быть равным 0. Функция, пославшая сообщение, возвращает значение предыдущей позиции.

Для определения шага, с которым будет увеличиваться длина индикатора, используется сообщение PBM_SETSTEP, wParam которого определяет шаг, а lParam должно быть равно нулю. По умолчанию, шаг приращения равен 10.

Сообщение PBM_STEPIT указывает индикатору, что необходимо осуществить увеличение длины, используя при этом все текущие установки, т. е. текущую позицию и текущий шаг. Оба параметра сообщения должны быть равны 0.

И, как всегда, демонстрационная программа, с помощью которой читатель сможет увидеть управление индикатором в действии.

Вся программа состоит из трех файлов. Первый - файл заголовков:

```
#define IDC_MSCTLS_TRACKBAR1    101
#define IDM_Exit                101
#define IDM_Dialog              102
#define IDM_About               103
#define ID_OK                   104
#define ID_Edit                 105
#define ID_Spin                 106
```

```
#define ID_ProgressBar                107
```
Второй файл - файл ресурсов, он также приводится ниже.
```
#include "ProgressBar.h"

ProgressBarMenu MENU
{
 POPUP "&File"
 {
  MENUITEM "E&xit", IDM_Exit
 }

 MENUITEM "&Dialog", IDM_Dialog
 POPUP "&Help"
 {
  MENUITEM "&About", IDM_About
 }

}
ProgressBarDialog DIALOG 0, 0, 100, 100
STYLE DS_MODALFRAME | DS_3DLOOK | WS_POPUP | WS_VISIBLE |
        WS_CAPTION | WS_SYSMENU
CAPTION "Progressbar Demo Dialog"
FONT 8, "MS Sans Serif"
{
 DEFPUSHBUTTON "OK", ID_OK, 25, 73, 50, 14
 CONTROL "", ID_Edit, "edit", ES_LEFT | WS_CHILD | WS_VISIBLE |
              WS_BORDER | WS_TABSTOP | ES_NUMBER, 25, 14, 50, 12
}
```
И, естественно, основной файл программы.
```
#include <windows.h>
#include <commctrl.h>
#include "ProgressBar.h"

HINSTANCE hInst;

LRESULT CALLBACK ProgressBarWndProc ( HWND, UINT, UINT, LONG );
BOOL CALLBACK ProgressBarDialogProc(HWND, UINT, WPARAM,
                                                LPARAM);

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                        LPSTR lpszCmdParam, int nCmdShow )

 {
  HWND hWnd ;
  WNDCLASS WndClass ;
  MSG Msg;
  char szClassName[] = "ProgressBarDemo";

  hInst = hInstance;
```

```
/* Registering our window class */
/* Fill WNDCLASS structure */
   WndClass.style = CS_HREDRAW | CS_VREDRAW;
   WndClass.lpfnWndProc = ProgressBarWndProc;
   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 = "ProgressBarMenu";
   WndClass.lpszClassName = szClassName;

   if ( !RegisterClass(&WndClass) )
      {
       MessageBox(NULL,"Cannot register class","Error",MB_OK);
       return 0;
      }
   hWnd = CreateWindow(szClassName, "Progressbar Demonstration Program",
                       WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                       CW_USEDEFAULT, NULL, NULL,
                       hInstance,NULL);
   if(!hWnd)
      {
       MessageBox(NULL,"Cannot create window","Error",MB_OK);
       return 0;
      }

   /* Show our window */
   ShowWindow(hWnd,nCmdShow);
   UpdateWindow(hWnd);

/* Beginning of messages cycle */
   while(GetMessage(&Msg, NULL, 0, 0))
    {
     TranslateMessage(&Msg);
     DispatchMessage(&Msg);
    }
   return Msg.wParam;
 }

LRESULT CALLBACK ProgressBarWndProc (HWND hWnd, UINT Message,
                                     UINT wParam, LONG lParam )
 {
  switch(Message)
   {
    case WM_COMMAND:
     switch(LOWORD(wParam))
      {
```

```
        case IDM_Dialog:
          DialogBox(hInst, "ProgressBarDialog", hWnd, ProgressBarDialogProc);
          break;
        case IDM_Exit:
          SendMessage(hWnd, WM_CLOSE, 0,0);
          break;
       }
      return 0;
    case WM_DESTROY:
      PostQuitMessage(0);
      return 0;
   }
  return DefWindowProc(hWnd,Message,wParam, lParam);
 }

BOOL CALLBACK ProgressBarDialogProc(HWND hDlg, UINT Message,
                                    WPARAM wParam, LPARAM lParam)
 {
  static HWND hEditWnd;
  static HWND hSpinWnd;
  static HWND hProgressBarWnd;
  int i;
  switch(Message)
   {
     case WM_INITDIALOG:
      hEditWnd = GetDlgItem(hDlg, ID_Edit);
      hSpinWnd = CreateUpDownControl(WS_CHILD | WS_BORDER |
                                     WS_VISIBLE |
                                     UDS_SETBUDDYINT |
                                     UDS_ALIGNRIGHT,
                                      0, 12, 50, 50,
                                     hDlg, ID_Spin,
                                     hInst,
                                     hEditWnd,
                                     10, 0, 5);
      hProgressBarWnd = CreateWindow(PROGRESS_CLASS,
                                    "ProgressBar Demo",
                                     WS_CHILD | WS_VISIBLE,
                                     10, 75, 130, 20,
                                     hDlg, NULL, hInst, NULL);
      SendMessage(hProgressBarWnd, PBM_SETRANGE, 0,
                  MAKELPARAM(0,10));
      SendMessage(hProgressBarWnd, PBM_SETSTEP, 1, 0);
      SendMessage(hProgressBarWnd, PBM_SETPOS, 5, 0);
      return TRUE;
     case WM_VSCROLL:
      SendMessage(hProgressBarWnd, PBM_SETPOS, GetDlgItemInt(hDlg,
                  ID_Edit, NULL,1), 0);
      return TRUE;
     case WM_COMMAND:
```

```
    switch(LOWORD(wParam))
     {
       case ID_OK:
        EndDialog(hDlg,0);
        return TRUE;
     }
    break;
  }
 return FALSE;
}
```

Вид диалогового окна, возникающего после выбора пользователем элемента «Dialog» в главном меню программы, показан на рис. 15.

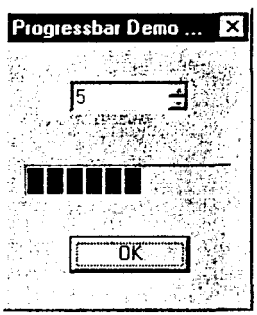


Рис. 15. Диалоговое окно со спином и индикатором

Эта программа по своему действию очень похожа на приведенную в предыдущем разделе. Разница состоит в том, что спин управляет не трекбаром, а индикатором. Рекомендую читателю нажать несколько раз кнопки спина для того чтобы позиция спина изменилась, и посмотреть, что произойдет с индикатором. При разборе программы особое внимание следует уделить обработке сообщений WM_INITDIALOG и WM_VSCROLL.

Далее рассмотрим окна подсказок (Tooltip Controls) и списки изображений (ImageLists). При изучении Win32 я не увидел тех моментов, когда два этих элемента управления использовались бы самостоятельно. Они являются только вспомогательными элементами.

РАБОТА С ОКНАМИ ПОДСКАЗОК

Окна подсказок - это небольшие всплывающие окна, которые содержат одну строку текста, объясняющую назначение какого-либо инструмента (tool) родительского окна. Под инструментом в данном случае понимается либо элемент управления, присутствующий в родительском

окне (пример - полоса инструментов в WinWord'e), либо прямоугольная область внутри рабочей области окна.

Окна подсказок почти всегда скрыты. Отображаются они только в том случае, если пользователь задерживает курсор мыши на инструменте. Подсказка появляется рядом с инструментом и исчезает тогда, когда пользователь либо нажимает клавишу (мыши или клавиатуры), либо убирает курсор с инструмента. Одна подсказка может появляться при задержке курсора на разных инструментах.

К сожалению, не все общие элементы управления имеют специальную функцию для своего создания (☺). Окна подсказок тоже создаются только посредством применения CreateWindow() или CreateWindowEx(). В этом случае для их создания необходимо использовать макрос TOOLTIPS_CLASS, который в файле commctrl.h описан следующим образом:

```
#ifdef _WIN32

#define TOOLTIPS_CLASSW     L"tooltips_class32"
#define TOOLTIPS_CLASSA     "tooltips_class32"

#ifdef UNICODE
#define TOOLTIPS_CLASS      TOOLTIPS_CLASSW
#else
#define TOOLTIPS_CLASS      TOOLTIPS_CLASSA
#endif

#else
#define TOOLTIPS_CLASS      "tooltips_class"
#endif
```

При создании подсказки могут использоваться два стиля, специально разработанные для окон этого типа - TTS_ALWAYSTIP и TTS_NOPREFIX. Подсказка, имеющая стиль TTS_ALWAYSTIP, появляется при помещении курсора на инструмент вне зависимости от того, активно или не активно родительское окно.

Необходимо отметить еще одну возможность создания окон подсказок. Дело в том, что в Win32 некоторые элементы управления имеют специальный стиль, обычно оканчивающийся на _TOOLTIPS. Он позволяет программе не создавать собственные окна подсказок, а использовать встроенные возможности системы. Разумеется, этот способ использования подсказок намного проще.

Достаточно часто подсказки используются в панели инструментов, при этом каждая кнопка в панели инструментов соответствует элементу

меню. При этом подсказки, как правило, совпадают с текстом, отображаемым в соответствующем элементе меню. Если окно подсказки создается со стилем TTS_NOPREFIX, то система автоматически удаляет знак амперсанта, если он присутствует в строке меню.

Помимо этого, подсказка сама по себе может быть активной и неактивной. Активная подсказка появляется при нахождении курсора на инструменте, неактивная подсказка не отображается ни в каких случаях.

Таблица 38. Сообщения, посылаемые "подсказкам"

Сообщение	Значение	Описание
TTM_ACTIVATE	WM_USER + 1	Подсказка делается активной или неактивной, wParam = TRUE - подсказка активна, lParam всегда = 0
TTM_SETDELAYTIME	WM_USER + 3	Определяются интервал инициализации, интервал отображения и интервал повторного отображения
TTM_ADDTOOL		Регистрирует инструмент, с которым будет работать подсказка
TTM_DELTOOL		Удаляет ранее добавленный инструмент
TTM_NEWTOOLRECT		Определяет границы окна подсказки
TTM_RELAYEVENT	WM_USER + 7	Передает сообщение от мыши окну подсказки для обработки
TTM_GETTOOLINFO		Запрос информации об инструменте, с которым работает подсказка
TTM_SETTOOLINFO		Установка информации для инструмента
TTM_HITTEST		Определение, попадает ли точка в указанный для инструмента прямоугольник, и, если попадает, получение информации об инструменте
TTM_GETTEXT		Получение текста подсказки, отображаемой с инструментом
TTM_UPDATETIPTEXT		Установка текста подсказки для инструмента
TTM_GETTOOLCOUNT	WM_USER + 13	Получение числа инструментов, с которыми работает подсказка
TTM_ENUMTOOLS		Позволяет программе последовательно перебрать все инструменты, с которыми работает окно подсказки
TTM_GETCURRENTTOOL		Получение информации о текущем инструменте
TTM_WINDOWFROMPOINT	WM_USER + 16	Выдача окна подсказки в месте, определяемом не положением курсора, а параметрами сообщения

Так как подсказка является окном, то для управления ею используются сообщения. Их список сообщений с кратким описанием приведен в табл. 38. Рассмотрим наиболее часто используемые сообщения.

Для того чтобы подсказка работала с тем или иным инструментом, необходимо этот инструмент включить в список инструментов, с которыми работает окно подсказки. Для этого окну подсказки надо направить сообщение TTM_ADDTOOL. WParam этого сообщения должен быть равен 0, а lParam содержать указатель на структуру типа TOOLINFO. Эта структура описана в файле commctrl.h :

```
typedef struct tagTOOLINFOA {
  UINT cbSize;
  UINT uFlags;
  HWND hwnd;
  UINT uId;
  RECT rect;
  HINSTANCE hinst;
  LPSTR lpszText;
} TOOLINFOA, NEAR *PTOOLINFOA, FAR *LPTOOLINFOA;

typedef struct tagTOOLINFOW {
  UINT cbSize;
  UINT uFlags;
  HWND hwnd;
  UINT uId;
  RECT rect;
  HINSTANCE hinst;
  LPWSTR lpszText;
} TOOLINFOW, NEAR *PTOOLINFOW, FAR *LPTOOLINFOW;

#ifdef UNICODE
#define TOOLINFO          TOOLINFOW
#define PTOOLINFO         PTOOLINFOW
#define LPTOOLINFO        LPTOOLINFOW
#else
#define TOOLINFO          TOOLINFOA
#define PTOOLINFO         PTOOLINFOA
#define LPTOOLINFO        LPTOOLINFOA
#endif
```

Первое поле этой структуры - cbSize - должно содержать размер в байтах структуры типа TOOLINFO. Сам факт присутствия поля, содержащего такую информацию, говорит о том, что фирма Microsoft не исключает возможности изменения и/или дополнения этой структуры.

Таблица 39. Битовый файлы, определяющие вид и поведение подсказки

Флаг	Значение	Описание
TTF_IDISHWND	0x01	Флаг установлен - поле uId содержит хэндл инструмента, иначе - идентификатор инструмента
TTF_CENTERTIP	0x02	Центрирует подсказку под инструментом
TTF_RTLREADING	0x04	Отображает текст справа налево, как в арабском языке
TTF_SUBCLASS	0x10	Подсказка должна перехватывать сообщения WM_MOUSEMOVE, адресованные инструменту

Следующее поле - uFlags - содержит флаги, определяющие внешний вид и поведение подсказки, а также представление информации в других полях этой структуры. Возможные флаги приведены в табл. 39.

Поле hwnd определяет родительское окно инструмента.

Поле uId обычно содержит идентификатор инструмента. Если поле uFlags включает TTF_IDISHWND, то поле uId содержит хэндл окна, внутри которого находится область, используемая в качестве инструмента.

Следующее поле - rect - определяет координаты окна инструмента относительно левого верхнего угла клиентской области окна, определяемого полем hwnd. Если поле uFlags включает флаг TTF_IDISHWND, поле rect игнорируется.

В поле hinst хранится хэндл экземпляра программы, которая содержит строковый ресурс, определяющий текст подсказки. Если это поле не равно нулю, то поле lpszText содержит идентификатор строкового ресурса.

Как, надеюсь, читатель уже понял, поле lpszText может интерпретироваться по-разному. Итак, вариант первый. Если значение поля lpszText равно LPSTR_TEXTCALLBACK, то именно окно, хэндл которого указан в поле hwnd, получает нотификационное сообщение TTN_NEEDTEXT, уведомляющее о том, что родительское окно инструмента должно определить, какой текст подсказки должен быть отображен. Второй вариант - поле содержит идентификатор строкового ресурса, в котором определен текст сообщения. Этот вариант используется тогда, когда поле hinst не равно 0. Кроме этого, признаком того, что поле содержит идентификатор строкового ресурса, является старшее нулевое слово. И, наконец, третье, наиболее часто использующееся поле, содержит указатель на строку, содержащую текст подсказки.

Таким образом, мы видим, что структура типа TOOLINFO полностью определяет, что используется в качестве инструмента - элемент управления или область экрана, а также где находится текст подсказки - в ресурсах, в строке программы или он определяется родительским окном инструмента.

В любой момент программа может изменить текст подсказки. Для этого окну подсказки посылается сообщение TTM_UPDATETIPTEXT, wParam которого должен быть равным 0, а lParam - указывать на структуру типа TOOLINFO.

Программа может получить текст, который используется для выдачи подсказки об инструменте с помощью посылки окну подсказки сообщения TTM_GETTEXT. В этом сообщении wParam должен быть равным 0. lParam, как и в предыдущем случае, должен содержать указатель на структуру типа TOOLINFO, в которой определяется инструмент, подсказка о котором запрашивается. Поле lpszText указывает на буфер, в который будет записан текст подсказки.

Для того чтобы отобразиться, окно подсказки должно получить сообщение от мыши. Так как Windows посылает сообщения только тому окну, поверх которого находится курсор, программа должна использовать сообщение TTM_RELAYEVENT для того чтобы транслировать сообщение окну подсказки. wParam этого сообщения должен быть равным нулю, а lParam должен содержать указатель на структуру типа MSG, в которой хранится информация о транслируемом сообщении. При этом необходимо учесть, что окно подсказки обрабатывает информацию только о сообщениях, приведенных ниже:

- WM_LBUTTONDOWN;
- WM_LBUTTONUP;
- WM_MBUTTONDOWN;
- WM_MBUTTONUP;
- WM_MOUSEMOVE;
- WM_RBUTTONDOWN;
- WM_RBUTTONUP.

Если инструмент представляет собой прямоугольную часть окна, то тогда никаких сложностей не появляется. Если же инструмент является системным окном (таким, например, как кнопка), то в этом случае возникают определенные трудности. Программа должна будет или перехватывать сообщения посредством использования hook'ов, или подменить оконную функцию системного окна (осуществить subclassing). К сожалению, рассмотрение вопросов, связанных с hook'ами и subclassing'ом выходит за рамки этой книги, поэтому я вынужден буду остановиться только на подсказках, связанных с областью окна. Уважаемый читатель!

Предлагаю вам изучить вопросы о перехвате сообщений и подмене оконных функций самостоятельно.

Когда окно подсказки получает транслированное сообщение WM_MOUSEMOVE, оно определяет, находится ли курсор в области, к которой привязана подсказка. При положительном ответе окно подсказки устанавливает таймер. В конце интервала окно снова проверяет, находится ли курсор в нужной области. При втором положительном ответе формируется текст подсказки, копируется в окно подсказки и окно выдается на отображение. Подсказка отображается до тех пор, пока окно подсказки не получит сообщения о нажатии или отжатии клавиши мыши или о перемещении курсора за пределы интересующей области.

При работе окно подсказки использует три определенных временных интервала. Первый из них, называемый интервалом инициализации, определяет период, в течение которого курсор должен находиться в пределах интересующей области для того, чтобы отобразилась закладка. Второй - интервал повторного отображения - определяет задержку между последовательными отображениями окна подсказки в тех случаях, когда курсор скользит по инструментам, например, по панели инструментов. Третий интервал - интервал отображения - определяет время, в течение которого подсказка находится на отображении в тех случаях, когда курсор находится внутри интересующей области или в пределах границы инструмента. Все эти периоды могут быть определены с помощью сообщения TTM_SETDELAYTIME. wParam этого сообщения определяет, какой интервал устанавливается. lParam определяет длительность интервала в миллисекундах. Допустимые значения wParam приведены в табл. 40.

В том случае, когда инструментом является область окна, размеры и/или положение которой изменились, окну подсказки необходимо послать сообщение TTM_NEWTOOLRECT для того, чтобы подсказка появлялась в нужном месте. wParam этого сообщения всегда равен 0.

Таблица 40. Идентификаторы временных интервалов

Параметр	Значение	Описание
TTDT_AUTOMATIC	0	Все интервалы вычисляются автоматически на основе lParam
TTDT_RESHOW	1	Определяется интервал повторного отображения
TTDT_AUTOPOP	2	Определяется интервал отображения
TTDT_INITIAL	3	Определяется интервал инициализации

lParam этого сообщения должен указывать на структуру типа TOOLINFO, поля hwnd и uId которой должны определять инструмент, а поле rect - новые границы инструмента. В том случае, когда инструмент реализован как окно, информировать подсказку о его изменении не нужно, так как подсказка определит факт нахождения курсора в границах инструмента по хэндлу окна.

Перед своим отображением окно подсказки посылает родительскому окну нотификационное сообщение TTN_SHOW, а перед скрытием - TTN_POP. В данном случае нотификационные сообщения посылаются с помощью сообщения WM_NOTIFY.

Для получения информации об инструменте программа может использовать сообщения TTM_GETCURRENTTOOL и TTM_GETTOOLINFO. Изменить информацию об инструменте можно с помощью сообщения TTM_SETTOOLINFO. Если программе требуется, чтобы подсказка с данным инструментом больше не работала, окну подсказки нужно направить сообщение TTM_DELTOOL. Параметры этих сообщений однотипны и ничего сложного в них нет. Рекомендую читателю изучить работу этих сообщений самостоятельно.

Как уже было сказано, сами по себе окна подсказок не используются, поэтому демонстрационной программы я не приведу. Тем не менее, в разделе, посвященном закладкам, будут даны примеры другого способа использования подсказок, применимого, к сожалению, только к общим элементам управления, которые появились в Windows NT и Windows'95. В чем состоит этот способ? Дело в том, что при создании некоторых элементов управления можно указать стиль, позволяющий этим окнам реагировать на сообщения WM_NOTIFY с нотификационным кодом TTN_NEEDTEXT. Скажем, для закладок этот стиль называется TCS_TOOLTIPS, для панели инструментов - TBS_TOOLTIPS и т. д. В этих случаях родительскому окну элемента управления в качестве lParam сообщения WM_NOTIFY передается указатель на структуру типа TOOLTIPTEXT, описание которой, находящееся в файле commctrl.h имеет следующий вид:

```
typedef struct tagTOOLTIPTEXTA {
  NMHDR hdr;
  LPSTR lpszText;
  char szText[80];
  HINSTANCE hinst;
  UINT uFlags;
} TOOLTIPTEXTA, FAR *LPTOOLTIPTEXTA;
```

```
typedef struct tagTOOLTIPTEXTW {
  NMHDR hdr;
  LPWSTR lpszText;
  WCHAR szText[80];
  HINSTANCE hinst;
  UINT uFlags;
} TOOLTIPTEXTW, FAR *LPTOOLTIPTEXTW;


#ifdef UNICODE
#define TOOLTIPTEXT        TOOLTIPTEXTW
#define LPTOOLTIPTEXT      LPTOOLTIPTEXTW
#else
#define TOOLTIPTEXT        TOOLTIPTEXTA
#define LPTOOLTIPTEXT      LPTOOLTIPTEXTA
#endif
```

Первое поле этой структуры, тоже структура, но типа NMHDR, описана в файле winuser.h так:

```
typedef struct tagNMHDR
{
  HWND  hwndFrom;
  UINT  idFrom;
  UINT  code;       // NM_ code
}  NMHDR;
typedef NMHDR FAR * LPNMHDR;
```

hwndFrom - хэндл элемента, пославшего нотификационное сообщение, idFrom - идентификатор этого элемента, code - код нотификационного сообщения. Вроде бы все ясно.

Второе поле стуктуры типа TOOLTIPTEXT (как бы не запутаться с этими структурами!) - lpszText - может содержать указатель на строку, выдаваемую в качестве подсказки, или, если поле hinst не равно 0, содержит идентификатор строкового ресурса, определяющего текст выдаваемой подсказки.

Вместо того чтобы определять указатель на строку, можно скопировать эту строку в буфер szText, который является третьим полем структуры типа TOOLTIPTEXT.

Поле hinst является хэндлом экземпляра, содержащего строковый ресурс с текстом подсказки. Если lpszText является указателем на строку подсказки, это поле должно быть равным NULL.

И наконец, последнее поле - uFlags - содержит комбинацию флагов TTF_IDISHWND и TTF_RTLREADING, которые были рассмотрены ранее.

Анализируя поля этих двух структур, можно определить элемент, для которого определяется подсказка. Для того чтобы эта подсказка появилась на экране, достаточно определить либо указатель на строку подсказки, либо ее идентификатор в таблице строк (не забыть при этом о поле hinst!), либо скопировать эту строку в предлагаемый буфер. И все! Пример подобного использования подсказок приведен в разделе о работе с закладками.

РАБОТА СО СПИСКОМ ИЗОБРАЖЕНИЙ

В Windows предусмотрен интересный элемент, который лично я могу назвать элементом управления с большой натяжкой. Тем не менее, этот элемент активно используется при работе с другими элементами управления, например, с закладками, речь о которых еще впереди. Я имею в виду список изображений (Image List). Он представляет собой коллекцию изображений одинакового размера, к каждому из которых можно осуществить доступ по его индексу. Список изображений используется для эффективного управления и манипулирования большими наборами изображений.

Наверное, этот элемент управления самостоятельно не используется еще и потому, что он не является окном. Список изображений - это всего-навсего структура в памяти, обеспечивающая простой доступ к изображениям.

Как и в случае обычных списков, при работе со списками изображений можно создавать и удалять списки, добавлять, удалять и изменять элементы списков. Специфическим именно для списков изображений являются операции по прорисовке и перетаскиванию изображений.

Списки изображений могут быть немаскированными и маскированными. Немаскированный список представляет собой один большой цветной bitmap, который, в свою очередь, состоит из одного или нескольких изображений. Маскируемый список состоит из двух больших bitmap'ов, первый из которых, цветной, содержит непосредственно список изображений, а второй, монохромный, содержит список масок - по одной для каждого элемента.

При прорисовке немаскируемого изображения оно просто копируется на соответствующее устройство. В случае прорисовки маскируемого изображения биты изображения комбинируются с битами маски, создавая обычно прозрачные области, сквозь которые видно то изображение, которое было на устройстве до прорисовки.

Для создания списка изображений приложению необходимо вызвать функцию ImageList_Create(), которая описана в файле commctrl.h следующим образом:

```
WINCOMMCTRLAPI HIMAGELIST  WINAPI ImageList_Create(int cx, int cy,
                                                   UINT flags,
                                                   int cInitial, int cGrow);
```

В этой функции первые два аргумента, cx и cy, определяют размер в пикселах каждого изображения. Третий аргумент, flags, указывает тип списка изображений. Для каждого типа в файле commctrl.h предусмотрен макрос, начинающийся с ILC_. Список этих типов приведен в табл. 41.

Размер создаваемого bitmap'а вычисляется, исходя из значения четвертого аргумента. Этот аргумент определяет, сколько изображений должен включать bitmap. Если на каком-то этапе число изображений, включенных в bitmap, достигнет предельного значения, то система автоматически расширит bitmap, добавив место для хранения еще определенного числа изображений, которое определяется последним параметром функции - cGrow.

Таблица 41. Флаги, используемые при создании списка изображений

Флаг	Значение	Описание
ILC_COLOR	0x0000	Используется флаг по умолчанию, обычно ILC_COLOR4, для старых драйверов - ILC_COLORDDB
ILC_MASK	0x0001	Создается маскированный bitmap
ILC_COLORDDB	0x00FE	Используется bitmap, зависящий от устройства (устаревший формат)
ILC_COLOR4	0x0004	В качестве bitmap'а, содержащего изображения, используется 16-цветный bitmap
ILC_COLOR8	0x0008	В качестве bitmap'а, содержащего изображения, используется 256-цветный bitmap
ILC_COLOR16	0x0010	В качестве bitmap'а, содержащего изображения, используется bitmap, допускающий одновременное использование до 65536 цветов
ILC_COLOR24	0x0018	В качестве bitmap'а, содержащего изображения, используется bitmap, допускающий до 2**24 цветов
ILC_COLOR32	0x0020	В качестве bitmap'а, содержащего изображения, используется bitmap, допускающий до 2**32 цветов
ILC_PALETTE	0x0800	Со списком изображений используется цветовая палитра

При успешном выполнении функция возвращает хэндл созданного списка. NULL должен заставить программиста вздохнуть и найти ошибку.

Но что же происходит при создании списка изображений? Помимо создания непосредственно bitmap'а с указанными характеристиками, функция создает контекст, совместимый с экраном, и выбирает созданный bitmap в качестве текущего для этого контекста. В случае маскированного bitmap'а функция создает два экранно-совместимых контекста, при этом для одного в качестве текущего она выбирает bitmap с изображениями, а для другого - bitmap с масками.

Естественно, что при использовании такой техники резко уменьшается время, необходимое для копирования изображения на экран. Как результат - программы, использующие список изображений, могут работать значительно быстрее тех, которые перед использованием вынуждены загружать изображения из ресурсов.

Для того чтобы удалить список изображений из памяти (отдельные изображения, из которых состоит список, остаются на своих местах, уничтожаются только указатели, что, собственно, и делает набор изображений списком), необходимо вызвать функцию ImageList_Destroy(), которая описана следующим образом:

```
WINCOMMCTRLAPI BOOL WINAPI ImageList_Destroy(HIMAGELIST himl);
```

Единственным аргументом этой функции является хэндл удаляемого списка изображений. Возвращенное этой функцией значение FALSE говорит о том, что при удалении списка произошла какая-то ошибка, и список из памяти не удален.

Итак, надеюсь, читатель понял, что такое список изображений и для чего он служит. Теперь возникают очередные вопросы: как добавлять изображения в список, удалять их и производить манипуляции с ними?

Для того чтобы добавить изображение в список, необходимо воспользоваться функцией ImageList_Add(), описание которой имеет следующий вид:

```
WINCOMMCTRLAPI int WINAPI ImageList_Add(HIMAGELIST himl,
                                        HBITMAP hbmImage,
                                        HBITMAP hbmMask);
```

Первый аргумент этой функции - himl - очевиден: хэндл списка изображений. Второй аргумент - hbmImage - представляет собо хэндл добавляемого в список изображения. Третий аргумент - hbmMask - хэндл

монохромного bitmap'а, который содержит маски. В случае немаскированного списка третий аргумент игнорируется.

Описание функции ImageList_AddMasked() приведено ниже:

```
WINCOMMCTRLAPI int WINAPI ImageList_AddMasked(HIMAGELIST himl,
                                              HBITMAP hbmImage,
                                              COLORREF crMask);
```

Эта функция действует почти так же, как и предыдущая, но маска генерируется автоматически. Для генерации маски необходимо задать цвет. Если в изображении встречается пиксель указанного цвета, то цвет этого пикселя заменяется на черный, а соответствующий бит маски - на 0. В результате, при прорисовке изображения пиксели указанного цвета становятся прозрачными. Аргументы этой функции также очевидны. Первый - хэндл списка изображений, второй - хэндл включаемого в список изображения, третий - цвет пикселей, которые необходимо сделать прозрачными при прорисовке.

Для добавления в список иконки или курсора используется макрос ImageList_AddIcon(), первым аргументом которого необходимо указать хэндл списка изображений, а вторым - хэндл добавляемой иконки или добавляемого курсора. Макрос возвращает индекс добавленного изображения.

При необходимости программа может создать новую иконку или курсор, используя изображение и маску из списка изображений. Для этой цели необходимо использовать функцию ImageList_GetIcon(). Её описание приведено ниже:

```
WINCOMMCTRLAPI HICON WINAPI ImageList_GetIcon(HIMAGELIST himl,
                                              int i, UINT flags);
```

Первый аргумент этой функции - хэндл списка изображений. Второй - индекс изображения, на основе которого будет создана иконка или курсор. Третий аргумент - флаги прорисовки, которые можно найти в таблице, приведенной при описании функции ImageList_Draw().

Функция возвращает хэндл созданной иконки или курсора.

К этому моменту мы научились добавлять изображения в список. А для удаления изображения нужно вызвать функцию ImageList_Remove(), описанную так:

```
WINCOMMCTRLAPI BOOL WINAPI ImageList_Remove(HIMAGELIST himl,
                                            int i);
```

Автор уверен, что даже не заглядывая дальше, читатель догадался, что первым аргументом является хэндл списка изображений, а вторым - индекс удаляемого изображения. Если вместо индекса изображения подставить -1, то функция удалит все изображения из списка, но не удалит сам список. Для удаления всех изображений из списка можно воспользоваться макросом ImageList_RemoveAll(), единственным аргументом которого является хэндл списка изображений.

Для замены изображения в списке служит функция ImageList_Replace(). Её описание находим в файле commctrl.h:

```
WINCOMMCTRLAPI BOOL WINAPI ImageList_Replace(HIMAGELIST himl,
                                             int i,
                                             HBITMAP hbmImage,
                                             HBITMAP hbmMask);
```

Аргументы этой функции вполне понятны: первый - хэндл списка изображений; второй - индекс замещаемого изображения; третий и четвертый - хэндлы нового изображения и его маски. Если список немаскированный, четвертый аргумент игнорируется.

Очередная функция - ImageList_ReplaceIcon() - описана так:

```
WINCOMMCTRLAPI int WINAPI ImageList_ReplaceIcon(HIMAGELIST himl,
                                                int i,
                                                HICON hicon);
```

Нужно ли описывать аргументы этой функции?

Если в функции ImageList_ReplaceIcon() второй аргумент заменить на -1, то иконка или курсор будут не замещать старое изображение, а добавляться в список. Этот нюанс используется в макросе ImageList_AddIcon(), аргументами которого являются хэндл списка изображений и хэндл добавляемой иконки или курсора.

Вполне вероятно, что если второй аргумент функции ImageList_Replace() равен -1, то изображение не замещает старое, а добавляется. Проверку этой гипотезы я оставляю читателю.

Для того чтобы прорисовать изображение, хранящееся в списке, необходимо использовать функцию ImageList_Draw(). Вполне понятно, что для прорисовки изображения вызывается что-то типа функции BitBlt(), которая требует два контекста устройства, координаты и прочее. Вспомним, что контекст устройства, в котором хранится список изображений, у нас есть. Координаты отдельного изображения моментально вычисляются по его индексу. И что остается? Хэндл контекста, на кото-

рый будет копироваться изображение, координаты в этом контексте и флаги прорисовки.

Функция ImageList_Draw() имеет следующий прототип:

```
WINCOMMCTRLAPI BOOL WINAPI ImageList_Draw(HIMAGELIST himl,
                                          int i,
                                          HDC hdcDst,
                                          int x,
                                          int y,
                                          UINT fStyle);
```

Сравните аргументы этой функции с теми, наличие которых мы вычислили чуть выше. На всякий случай поясню. Первый аргумент - хэндл списка изображений, второй - индекс изображения, третий - хэндл контекста, на который будет копироваться изображение, четвертый и пятый - координаты в этом контексте, начиная с которых будет скопировано изображение, и, наконец, шестой - именно флаги прорисовки. Они приведены в табл. 42.

Привлекательность использования списков изображений повышает еще одно обстоятельство. При использовании списков возможно использование специальных функций, позволяющих пользователю перемещать изображения (drag-and-drop) на экране: во-первых, с минимальными затратами на написание нового кода; во-вторых, без заметного мерцания.

Таблица 42. Флаги прорисовки отдельного изображения в списке изображений

Флаг	Значение	Описание
ILD_NORMAL	0x0000	Обычное копирование изображения
ILD_TRANSPARENT	0x0001	Каждый белый бит маски заставляет соответствующий бит изображения прорисовываться как прозрачный
ILD_BLEND25	0x0002	Снижение интенсивности цветов изображения на 25 %
ILD_BLEND50	0x0004	Снижение интенсивности цветов изображения на 50 %
ILD_MASK	0x0010	Прорисовка маски
ILD_IMAGE	0x0020	Прорисовка изображения
ILD_OVERLAYMASK	0x0F00	
ILD_SELECTED	ILD_BLEND50	
ILD_FOCUS	ILD_BLEND25	
ILD_BLEND	ILD_BLEND50	

Операция перемещения изображения начинается вызовом функции ImageList_BeginDrag(), прототип которой выглядит следующим образом:

```
WINCOMMCTRLAPI BOOL WINAPI ImageList_BeginDrag(IMAGELIST himlTrack,
                                               int iTrack,
                                               int dxHotspot,
                                               int dyHotspot);
```

В число аргументов этой функции входят хэндл списка изображений, индекс перемещаемого изображения и координаты «горячего пятна» (hot spot'a - о проблемы перевода!) внутри изображения. Горячее пятно - это пиксель, по которому определяется точное положение изображения на экране. Обычно горячее пятно определяется таким образом, чтобы оно совпадало с горячим пятном курсора мыши. С этим горячим пятном нам еще предстоит помучиться.

Эта функция начинает операцию по перетаскиванию изображения, накладывая курсор мыши на изображение, определяя горячее пятно и прорисовывая изображение в начальной позиции. Помимо этого, функция также сообщает системе о том, что обновление каких-либо частей экрана запрещено.

Следом за функцией ImageList_BeginDrag() обычно используется функция ImageList_DragEnter(). Описание этой функции можно найти в файле commctrl.h:

```
WINCOMMCTRLAPI BOOL WINAPI ImageList_DragEnter(HWND hwndLock,
                                               int x, int y);
```

Эта функция запрещает обновление указанного окна во время выполнения операции «drag-and-drop» и прорисовывает перемещаемое изображение в промежуточных позициях (до того, как будет отпущена клавиша мыши). В некотором смысле можно сказать, что эта функция делает изображение курсором мыши (конечно, нельзя понимать это буквально, сходство чисто внешнее). Первый аргумент этой функции понятен - хэндл окна, обновление которого запрещается. Этим окном является то окно, в котором производится перемещение изображения. Второй и третий аргументы определяют координаты той точки, в которой необходимо прорисовать изображение. ВНИМАНИЕ! В данном случае необходимо указывать координаты оконные, а не координаты в рабочей области окна. Таким образом, до обращения к этой функции необходимо определить ширину границы окна, высоту заголовка и, при необходимости, ширину полосы меню. В демонстрационной программе это сделано при

функции GetSystemMetrics(). Рекомендую читателю изучить эту функцию самостоятельно.

Функция ImageList_DragMove() описана следующим образом:

```
WINCOMMCTRLAPI BOOL WINAPI ImageList_DragMove(int x, int y);
```

Эта функция перемещает изображение, но не прорисовывает его. Попробуйте в демонстрационной программе убрать функцию ImageList_DragEnter() и посмотреть, что получится.

Последней функцией, обеспечивающей «drag-and-drop», является ImageList_EndDrag(). Эта функция завершает перемещение, но не разрешает обновление окна и не производит прорисовку перемещенного изображения. Для разрешения обновления окна необходимо вызвать функцию ImageList_DragLeave(), передав ей в качестве аргумента хэндл окна, а затем прорисовать изображение, например, с помощью функции ImageList_Draw().

А теперь, как всегда, демонстрационная программа. В этой программе при создании окна производится прорисовка двух икон в левой верхней части рабочей области. Иконки могут быть скопированы в другое место при использовании операции «drag-and-drop». Думаю, читателю не составит труда при необходимости изменить эту программу так, чтобы иконки не копировались, а перемещались.

При разработке программы я допустил определенного рода плагиат. Одну из иконок я «выдрал» из программы pview95, другую - из примера mixtree, поставляемых с Borland C++ 5.0. Если читатель захочет, то он легко может заменить иконки на свои.

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

#define CX_ICON 32
#define CY_ICON 32

HINSTANCE hInst;

LRESULT CALLBACK ImageListWndProc ( HWND, UINT, UINT, LONG );

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow )
 {
  HWND hWnd ;
  WNDCLASS WndClass ;
  MSG Msg;
  char szClassName[] = "ImageList";
```

```
    hInst = hInstance;
  /* Registering our window class */
  /* Fill WNDCLASS structure */
    WndClass.style = CS_HREDRAW | CS_VREDRAW;
    WndClass.lpfnWndProc = ImageListWndProc;
    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 = "";
    WndClass.lpszClassName = szClassName;

    if ( !RegisterClass(&WndClass) )
       {
        MessageBox(NULL,"Cannot register class","Error",MB_OK);
        return 0;
       }
    hWnd = CreateWindow(szClassName, "Image List Demo Program",
                        WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, NULL, NULL,
                        hInstance,NULL);
    if(!hWnd)
       {
        MessageBox(NULL,"Cannot create window», «Error», MB_OK);
        return 0;
       }
    InitCommonControls();
    /* Show our window */
    ShowWindow(hWnd,nCmdShow);
    UpdateWindow(hWnd);

  /* Beginning of messages cycle */
    while(GetMessage(&Msg, NULL, 0, 0))
      {
       TranslateMessage(&Msg);
       DispatchMessage(&Msg);
      }
    return Msg.wParam;

   }

  LRESULT CALLBACK ImageListWndProc (HWND hWnd, UINT Message,
                                     UINT wParam, LONG lParam )
   {
    static HIMAGELIST hImageList;
    static int i;
    static HDC hDC, hPaintDC;
```

```
PAINTSTRUCT PaintStruct;
RECT rBigRect = {0, 0, CX_ICON * 2, CY_ICON};
RECT rLittleRect1 = {0, 0, CX_ICON, CY_ICON};
static POINT Point, pHotSpot;
static BOOL bCapture = FALSE;
static int nXBorder, nYBorder, nYCaption;

switch(Message)
 {
  case WM_CREATE:
   hDC = GetDC(hWnd);
   hImageList = ImageList_Create(CX_ICON, CY_ICON, ILC_MASK, 3, 3);
   ImageList_AddIcon(hImageList, LoadImage(hInst, "Mixtree.ico",
                          IMAGE_ICON, 0, 0, LR_LOADFROMFILE));
   ImageList_AddIcon(hImageList, LoadImage(hInst, "Pview95.ico",
                          IMAGE_ICON, 0, 0, LR_LOADFROMFILE));
   nXBorder = GetSystemMetrics(SM_CXBORDER);
   nYBorder = GetSystemMetrics(SM_CYBORDER);
   nYCaption = GetSystemMetrics(SM_CYCAPTION);
   return 0;
  case WM_PAINT:
   hPaintDC = BeginPaint(hWnd, &PaintStruct);
   for(i = 0; i < 2; i++)
    ImageList_Draw(hImageList, i, hPaintDC, i * CX_ICON, 0,
                        ILD_NORMAL);
   EndPaint(hWnd, &PaintStruct);
   return 0;
  case WM_LBUTTONDOWN:
   Point.x = LOWORD(lParam);
   Point.y = HIWORD(lParam);
   if (PtInRect(&rBigRect, Point))
     {
      SetCapture(hWnd);
      bCapture = TRUE;
      if(PtInRect(&rLittleRect1, Point))
        i = 0;
       else
        i = 1;
      pHotSpot.x = Point.x - i * CX_ICON;
      pHotSpot.y = Point.y;
      ImageList_BeginDrag(hImageList, i, pHotSpot.x, pHotSpot.y);
      ImageList_DragEnter(hWnd, Point.x + nXBorder, Point.y + nYBorder +
                                 nYCaption);
     }
   return 0;
  case WM_MOUSEMOVE:
   if(bCapture)
     ImageList_DragMove(LOWORD(lParam), HIWORD(lParam) + nYBorder
                                 + nYCaption);
   return 0;
```

```
    case WM_LBUTTONUP:
     if(bCapture)
        {
         ImageList_EndDrag();
         ImageList_DragLeave(hWnd);
         ImageList_Draw(hImageList, i, hDC, LOWORD(lParam)- pHotSpot.x,
                        HIWORD(lParam) - pHotSpot.y, ILD_NORMAL);
         ReleaseCapture();
         bCapture = FALSE;
        }
     return 0;
    case WM_DESTROY:
     ReleaseDC(hWnd, hDC);
     ImageList_Destroy(hImageList);
     PostQuitMessage(0);
     return 0;
  }
 return DefWindowProc(hWnd,Message,wParam, lParam);
}
```

Вид окна до произведения операций «drag-and-drop» показан на рис. 16. На рис. 17 приведен вид этого же окна после выполнения нескольких операций копирования иконок.

Мне бы хотелось, чтобы читатель обратил внимание на возможность легкого перемещения и копирования изображений. Что бы пришлось делать в том случае, если бы здесь не использовался список изображений?

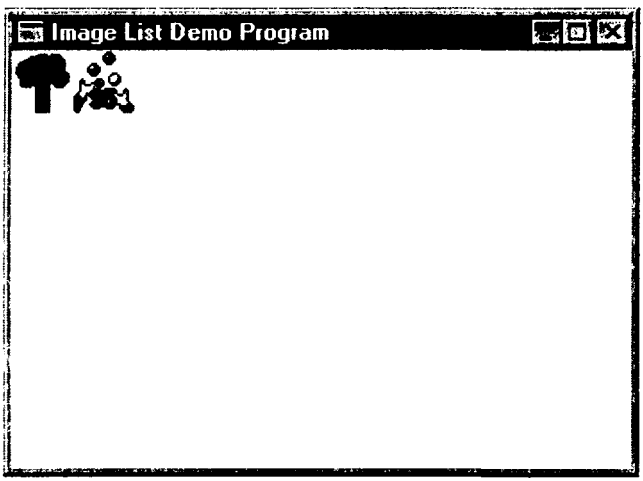


Рис. 16. Окно с двумя изображениями из списка изображений до операции "drag-and-drop"

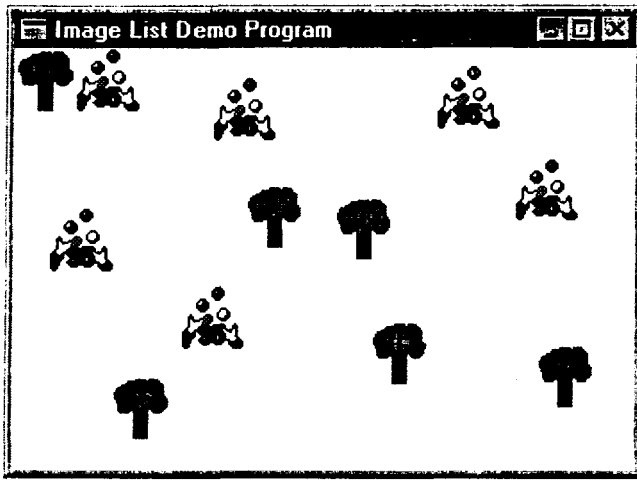


Рис. 17. Предыдущее окно после нескольких операций "drag-and-drop"

РАБОТА С ЗАКЛАДКАМИ

Достаточно интересным элементом управления, появившимся только в Win32, являются закладки. Их появление, как и появление большинства общих элементов управления, давно ожидалось. Этот элемент действует подобно «алфавиту» в записной книжке, при выборе определенной буквы (в данном случае - определенной закладки) всплывает нужная страница (в данном случае - диалоговое окно). Читатель вспомнит, надеюсь, интерфейс электронной таблицы Excel 5.0 и рабочие листы в ней, которые можно было перебрать с помощью переключателей в нижней части таблицы. Эти переключатели и являлись закладками (tab control'ами). Связав каждую из закладок со страницей информации, возможно разместить несколько страниц информации на одном и том же месте. Специальный тип закладок действует как кнопки - при выборе закладки вместо отображения очередной страницы просто производится посылка команды.

К сожалению, и в этом случае специальной функции для создания окна не предусмотрено. Программист должен использовать одну из функций - CreateWindow() или CreateWindowEx(). При этом в качестве имени класса необходимо указать макрос WC_TABCONTROL, который описан в файле commctrl.h следующим образом:

```
#ifdef _WIN32
#define WC_TABCONTROLA          "SysTabControl32"
```

```
#define WC_TABCONTROLW      L"SysTabControl32"
#ifdef UNICODE
#define WC_TABCONTROL       WC_TABCONTROLW
#else
#define WC_TABCONTROL       WC_TABCONTROLA
#endif

#else
#define WC_TABCONTROL       "SysTabControl"
#endif
```

При создании окна с закладками могут использоваться как общие стили, применяющиеся для всех окон, так и стили, специфические для закладок (табл. 43).

Таблица 43. Стили закладок

Стиль	Значение	Описание
TCS_TABS	0x0000	Закладки являются закладками, а не кнопками
TCS_SINGLELINE	0x0000	Закладки располагаются в одну линию и при необходимости скроллируются с помощью up-down control'a
TCS_RIGHTJUSTIFY	0x0000	
TCS_FORCEICONLEFT	0x0010	Иконка сдвинута к левому краю закладки, текст центрирован
TCS_FORCELABELLEFT	0x0020	И текст, и иконка сдвинуты к левому краю закладки
TCS_BUTTONS	0x0100	Закладки выглядят и действуют как кнопки
TCS_MULTILINE	0x0200	Закладки при необходимости располагаются в несколько строк
TCS_FIXEDWIDTH	0x0400	Все закладки имеют одинаковую ширину
TCS_RAGGEDRIGHT	0x0800	
TCS_FOCUSONBUTTONDOWN	0x1000	
TCS_OWNERDRAWFIXED	0x2000	За прорисовку закладок отвечает прикладная программа, а не система
TCS_TOOLTIPS	0x4000	Задержка курсора мыши на одной из закладок вызывает появление подсказки
TCS_FOCUSNEVER	0x8000	Закладка никогда не получает фокуса ввода

После создания окна с закладками, необходимо определить каждую закладку, в которой могут быть иконка, заголовок (текст) и дополнитель-

ные данные, определяемые приложением. Для этого в программе нужно заполнить столько структур типа TC_ITEM, сколько закладок планируется создать. Структура TC_ITEM определена в файле commctrl.h:

```
typedef struct _TC_ITEMA
{
  UINT mask;
  UINT lpReserved1;
  UINT lpReserved2;
  LPSTR pszText;
  int cchTextMax;
  int iImage;

  LPARAM lParam;
} TC_ITEMA;

typedef struct _TC_ITEMW
{
  UINT mask;
  UINT lpReserved1;
  UINT lpReserved2;
  LPWSTR pszText;
  int cchTextMax;
  int iImage;

  LPARAM lParam;
} TC_ITEMW;

#ifdef UNICODE
#define TC_ITEM          TC_ITEMW
#else
#define TC_ITEM          TC_ITEMA
#endif
```

Как следует из названий, поля lpReserved1 и lpReserved2 не используются, они зарезервированы Microsoft для применения в будущем.

В поле mask этой структуры указывается, какие данные определяют внешний вид закладки. Это поле может принимать значения, приведенные в табл. 44.

После того, как читатель ознакомился с этой таблицей, назначения полей pszText, iImage и lParam, надеюсь, стали понятны. Единственное поле, оставшееся нерассмотренным, - это cchTextMax. В случае, если структура типа TC_ITEM используется для получения информации о закладке, в поле cchTextMax определяется размер буфера, на который указывает pszText.

Таблица 44. Битовые флаги, определяющие внешний вид и поведение закладок

Флаг	Значение	Описание
TCIF_TEXT	0x0001	Поле pszText заполнено, в нем хранится указатель на строку - заголовок закладки или на буфер, в который будет записана информация
TCIF_IMAGE	0x0002	Полк iImage заполнено, в нем хранится индекс отображаемого на закладке изображения в списке изображений или -1, если список изображений не используется
TCIF_RTLREADING	0x0004	Текст отображается справа налево, как, например, в арабском языке
TCIF_PARAM	0x0008	Поле lParam заполнено и содержит данные, определяемые приложением

Тем не менее, сделаю одно замечание. Если размер данных, определяемых приложением, не равен 4 байтам, то приложение должно определить собственную структуру и использовать ее вместо TC_ITEM. Первым полем этой структуры должна быть другая структура, типа TC_ITEMHEADER. В файле commctrl.h она описана так:

```
typedef struct _TC_ITEMHEADERA
{
  UINT mask;
  UINT lpReserved1;
  UINT lpReserved2;
  LPSTR pszText;
  int cchTextMax;
  int iImage;
} TC_ITEMHEADERA;

typedef struct _TC_ITEMHEADERW
{
  UINT mask;
  UINT lpReserved1;
  UINT lpReserved2;
  LPWSTR pszText;
  int cchTextMax;
  int iImage;
} TC_ITEMHEADERW;

#ifdef UNICODE
#define TC_ITEMHEADER      TC_ITEMHEADERW
#else
#define TC_ITEMHEADER      TC_ITEMHEADERA
#endif
```

Как всегда, для управления окном с закладками используются сообщения. Их список приведен в табл. 45.

Таблица 45. Сообщения, посылаемые закладкам

Сообщение	Значение	Описание
TCM_FIRST	0x1300	
TCM_GETIMAGELIST	TCM_FIRST + 2	Получить хэндл используемого совместно с закладками списка изображений, wParam и lParam = 0, возвращается хэндл списка изображений
TCM_SETIMAGELIST	TCM_FIRST + 3	Связать список изображений с закладками, wParam = 0, lParam хэндлу списка изображений, возвращается хэндл предыдущего списка изображений
TCM_GETITEMCOUNT	TCM_FIRST + 4	Получить число закладок, wParam = 0, lParam = 0, возвращается число закладок
TCM_GETITEM		Получить информацию о закладке, wParam = индексу закладки, lParam указателю на структуру типа TC_ITEM, в которую будет записана информация
TCM_SETITEM		Установить атрибуты закладки, wParam = индексу закладки, lParam указателю на структуру типа TC_ITEM, которая определяет атрибуты
TCM_INSERTITEM		Вставить закладку, wParam = индексу новой закладки, lParam = указателю на структуру типа TC_ITEM, возвращается индекс новой закладки или -1
TCM_DELETEITEM	TCM_FIRST + 8	Удалить закладку, wParam = индексу закладки, lParam = 0, возвращается TRUE при успешном выполнении
TCM_DELETEALLITEMS	TCM_FIRST + 9	Удалить все закладки, wParam = 0, lParam = 0, возвращается TRUE при успешном выполнении
TCM_GETITEMRECT	TCM_FIRST + 10	
TCM_GETCURSEL	TCM_FIRST + 11	Получение индекса текущей закладки, wParam = 0, lParam = 0
TCM_SETCURSEL	TCM_FIRST + 12	Установка заданной закладки текущей, wParam = индексу закладки, возвращается индекс ранее выбранной закладки
TCM_HITTEST	TCM_FIRST + 13	

Окончание табл. 45

Сообщение	Значение	Описание
TCM_SETITEMEXTRA	TCM_FIRST + 14	Установка размера дополнительных данных для закладки, wParam = числу байт, выделяемых для дополнительных данных
TCM_ADJUSTRECT	TCM_FIRST + 40	
TCM_SETITEMSIZE	TCM_FIRST + 41	
TCM_REMOVEIMAGE	TCM_FIRST + 42	
TCM_SETPADDING	TCM_FIRST + 43	
TCM_GETROWCOUNT	TCM_FIRST + 44	
TCM_GETTOOLTIPS	TCM_FIRST + 45	
TCM_SETTOOLTIPS	TCM_FIRST + 46	
TCM_GETCURFOCUS	TCM_FIRST + 47	
TCM_SETCURFOCUS	TCM_FIRST + 48	

Сообщение TCM_SETITEMEXTRA может в работе окна с закладками использоваться только один раз и только до момента добавления первой закладки. Некоторые сообщения в этом списке, которые могут использовать параметры как в Unicode, так и в ANSI-кодировках, сами являются макросами. В таких случаях в графе «Значение» оставлен пропуск. Для примера ниже приведено описание макроса TCM_GETITEM:

```
#define TCM_GETITEMA        (TCM_FIRST + 5)
#define TCM_GETITEMW        (TCM_FIRST + 60)

#ifdef UNICODE
#define TCM_GETITEM         TCM_GETITEMW
#else
#define TCM_GETITEM         TCM_GETITEMA
#endif
```

В отличие от других элементов управления, для окна с закладками разработаны специальные макросы, которые облегчают работу с сообщениями. Вместо привычного SendMessage(...) можно использовать соответствующие макросы, о которых будет сказано дальше.

Имя каждого макроса образуется из имени сообщения:

1. От имени сообщения отбрасывается префикс TCM_.

2. Все слова оставшейся части изменяются таким образом, что прописной остается только первая буква слова, а все остальные делаются строчными, например, GetItem, InsertItem и т. д.

3. К полученному добавляется префикс TabCtrl_, например, TabCtrl_GetItem, TabCtrl_InsertItem.

Сообщение, которое посылается тем или иным макросом, определяется именем этого макроса. Каждый макрос может содержать один, два или три параметра. Число аргументов определяется очень просто - (число параметров сообщения, не равных нулю) + 1. Если у сообщения wParam и lParam равны 0, то у макроса определен только первый аргумент, если определен только wParam - макрос требует наличия двух аргументов. Первым аргументом макроса всегда является хэндл окна, которому посылается сообщение, т. е. хэндл окна с закладками. Второй и третий аргументы (при необходимости) - это wParam и lParam сообщения соответственно. К примеру, макрос TabCtrl_DeleteAllItems() имеет один аргумент, макрос TabCtrl_DeleteItem() - два аргумента, TabCtrl_InsertItem() - все три аргумента.

Итак, с управляющими сообщениями и макросами все ясно. А как обстоит дело с получением информации о том, что выбрана одна из закладок? Если пользователь что-то сделал с закладкой, то закладка посылает родительскому окну сообщение WM_NOTIFY, при этом wParam этого сообщения содержит идентификатор элемента управления, а lParam - указатель на структуру типа NMHDR. Эта структура описана в файле winuser.h и имеет вид, приведенный ниже:

```
typedef struct tagNMHDR
{
  HWND  hwndFrom;
  UINT  idFrom;
  UINT  code;       // NM_ code
}  NMHDR;
typedef NMHDR FAR * LPNMHDR;
```

Первое поле этой структуры - hwndFrom - содержит хэндл элемента управления, который послал сообщение WM_NOTIFY. Второе поле - idFrom - идентификатор элемента управления. Третье поле - code - содержит код нотификации, т. е. код того действия, которое было произведено с элементом управления. В случае закладки это может быть один из двух кодов - TCN_SELCHANGING или TCN_SELCHANGE.

Сообщение с кодом TCN_SELCHANGING посылается после того, как пользователь произвел действие, но до изменения состояния закладки. Это сообщение может быть использовано, скажем, для того, чтобы сохранить информацию, введенную пользователем в диалоговом окне, связанном с закладкой. После того, как состояние закладки изменилось, посылается сообщение TCN_SELCHANGE. При получении этого сообщения программа может произвести какие-либо действия по формированию вновь отображаемой страницы.

Но использование закладок именно тем и осложнено, что каждая из закладок обычно связана с диалогом. Другими словами, при выборе новой закладки старое диалоговое окно должно исчезнуть, а на его месте должно появиться новое. Так как каждая из закладок может быть выбрана в любой момент, то с закладками должны быть связаны немодальные диалоги. Для того чтобы отобразить впоследствии диалог, связанный с невыбранной закладкой, программа должна сохранять состояние диалога. В качестве примера приведена небольшая программа, иллюстрирующая работу с закладками. Эта программа создает три закладки. При размещении курсора мыши над любой из них возникает подсказка (помните, при разборе окон подсказок я обещал, что продемонстрирую их использование при описании закладок?). При выборе любой из закладок отображается окно диалога, связанное с этой закладкой. В программе используется файл ресурсов:

```
Dialog1 DIALOG 2, 40, 250, 108
STYLE DS_3DLOOK | DS_CONTEXTHELP | WS_POPUP | WS_VISIBLE
FONT 8, "MS Sans Serif"
{
 DEFPUSHBUTTON "OK", IDOK, 24, 65, 50, 14
 CONTROL "This is a text in the first dialog", -1, "static", SS_LEFT | WS_CHILD |
          WS_VISIBLE, 20, 13, 96, 9
}

Dialog2 DIALOG 2, 40, 250, 108
STYLE DS_3DLOOK | DS_CONTEXTHELP | WS_POPUP | WS_VISIBLE
FONT 8, "MS Sans Serif"
{
 DEFPUSHBUTTON "OK", IDOK, 24, 65, 50, 14
 CONTROL "This is a text in the second dialog", -1, "static", SS_LEFT | WS_CHILD |
          WS_VISIBLE, 20, 13, 120, 9
}

Dialog3 DIALOG 2, 40, 250, 108
STYLE DS_3DLOOK | DS_CONTEXTHELP | WS_POPUP | WS_VISIBLE
FONT 8, "MS Sans Serif"
{
 DEFPUSHBUTTON "OK", IDOK, 24, 65, 50, 14
 CONTROL "This is a text in the third dialog", -1, "static", SS_LEFT | WS_CHILD |
          WS_VISIBLE, 20, 13, 96, 9
}
```

Ниже приводится текст программы:

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

```
HINSTANCE hInst;
HWND hWnd;

LRESULT CALLBACK TabControlWndProc ( HWND, UINT, UINT, LONG );
BOOL CALLBACK DialogProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow )
 {
  WNDCLASS WndClass ;
  MSG Msg;
  char szClassName[] = "TabControl";

  hInst = hInstance;
/* Registering our window class */
/* Fill WNDCLASS structure */
  WndClass.style = CS_HREDRAW | CS_VREDRAW;
  WndClass.lpfnWndProc = TabControlWndProc;
  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 (LTGRAY_BRUSH);
  WndClass.lpszMenuName = "";
  WndClass.lpszClassName = szClassName;

  if ( !RegisterClass(&WndClass) )
    {
     MessageBox(NULL,"Cannot register class","Error",MB_OK);
     return 0;
    }

  hWnd = CreateWindow(szClassName, "Tab Control Demo Program",
                      WS_OVERLAPPEDWINDOW & ~WS_THICKFRAME &
                      ~WS_MAXIMIZEBOX,
                      CW_USEDEFAULT, CW_USEDEFAULT, 400, 300,
                      NULL, NULL,
                      hInstance,NULL);
  if(!hWnd)
    {
     MessageBox(NULL,"Cannot create window","Error",MB_OK);
     return 0;
    }

  InitCommonControls();
/* Show our window */
  ShowWindow(hWnd,nCmdShow);
  UpdateWindow(hWnd);
```

```
/* Beginning of messages cycle */
  while(GetMessage(&Msg, NULL, 0, 0))
   {
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
   }
  return Msg.wParam;

 }

LRESULT CALLBACK TabControlWndProc (HWND hWnd, UINT Message,
                                    UINT wParam, LONG lParam )
 {
  HWND hTabControlWnd;
  RECT Rect;
  LPNMHDR lpNMHdr;
  LPTOOLTIPTEXT lpToolTipText;
  static HWND hDlg = 0;
  int nTab;
  TC_ITEM TC_Item;

  switch(Message)
   {
    case WM_CREATE:
     GetClientRect(hWnd, &Rect);
     hTabControlWnd = CreateWindow(WC_TABCONTROL, "", WS_VISIBLE |
                                   WS_TABSTOP | WS_CHILD |
                                   TCS_TOOLTIPS,
                                   0, 0, Rect.right, Rect.bottom,
                                   hWnd, NULL, hInst, NULL);
     TC_Item.mask = TCIF_TEXT;
     TC_Item.iImage = -1;
     TC_Item.pszText = "The first dialog";
     TabCtrl_InsertItem(hTabControlWnd, 0, &TC_Item);
     TC_Item.pszText = "The second dialog";
     TabCtrl_InsertItem(hTabControlWnd, 1, &TC_Item);
     TC_Item.pszText = "The third dialog";
     TabCtrl_InsertItem(hTabControlWnd, 2, &TC_Item);
     hDlg = CreateDialog(hInst, "Dialog1", hTabControlWnd, DialogProc);
     return 0;
    case WM_NOTIFY:
     lpNMHdr = (LPNMHDR) lParam;
     switch(lpNMHdr->code)
      {
       case TTN_NEEDTEXT:
        lpToolTipText = (LPTOOLTIPTEXT) lParam;
        sprintf(lpToolTipText->lpszText, "Tip about tab No %d",
                lpToolTipText->hdr.idFrom);
        break;
       case TCN_SELCHANGE:
```

```
        if(hDlg)
          DestroyWindow(hDlg);
        nTab = TabCtrl_GetCurSel( (HWND) lpNMHdr->hwndFrom);
        switch(nTab)
         {
          case 0:
           hDlg = CreateDialog(hInst, "Dialog1", hTabControlWnd, DialogProc);
           break;
          case 1:
           hDlg = CreateDialog(hInst, "Dialog2", hTabControlWnd, DialogProc);
           break;
          case 2:
           hDlg = CreateDialog(hInst, "Dialog3", hTabControlWnd, DialogProc);
           break;
         }
        break;
       }
     return 0;
    case WM_DESTROY:
     PostQuitMessage(0);
     return 0;
   }
  return DefWindowProc(hWnd,Message,wParam, lParam);
}

BOOL CALLBACK DialogProc(HWND hDlg, UINT Message,
                         WPARAM wParam, LPARAM lParam)
{
 switch(Message)
  {
   case WM_COMMAND:
    PostQuitMessage(0);
    return 1;
  }
 return 0;
}
```

Вид окна, создаваемого программой, приведен на рис. 18.

В этом примере не была использована одна из возможностей, которая резко улучшает внешний вид программы. В закладках можно использовать изображения из созданного ранее списка изображений. Я этого не сделал. В качестве упражнения рекомендую читателю самостоятельно добавить изображения к закладкам.

Ещё один элемент управления рассмотрен. А сколько их осталось? Рассмотрим еще один интересный и достаточно сложный элемент управления, который называется окно просмотра деревьев (Tree View control).

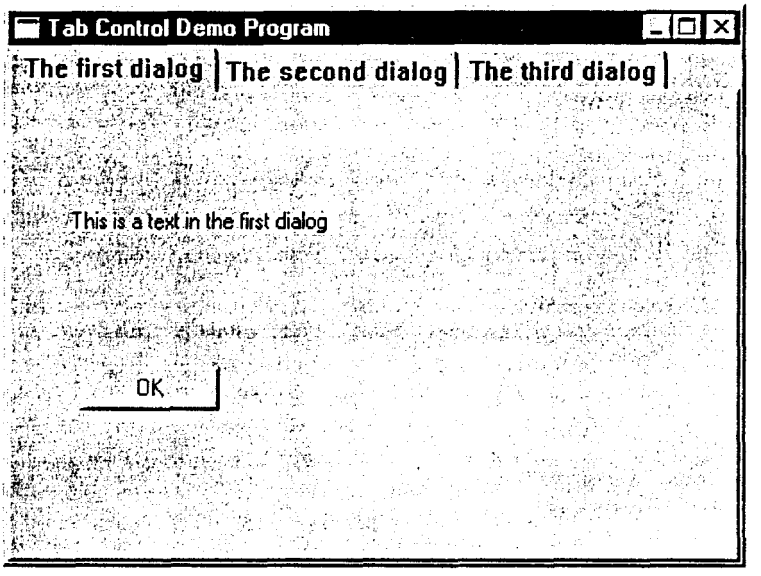


Рис. 18. Окно с закладками

РАБОТА С ОКНОМ ПРОСМОТРА ДЕРЕВЬЕВ

Окно просмотра деревьев используется для просмотра списка объектов, имеющего иерархическую структуру, как, например, файлы на диске. Ярким примером использования окна просмотра деревьев является Explorer, который использует окна этого типа для отображения структуры информации на диске.

И этот элемент управления не имеет специальной функции для своего создания, т. е. для того, чтобы создать окно просмотра деревьев, программа должна использовать функции CreateWindow() или CreateWindowEx(). При этом в качестве имени класса создаваемого окна необходимо использовать макрос WC_TREEVIEW, который описан в файле commctrl.h следующим образом:

```
#ifdef _WIN32
#define WC_TREEVIEWA        "SysTreeView32"
#define WC_TREEVIEWW        L"SysTreeView32"

#ifdef UNICODE
#define  WC_TREEVIEW          WC_TREEVIEWW
#else
```

```
#define WC_TREEVIEW          WC_TREEVIEWA
#endif

#else
#define WC_TREEVIEW          "SysTreeView"
#endif
```

При создании окна просмотра деревьев можно использовать несколько стилей, разработанных специально для окон данного типа. Все эти стили приведены в табл. 46.

На некоторых из этих стилей остановимся более подробно. Начнем со стиля TVS_HASBUTTONS. Если читатель запустит Explorer, то увидит, что у директорий, содержащих внутри себя еще что-то, справа есть небольшой квадратик. Это и есть те кнопки, наличие которых и предполагает стиль. Если элементы следующего для директории уровня отображаются, то внутри квадратика содержится знак «+», в противном случае - знак «-». Для сворачивания и разворачивания ветви дерева необходимо щелкнуть кнопкой мыши на этой кнопке. К сожалению, этот стиль не добавляет кнопки к элементам наивысшего уровня. Для того чтобы появились кнопки и у этих элементов, необходимо комбинировать стили TVS_HASLINES, TVS_LINESATROOT и TVS_HASBUTTONS.

При использовании стиля TVS_HASLINES есть одна особенность. Линиями соединяются только родительские и дочерние элементы.

Таблица 46. Стили окна просмотра деревьев

Стиль	Значение	Описание
TVS_HASBUTTONS	0x0001	К элементам, имеющим дочерние элементы, слева добавляются небольшие кнопки, позволяющие раскрывать и закрывать список подчиненных элементов
TVS_HASLINES	0x0002	Дочерние элементы списка соединяются с родительским элементом линиями, элементы высшего уровня не соединяются
TVS_LINESATROOT	0x0004	Элементы высшего уровня соединяются друг с другом
TVS_EDITLABELS	0x0008	Названия элементов списка могут быть изменены
TVS_DISABLEDRAGDROP	0x0010	Запрещает операции drag-and-drop с элементами списка
TVS_SHOWSELALWAYS	0x0020	Выбранные элементы остаются таковыми даже тогда, когда окно теряет фокус

Таблица 47. Сообщения, посылаемые окнам просмотра деревьев

Сообщение	Значение	Описание
TV_FIRST	0x1100	
TMV_INSERTITEM		Вставка элемента в список
TVM_DELETEITEM	TV_FIRST + 1	Удаление элемента из списка
TVM_EXPAND	TV_FIRST + 2	«Распахнуть» или «свернуть» элемент
TVM_GETITEMRECT	TV_FIRST + 4	Получить ограничивающий прямоугольник для элемента списка
TVM_GETCOUNT	TV_FIRST + 5	Вернуть количество элементов в списке
TVM_GETINDENT	TV_FIRST + 6	Получить значение отступа
TVM_SETINDENT	TV_FIRST + 7	Установить значение отступа
TVM_GETIMAGELIST	TV_FIRST + 8	Получить хэндл списка изображений, связанного с окном просмотра деревьев
TVM_SETIMAGELIST	TV_FIRST + 9	Связать список изображений с окном просмотра деревьев
TVM_GETNEXTITEM	TV_FIRST + 10	Получить дополнительную информацию об элементе, находящемся с текущим в определенных отношениях (следующий, родительский, первый дочерний и так далее)
TVM_SELECTITEM	TV_FIRST + 11	Сделать элемент текущим
TVM_GETITEM		Получить информацию об элементе
TVM_SETITEM		Установить параметры элемента
TVM_EDITLABEL		Изменить текст элемента
TVM_GETEDITCONTROL	TV_FIRST + 15	
TVM_GETVISIBLECOUNT	TV_FIRST + 16	Получить число видимых элементов списка
TVM_HITTEST	TV_FIRST + 17	
TVM_CREATEDRAGIMAGE	TV_FIRST + 18	
TVM_SORTCHILDREN	TV_FIRST + 19	Отсортировать дочерние элементы в алфавитном порядке
TVM_ENSUREVISIBLE	TV_FIRST + 20	
TVM_SORTCHILDRENCB	TV_FIRST + 21	Отсортировать дочерние элементы в соответствии с критерием, определенным программой
TVM_ENDEDITLABELNOW	TV_FIRST + 22	
TVM_GETISEARCHSTRING		

Элементы наивысшего уровня друг с другом не соединяются, т. е. визуально отображаются несколько отдельных деревьев. Если пользователь хочет, чтобы отобразилось единое дерево, необходимо указать комбинацию стилей TVS_HASLINES и TVS_LINESATROOT.

Раз окно просмотра деревьев является окном (прошу извинить меня за тавтологию), то обмен информацией с этим окном и управление им

осуществляется с помощью сообщений. Список всех возможных сообщений, используемых при работе с окном просмотра деревьев, приведен в табл. 47.

Как и в случае с закладками, посылка сообщений дереву просмотра деревьев может быть осуществлена с помощью макросов. Имена макросов для сообщений формируются точно так же, как и в случае закладок. Единственное отличие состоит в том, что в качестве префикса используется не TabCtrl, а TreeView.

А теперь, после краткого знакомства с сообщениями, применяемыми в работе с окнами просмотра деревьев, рассмотрим некоторые из этих сообщений, наиболее части применяемые в прикладных программах.

Для того чтобы вставить элемент в список, необходимо послать окну просмотра деревьев сообщение TVM_INSERTITEM или, что то же самое, использовать макрос TreeView_InsertItem. При этом параметр wParam должен быть равным 0, а lParam должен содержать указатель на структуру типа TV_INSERTSTRUCT. Эта структура описана в файле commctrl.h так:

```
typedef struct _TV_INSERTSTRUCTA {
  HTREEITEM hParent;
  HTREEITEM hInsertAfter;
  TV_ITEMA item;
} TV_INSERTSTRUCTA, FAR *LPTV_INSERTSTRUCTA;

typedef struct _TV_INSERTSTRUCTW {
  HTREEITEM hParent;
  HTREEITEM hInsertAfter;
  TV_ITEMW item;
} TV_INSERTSTRUCTW, FAR *LPTV_INSERTSTRUCTW;

#ifdef UNICODE
#define TV_INSERTSTRUCT     TV_INSERTSTRUCTW
#define LPTV_INSERTSTRUCT   LPTV_INSERTSTRUCTW
#else
#define TV_INSERTSTRUCT     TV_INSERTSTRUCTA
#define LPTV_INSERTSTRUCT   LPTV_INSERTSTRUCTA
#endif
```

Поле первое - hParent - хэндл родительского элемента. Если этот элемент равен TVI_ROOT или NULL, то элемент не имеет родителей и добавляется в список наивысшего уровня.

Второе поле - hInsertAfter - определяет хэндл элемента, после которого вставляется новый элемент. Помимо этого, поле может принимать следующие значения:

TVI_FIRST - элемент вставляется в начало списка;
TVI_LAST - элемент вставляется в конец списка;
TVI_SORT - элемент вставляется в список в алфавитном порядке.

Третье поле - item - описывает непосредственно вставляемый элемент. Он представляет собой очередную структуру (структура в структуре!). Тип этой структуры - TV_ITEM - описан в файле commctrl.h:

```
typedef struct _TV_ITEMA {
  UINT     mask;
  HTREEITEM hItem;
  UINT     state;
  UINT     stateMask;
  LPSTR    pszText;
  int      cchTextMax;
  int      iImage;
  int      iSelectedImage;
  int      cChildren;
  LPARAM   lParam;
} TV_ITEMA, FAR *LPTV_ITEMA;

typedef struct _TV_ITEMW {
  UINT     mask;
  HTREEITEM hItem;
  UINT     state;
  UINT     stateMask;
  LPWSTR   pszText;
  int      cchTextMax;
  int      iImage;
  int      iSelectedImage;
  int      cChildren;
LPARAM   lParam;
} TV_ITEMW, FAR *LPTV_ITEMW;

#ifdef UNICODE
#define TV_ITEM          TV_ITEMW
#define LPTV_ITEM        LPTV_ITEMW
#else
#define TV_ITEM          TV_ITEMA
#define LPTV_ITEM        LPTV_ITEMA
#endif
```

Теперь наберемся сил и рассмотрим структуру типа TV_ITEM. Это поможет нам понять, что представляет собой элемент списка. Кроме этого, при её рассмотрении мы выясним, какого рода информацию об элементе списка можно получить, так как структура именно этого типа используется и для получения информации об элементе.

Таблица 48. Флаги, определяющие в каком поле структуры типа TV_ITEM содержится (или куда должны записываться) информация

Поле	Значение	Описание
TVIF_TEXT	0x0001	Информация содержится в полях pszText и cchTextMax
TVIF_IMAGE	0x0002	Информация содержится в поле iImage
TVIF_PARAM	0x0004	Информация содержится в поле lParam
TVIF_STATE	0x0008	Информация содержится в полях state и stateMask
TVIF_HANDLE	0x0010	Информация содержится в поле hItem
TVIF_SELECTEDIMAGE	0x0020	Информация содержится в поле iSelectedItem
TVIF_CHILDREN	0x0040	Информация содержится в поле cChildren

Первое поле - mask - определяет, в каком из полей этой структуры содержится используемая информация. Это поле может принимать значения, приведенные в табл. 48.

Дело за малым - выяснить, что может храниться в каждом из этих полей.

hItem - хэндл элемента, информация о котором содержится в структуре.

Поле state определяет флаги состояние элемента, а поле stateMask - какое состояние элемента должно быть установлено или получено. Поле state может принимать значения, приведенные в табл. 49.

Таблица 49. Флаги, определяющие внешний вид и состояние окна просмотра деревьев

Состояние	Значение	Описание
TVIS_FOCUSED	0x0001	Элемент получил фокус ввода, т. е. он обрамлен стандартным прямоугольником
TVIS_SELECTED	0x0002	Элемент выбран
TVIS_CUT	0x0004	Элемент выбран для операции копирования
TVIS_DROPHILITED	0x0008	Элемент выбран как место назначения для операции drag-and-drop
TVIS_BOLD	0x0010	Текст элемента написан жирным шрифтом
TVIS_EXPANDED	0x0020	Дочерние элементы списка видны, т. е. элемент «распахнут»
TVIS_EXPANDEDONCE	0x0040	Элемент «распахивался» минимум один раз
TVIS_OVERLAYMASK	0x0F00	
TVIS_STATEIMAGEMASK	0xF000	
TVIS_USERMASK	0xF000	То же, что и предыдущее

Очередное поле - lpszText - содержит указатель на строку, появляющуюся в элементе списка. Помимо этого, поле может иметь значение LPSTR_CALLBACK, в этом случае родительское окно отвечает за формирование текста элемента.

Если структура используется для получения информации об элементе, поле lpszText содержит указатель на буфер, в который будет записан текст элемента. В этом случае поле cchTextMax определяет размер выделенного буфера.

Поля iImage и iSelectedImage определяют индексы изображений, использующихся с элементом, для прорисовки невыбранного и выбранного элементов. Если значение поля равно I_IMAGECALLBACK, то за формирование изображения отвечает родительское окно.

Поле cChildren имеет значение, равное единице, в том случае, когда у элемента есть дочерние элементы. В противном случае значение этого поля равно нулю.

И наконец, последнее поле - lParam - хранит данные, связанные с элементом. Об этих данных мы уже говорили при обсуждении окон списков.

На этом заканчивается рассмотрение параметров сообщения TVM_INSERTITEM. Много ли еще подобных структур ждет нас? Конечно, работает все это эффективно и эффектно (самоё Windows'95 и Windows NT тому подтверждение!), но, по-моему, иногда фирме Microsoft неплохо было бы подумать и о тех, кто будет изучать ее творения! (☺)

Мы изучили только параметры сообщения. Еще нужно узнать, что возвращает функция, с помощью которой было послано данное сообщение. А возвращает она при успешном завершении хэндл элемента, а в случае неудачи - NULL.

Те же действия, как читатель уже догадался, могут быть произведены с помощью макроса TreeView_InsertItem().

После изучения структуры типа TV_ITEM, рекомендую читателю самостоятельно изучить работу сообщений TVM_GETITEM и TVM_SETITEM. В них тоже используется структура этого типа, поэтому никаких сложностей встретиться не должно.

В каждый момент элемент, у которого есть дочерние элементы, может быть «свернут» или «распахнут». Элемент автоматически меняет свое состояние либо при двойном щелчке мышью на нем, либо при щелчке мышью на кнопке элемента, если, конечно, у элемента установлен стиль TVS_HASBUTTONS. Программа может изменять состояние элемента с помощью посылки окну просмотра деревьев сообщений TVM_EXPAND или, что то же самое, обращением к макросу TreeView_Expand(). lParam этого сообщения определяет хэндл элемента, с которым производится

действие, а wParam определяет, что нужно произвести с элементом. В данном случае wParam может принимать значения, приведенные в табл. 50.

В случае изменения состояния элемента окно просмотра деревьев посылает родительскому окну сообщение WM_NOTIFY, посредством которого передает информацию о том, что состояние элемента каким-то образом изменилось. В случае «распахивания» или «сворачивания» элемента родительскому окну посылаются сообщения TVN_ITEMEXPANDING до «распахивания» или «сворачивания» и TVN_ITEMEXPANDED - после.

В том случае, когда элемент «распахнут», дочерние элементы списка отображаются смещенными вправо относительно родительского элемента. Получить значение смещения или установить это значение можно с помощью сообщений TVM_GETINDENT и TVM_SETINDENT.

Программа может дать пользователю возможность изменить текст элемента, послав этому элементу сообщение TVM_EDITLABEL. В этом случае родительское окно получает нотификационные сообщения TVN_BEGINLABELEDIT перед началом редактирования и TVM_ENDLABELEDIT после его окончания.

При необходимости программа может отсортировать элементы списка в алфавитном порядке, послав окну сообщение TVM_SORTCHILDREN. Если необходима сортировка списка по какому-то другому критерию, то тогда окну просмотра деревьев необходимо послать сообщение TVM_SORTCHILDRENCB, указав в качестве lParam адрес процедуры сортировки.

Таблица 50. Действия, производимые с элементом окна просмотра деревьев при посылке окну сообщения TVM_EXPAND

wParam	Значение	Описание
TVE_COLLAPSE	0x0001	Элемент «сворачивается»
TVE_EXPAND	0x0002	Элемент «распахивается»
TVE_TOGGLE	0x0003	Если элемент «свернут», то он «распахивается», и наоборот
TVE_COLLAPSERESET	0x8000	Элемент «сворачивается», при этом дочерние элементы удаляются, действует только в паре с TVE_COLLAPSE

Как и в случае окна закладок, при смене выбранного элемента родительскому окну посылаются нотификационные сообщения TVN_SELCHANGING перед сменой и TVN_SELCHANGED после смены

выбранного элемента. Для того чтобы сменить выбор, программа должна послать окну сообщение TVM_SELECTITEM.

Для того чтобы получить информацию об элементе, необходимо послать окну сообщение TVM_GETITEM. Сообщение TVM_GETNEXTITEM, вопреки своему названию, позволяет получить информацию не только о следующем за текущим элементе, но и о других элементах, находящихся в определенных отношениях с текущим.

Сообщение TVM_GETCOUNT позволяет получить число элементов списка, а сообщение TVM_GETVISIBLECOUNT - число элементов списка, видимых в данный момент.

Для того чтобы связать с окном просмотра деревьев список изображений, нужно воспользоваться сообщением TVM_SETIMAGELIST. Сообщение TVM_GETIMAGELIST позволяет получить хэндл списка изображений, связанного с окном просмотра деревьев.

Для того чтобы пояснить то, о чем шла речь в этом разделе, ниже приведена демонстрационная программа. В ней элементами наивысшего уровня являются десятки от 0 до 100 (0, 10, 20...100), а элементами второго уровня - числа, располагающиеся на числовой оси между целыми десятками. Вот текст этой программы:

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

HINSTANCE hInst;

LRESULT CALLBACK TreeViewWndProc ( HWND, UINT, UINT, LONG );

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow )
 {
  HWND hWnd ;
  WNDCLASS WndClass ;
  MSG Msg;
  char szClassName[] = "TreeView";

  hInst = hInstance;
/* Registering our window class */
/* Fill WNDCLASS structure */
  WndClass.style = CS_HREDRAW | CS_VREDRAW;
  WndClass.lpfnWndProc = TreeViewWndProc;
  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 = "";
  WndClass.lpszClassName = szClassName;

  if ( !RegisterClass(&WndClass) )
    {
     MessageBox(NULL,"Cannot register class","Error",MB_OK);
     return 0;
    }

  hWnd = CreateWindow(szClassName, "TreeView Demo Program",
                      WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                      CW_USEDEFAULT, CW_USEDEFAULT,
                      CW_USEDEFAULT, NULL, NULL,
                       hInstance,NULL);
  if(!hWnd)
    {
     MessageBox(NULL,"Cannot create window","Error",MB_OK);
     return 0;
    }
  InitCommonControls();
/* Show our window */
  ShowWindow(hWnd,nCmdShow);
  UpdateWindow(hWnd);

/* Beginning of messages cycle */
  while(GetMessage(&Msg, NULL, 0, 0))
   {
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
   }
  return Msg.wParam;

 }

LRESULT CALLBACK TreeViewWndProc (HWND hWnd, UINT Message,
                                  UINT wParam, LONG lParam )
 {

  static HWND hTreeView;
  RECT Rect;
  TV_INSERTSTRUCT TV_InsertStruct;
  TV_ITEM TV_Item;
  int i, j;
  char cBuffer[12];

  switch(Message)
   {
    case WM_CREATE:
```

```
      GetClientRect(hWnd, &Rect);
      hTreeView = CreateWindow(WC_TREEVIEW, "",
                               WS_VISIBLE | WS_TABSTOP | WS_CHILD |
                               TVS_HASLINES | TVS_HASBUTTONS |
                               TVS_LINESATROOT,
                               0, 0, Rect.right, Rect.bottom,
                               hWnd, NULL, hInst, NULL);
      TV_InsertStruct.hInsertAfter = TVI_LAST;
      TV_Item.mask = TVIF_TEXT;
      for(i = 0; i < 100; i+= 10)
       {
        TV_InsertStruct.hParent = TVI_ROOT;
        TV_Item.pszText = itoa(i, cBuffer, 10);
        TV_InsertStruct.item = TV_Item;
        TV_InsertStruct.hParent = TreeView_InsertItem(hTreeView,
                                                      &TV_InsertStruct);
        for(j = 1; j < 10; j++)
         {
          TV_Item.pszText = itoa(i + j, cBuffer, 10);
          TV_InsertStruct.item = TV_Item;
          TreeView_InsertItem(hTreeView, &TV_InsertStruct);
         }
       }
      return 0;
     case WM_DESTROY:
      PostQuitMessage(0);
      return 0;
   }
  return DefWindowProc(hWnd,Message,wParam, lParam);
}
```

На рис. 19 показан вид создаваемого программой окна.

Как всегда, самое трудное (☺) - добавить изображения в список - я оставляю читателю в качестве упражнения.

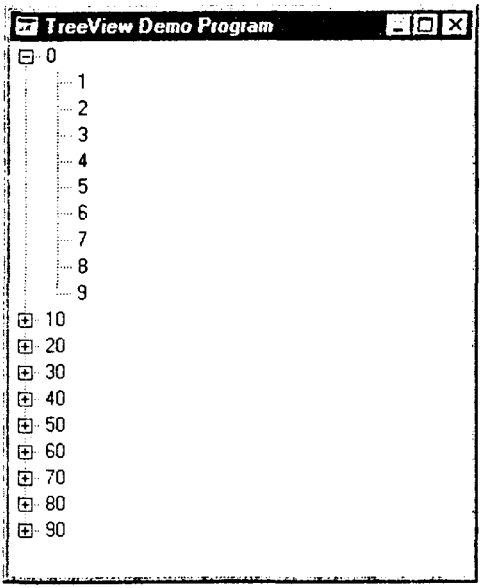


Рис.19. Окно просмотра деревьев с одним «распахнутым» и девятью «нераспахнутыми» элементами

ОКНО РЕДАКТИРОВАНИЯ, ПОДДЕРЖИВАЮЩЕЕ ФОРМАТИРОВАНИЕ ТЕКСТА (Rich Edit Control)

Окно редактирования, о котором пойдет речь ниже (назовём его REC – rich edit control), отличается от обычного окна редактирования тем, что в нем можно форматировать отдельные символы, выделенные элементы текста, абзацы. Кроме этого, этот элемент может включать объекты OLE. Из этого вытекает тот факт, что этот элемент должен предоставлять интерфейс для форматирования текста. Так оно и есть на самом деле. Более того, REC поддерживает почти все сообщения, которые используются обычным многострочным окном редактирования. Другими словами, для того, чтобы заменить в приложении многострочное окно редактирования на REC, требуется минимум усилий.

Как и в случае всех остальных окон, приложение общается с REC при помощи сообщений. Приложение посылает в адрес REC сообщения, на которые REC реагирует надлежащим образом. При изменении своего со-

стояния REC выдаёт нотификационные сообщения, которые позволяют другим окнам реагировать на изменение состояния REC.

Теперь пришло время узнать, каким образом мы можем создать окно редактирования, поддерживающее форматирование текста. Во-первых, мы должны четко уяснить, что всю функциональность окон этого класса обеспечивает библиотека динамической компоновки RICHED32.DLL. Из этого следует, что перед созданием окна этого класса нам необходимо загрузить упомянутую выше библиотеку при помощи функции «LoadLibrary()». Во-вторых, для создания REC в программе должна быть использована функция CreateWindowEx(), при этом в качестве названия класса программы должно быть использовано имя «RichEdit».

Как мы уже знаем, у каждого класса окон, в том числе и у окон редактирования, есть свои стили. Список стилей, поддерживаемых REC, приведен в таблице:

Стиль	Значение	Описание
ES_DISABLENOSCROLL	0x00002000	Если линейки прокрутки не нужны, то окно запрещает их вместо того, чтобы скрыть их
ES_SUNKEN	0x00004000	
ES_SAVESEL	0x00008000	В том случае, если окно теряет фокус, выделение сохраняется
ES_SELFIME	0x00040000	Используется только при работе с азиатскими языками
ES_NOIME	0x00080000	Используется только при работе с азиатскими языками
ES_VERTICAL	0x00400000	Используется только при работе с азиатскими языками
ES_EX_NOCALLOLEINIT	0x01000000	Запрещает вызывать функцию OleInitialize() при создании окна. Используется только в шаблонах диалогов

REC поддерживает следующие стили, используемые с обычными окнами редактирования: ES_AUTOHSCROLL, ES_NOHIDESEL, ES_AUTOVSCROLL, ES_READONLY, ES_CENTER, ES_RIGHT, ES_LEFT, ES_WANTRETURN, ES_MULTILINE.

REC не поддерживает следующие стили: ES_LOWERCASE, ES_PASSWORD, ES_OEMCONVERT, ES_UPPERCASE.

Представим, что окно REC мы уже создали. Давайте теперь посмотрим, каким образом мы можем использовать преимущества, предоставляемые REC, по сравнению с обычным окном редактирования. Итак, какие действия мы можем произвести с текстом, используя возможности, предоставляемые REC?

Во-первых, мы можем отформатировать сразу весь абзац при помощи сообщения EM_SETPARAFORMAT. Получить информацию о том, каким образом отформатирован абзац, мы можем при помощи сообщения EM_GETPARAFORMAT.

Во-вторых, мы можем отформатировать один или несколько символов текста при помощи сообщения EM_SETCHARFORMAT. А узнать о том, какие атрибуты присвоены тому или иному символу, мы можем при помощи сообщения EM_GETCHARFORMAT.

В третьих, для того, чтобы изменить цвет текста, мы используем форматирование абзаца или символов. Но цвет фона – это собственность REC, поэтому мы можем установить цвет фона при помощи сообщения EN_SETBKGNDCOLOR.

Но после того, что я сказал, у пытливого читателя может возникнуть один вопрос: а каким образом программа определит, какие символы необходимо форматировать? Вероятно, ей необходимо каким-то образом указать тот участок текста, с которым необходимо произвести все действия? Естественно, для того, чтобы определить тот участок текста, с которым будут производиться те или иные действия, мы должны ВЫДЕЛИТЬ этот участок при помощи клавиатуры, мышки или программно. Как выделить участок текста при помощи клавиатуры или мышки, я надеюсь, читатель знает. А как выделить участок текста программным способом, я описал в разделе об окнах редактирования. Но здесь читателя поджидают несколько ловушек, связанных с размерами буферов обычного окна редактирования и REC. Для того чтобы предоставить читателю возможность в полной мере использовать все возможности REC, я хотел бы остановиться на этом моменте более подробно.

Как читатель помнит, обычное окно редактирования для установки выделения и получения информации о выделении использует сообщения EM_SETSEL и EM_GETSEL. При установке выделения при помощи сообщения EM_SETSEL программист должен указать в качестве wParam и lParam номер начального и конечного символов выделения. Так как каждый из этих параметров занимают одно слово, то, очевидно, что выделены могут быть максимум 64 К текста. Но! По умолчанию REC содержит не более 32 К текста! Это и есть первая ловушка. Для того чтобы увеличить размер буфера REC, нам необходимо послать окну сообщение EM_EXITLIMITTEXT. При этом wParam этого сообщения должен быть

установлен в нуль, а lParam должен содержать устанавливаемый размер буфера REC. lParam в данном случае является двойным словом, т. е. мы фактически можем не ограничивать размер буфера. Здесь уже прорисовывается вторая ловушка. А как быть в том случае, если мы установили размер буфера REC более 64 К? Что, неужели мы сможем выделять текст, который находится только в границах первых 64 К? Конечно же, на этот вопрос следует дать отрицательный ответ. Дело в том, что в этом случае нужно будет воспользоваться не сообщением EM_SETSEL, а другим сообщением – EM_EXSETSEL. Оно предназначено тоже для выделения какого-то участка текста, но его параметры несколько отличны от параметров сообщения EM_SETSEL. Параметр wParam сообщения EM_SETSEL должен быть равным нулю, а параметр lParam должен указывать на структуру CHARRANGE. Эта структура в файле richedit.h описана следующим образом:

```
typedef struct _charrange
{
    LONG   cpMin;
    LONG   cpMax;
} CHARRANGE;
```

В первом поле этой структуры должен быть указан номер первого символа выделенного участка текста, а во втором – номер последнего символа выделения. Заметьте, уважаемый читатель, что в данном случае каждый из параметров имеет размер двойного слова, т. е. практически мы можем получить выделенный участок текста ПРАКТИЧЕСКИ неограниченного размера, что нам и требовалось!

А каким образом можно узнать о том, что выделенный участок изменился? В том случае, если произошло какое-то изменение выделенного участка, элемент REC выдаёт нотификационное сообщение EL_SELCHANGE, wParam которого содержит идентификатор REC, а lParam – указатель на структуру SELCHANGE, описанную в файле richedit.h следующим образом:

```
typedef struct _selchange
{
    NMHDR nmhdr;
    CHARRANGE chrg;
```

```
    WORD seltyp;
} SELCHANGE;
```

Первое поле этой структуры содержит заголовок нотификационного сообщения, второе поле само по себе является структурой типа CHARRANGE, в котором содержится информация о выделенном участке. И наконец, третье поле определяет содержимое выделенного участка. Значения, которые может принимать это поле, приведены в таблице:

Название	Значение	Назначение
SEL_EMPTY	0x0000	Выделенный участок отсутствует
SEL_TEXT	0x0001	Выделенный участок содержит текст
SEL_OBJECT	0x0002	Выделенный участок содержит по меньшей мере один объект OLE
SEL_MULTICHAR	0x0004	Выделенный участок содержит более одного символа текста
SEL_MULTIOBJECT	0x0008	Выделенный участок содержит более одного объекта OLE

Обрабатывая нотификационное сообщение EL_SELCHANGE приложение имеет возможность реагировать на изменения выделенного участка текста.

Теперь еще несколько слов о выделении. По умолчанию окно REC скрывает выделение в том случае, если оно теряет фокус, и вновь показывает его, когда вновь получает фокус. Иногда это бывает не совсем удобно. Желательно иметь возможность скрывать и показывать выделение в любой момент независимо от фокуса. Для этой цели можно использовать сообщение EM_HIDESELECTION. Параметр wParam этого сообщения определяет, что необходимо сделать с выделенным участком текста – скрыть или показать его. Если этот параметр равен нулю, выделение показывается. В противном случае выделение скрывается. Второй параметр, lParam, определяет, каким образом необходимо обрабатывать сообщение. Если этот параметр равен нулю, то выделение ВРЕМЕННО скрывается или показывается. Если же lParam не равен нулю, то изменяется значение стиля ES_NOHIDESEL окна REC.

Для того чтобы отформатировать абзац, окну REC необходимо послать сообщение EM_SETPARAFORMAT. При этом параметр wParam должен быть равным нулю, а lParam должен указывать на структуру типа PARAFORMAT, в которой должна содержаться информация о форматировании. Эта структура описана в файле richedit.h следующим образом:

```
typedef struct _paraformat
{
    UINT   cbSize;
    _WPAD  _wPad1;
    DWORD  dwMask;
    WORD   wNumbering;
    WORD   wReserved;
    LONG   dxStartIndent;
    LONG   dxRightIndent;
    LONG   dxOffset;
    WORD   wAlignment;
    SHORT  cTabCount;
    LONG   rgxTabs[MAX_TAB_STOPS];
} PARAFORMAT;
```

Теперь я прошу уважаемого читателя набраться терпения, ибо нам необходимо рассмотреть назначение полей этой структуры. Итак...

Первое поле, cbSize, должно содержать размер этой структуры в байтах. О чем это говорит? Да только о том, что в Microsoft допускают возможность изменения в будущем формата этой структуры. Примем этот факт к сведению.

Следующее поле, dwMask, занимает двойное слово, что следует из его названия, и представляет собой набор флагов. Эти флаги определяют, какие поля используются при текущем обращении к структуре. Значения и назначение флагов приведены в таблице:

Название флага	Значение	Назначение
PFM_STARTINDENT	0x00000001	Используется поле dxStartIndent (если одновременно установлены флаги PFM_OFFSETINDENT и PFM_STARTINDENT, то последний имеет преимущество)
PFM_RIGHTINDENT	0x00000002	Используется поле dxRightIndent
PFM_OFFSET	0x00000004	Используется поле dxOffset
PFM_ALIGNMENT	0x00000008	Используется поле wAlignment

PFM_TABSTOPS	0x00000010	Используются поля cTabStops и rgxTabStops
PFM_NUMBERING	0x00000020	Используется поле wNumbering
PFM_OFFSETINDENT	0x80000000	Используется поле dxStartIndent и указывается смещение относительно этого поля

Так как поле dwMask используется как логическая шкала, то при установке флагов могут быть использованы операторы логического сложения (OR) и логического умножения (AND).

Поле wNumbering указывает параметры нумерации. Оно может принимать всего два значения – нуль и PFN_BULLET. Кстати, макрос PFN_BULLET в файле richedit.h описан следующим образом:

```
#define PFN_BULLET          0x0001
```

Название следующего поля – wReserved – говорит само за себя. Это поле длиной в одно слово, которое зарезервировано фирмой Microsoft для использования в будущем. Пока никакого интереса для нас оно не представляет.

Поле dxStartIndent содержит размер отступа первой строки абзаца. Однако, если абзац уже отформатирован, но в поле dwMask установлен флаг PFM_OFFSETINDENT, то значение поля dxStartIndent добавляется к отступу первой строки параграфа.

Поле PFM_RIGHTINDENT содержит смещение обычных строк абзаца относительно правой границы текста.

Поле dxOffset определяет отступ второй и и всех последующих строк абзаца относительно первой строки. Если значение этого поля отрицательное, то первая строка имеет отступ. При положительном значении этого поля первая строка имеет выступ.

Поле wAlignment определяет характер выравнивания строк абзаца и может принимать следующие значения:

Название флага	Значение	Назначение
PFA_LEFT	0x0001	Абзац выравнивается по левой границе
PFA_RIGHT	0x0002	Абзац выравнивается по правой границе
PFA_CENTER	0x0003	Абзац выравнивается по центру страницы

А следующие два поля работают в связке. Поле cTabCount определяет количество позиций табуляции, которые будут использоваться в форматируемом абзаце.

Последнее поле, rgxTabs, является массивом, с числом элементов, равным MAX_TAB_STOPS. Этот макрос в файле richedit.h определяется следующим образом:

```
#define MAX_TAB_STOPS 32
```

А где же связка, может спросить любознательный читатель? Дело в том, что поле cTabCount определяет число фактически использующихся элементов массива.

Вот, кажется, и все, что необходимо знать для того, чтобы отформатировать абзац. А как получить информацию о том, каким образом отформатирован абзац? Нет ничего проще! Необходимо только послать окну REC сообщение EM_GETPARAFORMAT. wParam этого сообщения должен быть равен нулю, а второй параметр должен содержать указатель на структуру типа PARAFORMAT, которая будет заполнена информацией о форматировании абзаца.

Для того чтобы отформатировать символ или несколько символов, мы можем послать окну REC сообщение EM_SETCHARFORMAT. Первый параметр, wParam, - это флаг. Значения этого флага приведены в таблице:

Название	Значение	Назначение
SCF_SELECTION	0x0001	Форматирование применяется к выделенному участку текста, или устанавливает форматирование по умолчанию, если ничего в тексте не выделено
SCF_WORD	0x0002	Форматирование применяется к текущему слову или словам. Если в тексте ничего не выделено, но указатель находится внутри слова, форматирование применяется к этому слову

Второй параметр, lParam, должен указывать на структуру типа CHARFORMAT, которая описана в файле richedit.h следующим образом:

```
typedef struct _charformat
{
    UINT        cbSize;
```

```
    _WPAD       _wPad1;
    DWORD       dwMask;
    DWORD       dwEffects;
    LONG        yHeight;
    LONG        yOffset;            /* > 0 for superscript, < 0 for subscript */
    COLORREF    crTextColor;
    BYTE        bCharSet;
    BYTE        bPitchAndFamily;
    TCHAR       szFaceName[LF_FACESIZE];
    _WPAD       _wPad2;
} CHARFORMAT;
```

Теперь приходится (пусть читатель меня извинит, но ведь я тоже человек, и мне тоже надоедают бесконечные описания структур, полей, флагов и прочего) рассматривать назначение всех полей этой структуры. Итак, начнем.

Первое поле, cbSize, должно содержать размер структуры типа CHARFORMAT. Как и в предыдущем случае, это говорит только о том, что фирма Microsoft в будущем может изменить формат этой структуры.

Двойное слово dwMask представляет собой набор флагов. Эти флаги определяют, какие характеристики символов устанавливаются. Назначение этих флагов приведено в таблице:

Название	Значение	Назначение
CFM_BOLD	0x00000001	Устанавливается значение, определённое флагом CFE_BOLD поля dwEffects
CFM_ITALIC	0x00000002	Устанавливается значение, определённое флагом CFE_ITALIC поля dwEffects
CFM_UNDERLINE	0x00000004	Устанавливается значение, определённое флагом CFE_UNDERLINE поля dwEffects
CFM_STRIKEOUT	0x00000008	Устанавливается значение, определённое флагом CFE_STRIKEOUT поля dwEffects
CFM_PROTECTED	0x00000010	Устанавливается значение, определённое флагом CFE_PROTECTED поля dwEffects
CFM_CHARSET	0x08000000	
CFM_OFFSET	0x10000000	

CFM_FACE	0x20000000	
CFM_COLOR	0x40000000	
CFM_SIZE	0x80000000	Устанавливается высота символа, определяемая полем yHeight

Поле dwEffects определяет, как я уже сказал выше, характеристики символа. Это поле может быть комбинацией следующих характеристик:

Название	Значение	Назначение
CFE_BOLD	0x0001	Символ делается жирным
CFE_ITALIC	0x0002	Символ делается наклонным
CFE_UNDERLINE	0x0004	Символ подчёркивается
CFE_STRIEOUT	0x0008	
CFE_PROTECTED	0x0010	Символ делается защищённым
CFE_AUTOCOLOR	0x40000000	Цвет текста устанавливается как значение, возвращаемое функцией GetSysColor (COLOR_WINDOWTEXT)

Поле yHeight указывает размер символа, т. е. фактически его высоту, что и следует из названия поля.

Поле yOffset определяет смещение символа по вертикали от основной линии строки. Положительное смещение говорит о том, что символ приподнят над основной линией строки, отрицательное смещение говорит о том, что символ находится ниже основной линии строки.

Поле crTextColor определяет цвет символа. Значение этого поля игнорируется, если установлены флаги CFM_COLOR поля dwMask и CFE_AUTOCOLOR поля dwEffects.

Поле bCharSet определяет набор символов, которому принадлежит форматируемый символ. Это поле может принимать следующие значения:

ANSI_CHARSET

DEFAULT_CHARSET

SYMBOL_CHARSET

SHIFTJIS_CHARSET

GB2312_CHARSET

HANGEUL_CHARSET

CHINESEBIG5_CHARSET

OEM_CHARSET

JOHAB_CHARSET

HEBREW_CHARSET

ARABIC_CHARSET

GREEK_CHARSET

TURKISH_CHARSET

THAI_CHARSET

EASTEUROPE_CHARSET

RUSSIAN_CHARSET

MAC_CHARSET

BALTIC_CHARSET

Следующее поле, bPitchAndFamily, определяет ширину символов и характеристики шрифтов. Младшие два байта определяют ширину символов:

Название	Значение	Назначение
DEFAULT_PITCH	0	Ширина символов, принятая по умолчанию
FIXED_PITCH	1	Фиксированная ширина символов
VARIABLE_PITCH	2	Переменная ширина символов
MONO_FONT	8	

Биты с четвертого по седьмой определяют характеристики шрифта. Эти характеристики определены в файле wingdi.h следующим образом:

Название	Значение	Назначение
FF_DONTCARE	(0<<4)	Семейство шрифтов неизвестно или безразлично
FF_ROMAN	(1<<4)	Семейство шрифтов с переменной шириной символов и засечками
FF_SWISS	(2<<4)	Семейство шрифтов с переменной шириной символов и без засечек

FF_MODERN	(3<<4)	Семейство шрифтов с постоянной шириной символов, с засечками или без засечек
FF_SCRIPT	(4<<4)	Семейство курсивных шрифтов или шрифтов, имитирующих рукописный текст
FF_DECORATIVE	(5<<4)	Семейство декоративных шрифтов

И наконец, последнее поле, szFaceName, является символьным массивом, число элементов в котором определено как LF_FACESIZE. Этот макрос определен в файле wingdi.h следующим образом:

```
#define LF_FACESIZE      32
```

В этот массив записывается наименование шрифта.

Ну, вот, кажется, и все о структуре типа CHARFORMAT. Но нет, не все. Еще раз мы встретимся с этой структурой тогда, когда нам потребуется информация о характеристиках символов. Для того чтобы эту информацию получить, нам необходимо послать элементу REC сообщение EM_GETCHARFORMAT. Параметр wParam этого сообщения должен определять, какие данные о форматировании мы хотим получить. Если wParam равен нулю, то это означает, что мы пытаемся получить информацию о форматировании по умолчанию. В противном случае будет возвращена информация о форматировании выделенного участка текста. Параметр lParam этого сообщения должен указывать на структуру типа CHARFORMAT, которая будет заполнена информацией о форматировании.

Я надеюсь, я рассказал достаточно о работе окна редактирования, поддерживающего форматирование текста. Теперь, как всегда, настало время для демонстрационной программы. Не мудрствуя лукаво, я попробую несколько видоизменить программу, которую я привел в разделе, посвященном обычному окну редактирования. Все изменение заключается в том, что вместо обычного окна редактирования я буду использовать окно редактирования, поддерживающее форматирование. В этом окне мы сможем, в дополнение ко всему, изменять шрифт текста, делать выделенные символы курсивными, подчёркнутыми и выделять их жирным написанием. Программа использует следующий файл ресурсов:

```
RTF_MENU MENU
{
POPUP "&File"
{
```

```
 MENUITEM "E&xit", 101
}

POPUP "&Edit"
{
 MENUITEM "&Bold", 201
 MENUITEM "&Italic", 202
 MENUITEM "&Underline", 203
 MENUITEM "F&onts", 204
}
}
```

А ниже я привожу текст основной программы:

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

#define IDM_Exit 101
#define IDM_Bold 201
#define IDM_Italic 202
#define IDM_Underline 203

HINSTANCE hInst;

LRESULT CALLBACK EditDemoWndProc ( HWND, UINT, UINT, LONG );

int WINAPI WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                     LPSTR lpszCmdParam, int nCmdShow )

 {
```

```
    HWND hWnd ;
    WNDCLASS WndClass ;
    MSG Msg;
    char szClassName[] = "EditDemo";

    hInst = hInstance;
/* Registering our window class */
/* Fill WNDCLASS structure */
    WndClass.style = CS_HREDRAW | CS_VREDRAW;
    WndClass.lpfnWndProc = EditDemoWndProc;
    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 = "RTF_MENU";
    WndClass.lpszClassName = szClassName,

    if ( !RegisterClass(&WndClass) )
      {
       MessageBox(NULL,"Cannot register class","Error",MB_OK);
       return 0;
      }

    hWnd = CreateWindow(szClassName, "EditDemo",
                        WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, NULL, NULL,
```

```
                    hInstance,NULL);
  if(!hWnd)
     {
      MessageBox(NULL,"Cannot create window","Error",MB_OK);
      return 0;
     }

  /* Show our window */
  ShowWindow(hWnd,nCmdShow);
  UpdateWindow(hWnd);

/* Beginning of messages cycle */
  while(GetMessage(&Msg, NULL, 0, 0))
    {
     TranslateMessage(&Msg);
     DispatchMessage(&Msg);
    }
  return Msg.wParam;
 }

LRESULT CALLBACK EditDemoWndProc (HWND hWnd, UINT Message,
                    UINT wParam, LONG lParam )
 {
  static HWND hRtfEditWnd;
  RECT Rect;
  HINSTANCE hLibrary;
  CHARFORMAT CharFormat;

  switch(Message)
```

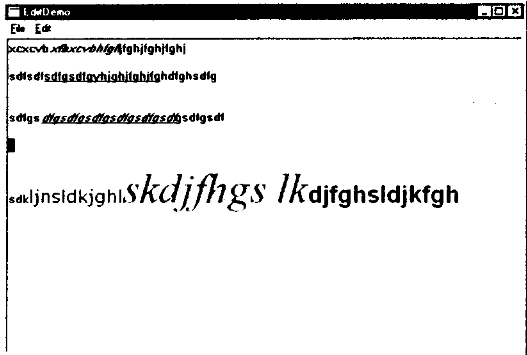
```
{
case WM_CREATE:
 hLibrary = LoadLibrary("riched32.dll");
 GetClientRect(hWnd, &Rect);
 hRtfEditWnd = CreateWindow("RichEdit", NULL,
                WS_CHILD | WS_VISIBLE |
                WS_HSCROLL | WS_VSCROLL |
                WS_BORDER | ES_LEFT |
                ES_MULTILINE | ES_AUTOHSCROLL |
                ES_AUTOVSCROLL,
                0, 0, 0, 0,
                hWnd, NULL, hInst, NULL);
 return 0;
case WM_SIZE:
 MoveWindow(hRtfEditWnd, 0, 0, LOWORD(lParam),
              HIWORD(lParam), TRUE);
 return 0;
case WM_SETFOCUS:
 SetFocus(hRtfEditWnd);
 return 0;
case WM_COMMAND:
 switch(LOWORD(wParam))
  {
   case IDM_Exit:
    SendMessage(hWnd, WM_DESTROY, 0, 0);
    break;
   case IDM_Bold:
    CharFormat.cbSize = sizeof(CHARFORMAT);
    CharFormat.dwMask = CFM_BOLD;
```

```
  SendMessage(hRtfEditWnd, EM_GETCHARFORMAT, TRUE,
              (LPARAM)&CharFormat);
  CharFormat.dwMask = CFM_BOLD;
  CharFormat.dwEffects ^= CFE_BOLD;
  SendMessage(hRtfEditWnd, EM_SETCHARFORMAT,
              SCF_SELECTION, (LPARAM)&CharFormat);
  break;
 case IDM_Italic:
  CharFormat.cbSize = sizeof(CHARFORMAT);
  CharFormat.dwMask = CFM_ITALIC;
  SendMessage(hRtfEditWnd, EM_GETCHARFORMAT, TRUE,
              (LPARAM)&CharFormat);
  CharFormat.dwMask = CFM_ITALIC;
  CharFormat.dwEffects ^= CFE_ITALIC;
  SendMessage(hRtfEditWnd, EM_SETCHARFORMAT,
              SCF_SELECTION, (LPARAM)&CharFormat);
  break;
 case IDM_Underline:
  CharFormat.cbSize = sizeof(CHARFORMAT);
  CharFormat.dwMask = CFM_UNDERLINE;
  SendMessage(hRtfEditWnd, EM_GETCHARFORMAT, TRUE,
              (LPARAM)&CharFormat);
  CharFormat.dwMask = CFM_UNDERLINE;
  CharFormat.dwEffects ^= CFE_UNDERLINE;
  SendMessage(hRtfEditWnd, EM_SETCHARFORMAT,
              SCF_SELECTION, (LPARAM)&CharFormat);
  break;
 case IDM_Fonts:
  CharFormat.cbSize = sizeof(CHARFORMAT);
```

```
SendMessage(hRtfEditWnd, EM_GETCHARFORMAT, TRUE,
                (LPARAM)&CharFormat);
memset(&ChooseFontStructure, 0, sizeof(CHOOSEFONT));
memset(&LogFont, 0, sizeof(LOGFONT));
hDC = GetDC(hWnd);
LogFont.lfHeight = CharFormat.yHeight/-20;
if (CharFormat.dwEffects & CFE_BOLD)
  LogFont.lfWeight = FW_BOLD;
 else
  LogFont.lfWeight = FW_NORMAL;
  LogFont.lfItalic = (BOOL)(CharFormat.dwEffects & CFE_ITALIC);
  LogFont.lfUnderline = (BOOL)(CharFormat.dwEffects &
                                CFE_UNDERLINE);
  LogFont.lfCharSet = DEFAULT_CHARSET;
  LogFont.lfQuality = DEFAULT_QUALITY;
  LogFont.lfPitchAndFamily = CharFormat.bPitchAndFamily;
  lstrcpy(LogFont.lfFaceName, CharFormat.szFaceName);
  ChooseFontStructure.lStructSize = sizeof(ChooseFontStructure);
  ChooseFontStructure.hwndOwner = hWnd;
  ChooseFontStructure.hDC = hDC;
  ChooseFontStructure.lpLogFont = &LogFont;
  ChooseFontStructure.Flags = CF_SCREENFONTS |
                              CF_INITTOLOGFONTSTRUCT;
ChooseFontStructure.rgbColors = RGB(0,0,0);
ChooseFontStructure.nFontType = SCREEN_FONTTYPE;
if (ChooseFont(&ChooseFontStructure))
  {
   CharFormat.dwMask = CFM_BOLD | CFM_FACE | CFM_ITALIC |
                       CFM_OFFSET | CFM_SIZE |
```

```
                                    CFM_UNDERLINE;
            CharFormat.yHeight = LogFont.lfHeight * -20;
            CharFormat.dwEffects = 0;
            if (FW_BOLD == LogFont.lfWeight)
              CharFormat.dwEffects |= CFE_BOLD;
            if (LogFont.lfItalic)
              CharFormat.dwEffects |= CFE_ITALIC;
            if (LogFont.lfUnderline)
              CharFormat.dwEffects |= CFE_UNDERLINE;
            CharFormat.bPitchAndFamily = LogFont.lfPitchAndFamily;
            lstrcpy(CharFormat.szFaceName, LogFont.lfFaceName);
            SendMessage(hRtfEditWnd, EM_SETCHARFORMAT,
                        SCF_SELECTION, (LPARAM)&CharFormat);
          }
        ReleaseDC(hWnd, hDC);
        break;
      default:
        break;
    }
  break;
 case WM_DESTROY:
   FreeLibrary(hLibrary);
   PostQuitMessage(0);
   return 0;
 }
return DefWindowProc(hWnd,Message,wParam, lParam);
}
```

А теперь я привожу внешний вид окна, которое возникает при работе этой демонстрационной программы:



РЕЕСТР

Реестр – это база данных, определенная в Windows, которая используется приложениями для того, чтобы хранить в ней конфигурационные данные.

Надеюсь, что читатель помнит массу файлов с расширением .ini в подавляющем большинстве случаев, которые приложения, разработанные для Windows более старых версий, использовали для хранения данных. Там хранились данные, нужные только данному приложению для работы. Для работы с ними использовались функции, имена которых содержали строку PrivateProfile. В Win32 для хранения подобных данных разработан совершенно новый механизм, получивший название реестра (registry – реестр, регистратура). Этот механизм, во-первых, облегчил работу с данными приложений, и, во-вторых, упростил работу с ними. При этом следует заметить, что хотя никаких особых ограничений для хранения информации нет, хранить в нем следует только инициализационные и конфигурационные данные. В helh'е по Win'32 API записано, что если данные превышают один килобайт, их целесообразно хранить в отдельном файле, а не в регистре. Мне кажется, что в большинстве случаев следует поступать именно так. Какой же должна быть программа, чтобы данные инициализации и конфигурационные данные занимали бы 1 кбайт!

СТРУКТУРА РЕЕСТРА

Данные в реестре хранятся в виде иерархического дерева. Каждый элемент этого дерева называется ключом. Каждый ключ может содержать как ключи более низкого уровня, так и конечные элементы данных. При необходимости приложение открывает ключ и использует данные, сохра-

ненные в нем. Каждый ключ может содержать любое число данных (конечно, все ограничивается объемом памяти), при этом данные могут быть в произвольном формате. Учитывая то, о чем я говорил выше, если данных очень много и они хранятся в отдельном файле, то в реестре может быть создан ключ, который ссылался бы на этот файл. Имена ключей не могут содержать обратные слеши (\ ), пробелы, звездочки (*) и вопросительные знаки. Имя ключа не должно совпадать с именами ключей, располагающихся выше него по иерархии.

РАБОТА С РЕЕСТРОМ

СОЗДАНИЕ И ОТКРЫТИЕ КЛЮЧЕЙ

Для того чтобы работать с данными реестра, приложение должно сначала создать собственный ключ или открыть ключ, созданный ранее. Для создания ключа приложению необходимо вызвать функцию RegCreateKeyEx(), которая описана в файле winreg.h так:

```
WINADVAPI LONG APIENTRY RegCreateKeyExA (HKEY hKey,
                                         LPCSTR lpSubKey,
                                         DWORD Reserved,
                                         LPSTR lpClass,
                                         DWORD dwOptions,
                                         REGSAM samDesired,
                    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                    PHKEY phkResult, LPDWORD lpdwDisposition);
WINADVAPI LONG APIENTRY RegCreateKeyExW (HKEY hKey,
                                         LPCWSTR lpSubKey,
                                         DWORD Reserved,
                                         LPWSTR lpClass,
                                         DWORD dwOptions,
                                         REGSAM samDesired,
                     LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                     PHKEY phkResult, LPDWORD lpdwDisposition);
#ifdef UNICODE
#define RegCreateKeyEx  RegCreateKeyExW
#else
#define RegCreateKeyEx  RegCreateKeyExA
#endif // !UNICODE
```

Опять функция с массой аргументов! Первый аргумент - hKey -хэндл ранее открытого ключа или одно из следующих значений:

HKEY_CLASSES_ROOT;
HKEY_CURRENT_USER;
HKEY_LOCAL_MACHINE;
HKEY_USERS.

Здесь нужно остановиться и рассмотреть, что за значения были приведены выше.

При инсталляции Windows создаются четыре ключа. Их имена совпадают со значениями, приведенными выше. Другими словами, эти ключи являются основой для создания иерархии ключей.

Ключи, находящиеся по иерархии ниже первого из предопределенных ключей, HKEY_LOCAL_MACHINE, определяют физическое состояние компьютера, включая данные о типе шины, системной памяти, инсталлированном аппаратном и программном обеспечении.

Ключи, находящиеся по иерархии ниже HKEY_CLASSES_ROOT, определяют типы (или классы) файлов и свойства, ассоциированные с этими классами. Свойства классов определяются только программистом. Обычно эти свойства применяются при работе приложений, использующих внедрение и связывание объектов, а также приложений, использующих среду Windows (shell applications). К примеру, при открытии файлов в Explorer'е используются свойства файлов, записанные в реестре.

Ключи, подчиненные HKEY_USERS, определяют конфигурацию по умолчанию при подключении нового пользователя на локальной машине и конфигурацию текущего пользователя.

И наконец, ключи, подчиненные HKEY_CURRENT_USER, определяют установки, сделанные текущим пользователем, касающиеся переменных окружения, данных о принтерах, сетевых подключениях и т. д. Кроме этого, в этой ветви дерева хранятся установки, сделанные конкретными приложениями.

Возвращаясь к аргументам функции RegCreateKeyEx(), я теперь могу сказать, что перед созданием нового ключа необходимо продумать, в какую ветвь дерева необходимо включить новый ключ. Если новый ключ необходимо создать подчиненным ключу более низкого уровня, то определенным образом можно пройти по дереву и найти тот ключ, который необходим. Кроме этого, ключ, хэндл которого указан в первом аргументе, должен быть открыт с атрибутом доступа KEY_CREATE_SUB_KEY. Об атрибутах доступа мы поговорим при рассмотрении шестого аргумента функции.

Вторым аргументом - lpSubKey - является указатель на строку, содержащую имя создаваемого ключа. Создаваемый ключ будет подчиненным ключа, хэндл которого указан в первом аргументе.

Третий аргумент - Reserved -зарезервирован и должен быть равным нулю.

Четвертый аргумент - lpClass - указатель на строку, определяющую класс создаваемого ключа.

Очередной, пятый аргумент - dwOptions, определяет опции создаваемого ключа. Этот аргумент может принимать одно из значений - REG_OPTION_VOLATILE или REG_OPTION_NON_VOLATILE. В Windows'95 первое значение не используется. Второе значение указывает, что при перезагрузке системы значение этого ключа сохраняется, т. е. информация сохраняется в файле, а не в памяти.

Следующий, шестой аргумент - samDesired, определяет маску доступа к ключу. Этот параметр представляет собой битовую шкалу и может быть комбинацией флагов, приведенных в табл. 51.

Седьмой аргумент - lpSecurityAttributes - указатель на структуру типа SECURITY_ATTRIBUTES, которая определяет атрибуты безопасности создаваемого ключа. К сожалению, Windows'95 не поддерживает безопасность, поэтому этот параметр игнорируется.

Туда, куда указывает восьмой аргумент - phkResult - записывается хэндл созданного ключа.

Таблица 51. Флаги, составляющие маску доступа к ключу

Флаг	Значение	Описание
KEY_QUERY_VALUE	0x0001	Права запрашивать данные подключей
KEY_SET_VALUE	0x0002	Права устанавливать данные подключей
KEY_CREATE_SUB_KEY	0x0004	Права создавать подключи
KEY_ENUMERATE_SUB_KEY	0x0008	Права перебирать подключи
KEY_NOTIFY	0x0010	Права изменять нотификацию
KEY_CREATE_LINK	0x0020	Права создавать символическую связь
KEY_READ		(STANDARD_RIGHTS_READ | KEY_QUERY_VALUE | KEY_ENUMERATE_SUB_KEYS | KEY_NOTIFY) & (~SYNCRONIZE)
KEY_WRITE		(STANDARD_RIGHTS_WRITE | KEY_SET_VALUE | KEY_CREATE_SUB_KEY) & (~SYNCRONIZE)
KEY_EXECUTE		KEY_READ & (~SYNCRONIZE)
KEY_ALL_ACCESS		(STANDARD_RIGHTS_ALL | KEY_QUERY_VALUE | KEY_SET_VALUE | KEY_CREATE_SUB_KEY | KEY_ENUMERATE_SUB_KEYS | KEY_NOTIFY | KEY_CREATE_LINK) & (~SYNCRONIZE))

И наконец, последний, девятый аргумент - lpdwDisposition - указывает место, куда будет записана информация о том, что произошло с ключом. Дело в том, что если с помощью этой функции производится попытка создать ключ, который уже существует, то ключ не создается, а просто открывается. Поэтому приложению необходимо знать, что произошло при создании ключа. Если ключ был создан, то в поле, определяемое lpdwDisposition, записывается значение REG_CREATED_NEW_KEY. В том случае, если ключ существовал и был открыт, записываемое значение равно REG_OPENED_EXISTING_KEY. Это поле может быть использовано и для того, чтобы узнать, не открыт ли ключ другим приложением. Открытый ключ доступен только тому приложению, которое создало его. Таким образом, если приложение открывает заведомо существующий ключ и получает в ответ значение REG_CREATED_NEW_KEY, то можно сделать вывод о том, что ключ занят другим приложением.

Функция возвращает значение ERROR_SUCCESS в том случае, если ключ создан или открыт удачно. Любое другое значение является свидетельством того, что при создании или открытии ключа встретилась ошибка.

Итак, считаем, что ключ мы создали. А что необходимо сделать для того, чтобы не создать, а открыть существующий ключ? Для этого нужно вызвать функцию RegOpenKeyEx(), описание которой приведено ниже:

```
WINADVAPI LONG APIENTRY RegOpenKeyExA (HKEY hKey,
                                       LPCSTR lpSubKey,
                                       DWORD ulOptions,
                                       REGSAM samDesired,
                                       PHKEY phkResult);
WINADVAPI LONG APIENTRY RegOpenKeyExW (HKEY hKey,
                                       LPCWSTR lpSubKey,
                                       DWORD ulOptions,
                                       REGSAM samDesired,
                                       PHKEY phkResult);
#ifdef UNICODE
#define RegOpenKeyEx  RegOpenKeyExW
#else
#define RegOpenKeyEx  RegOpenKeyExA
#endif // !UNICODE
```

Надеюсь, что читатель, сравнив описания обеих функций, разберется с аргументами открывающей ключ функции самостоятельно (небольшая подсказка - полю Reserved функции RegCreateKeyEx() соответствует поле ulOptions).

Так, создавать и открывать ключи мы научились. Теперь необходимо научиться ключи закрывать.

ЗАКРЫТИЕ КЛЮЧЕЙ И СОХРАНЕНИЕ ПРОИЗВЕДЕННЫХ В НИХ ИЗМЕНЕНИЙ

Закрывается ключ с помощью функции RegCloseKey(), описание которой, приведенное ниже, можно встретить в файле winreg.h:

WINADVAPI LONG APIENTRY RegCloseKey (HKEY hKey);

Единственным аргументом этой функции является хэндл закрываемого ключа. Но при выполнении этой функции читатель может встретиться с одной проблемой, незаметной с первого взгляда. Дело в том, что данные из реестра на время работы с ними переписываются в кэш и записываются обратно на диск при выполнении функции RegFlushKey(), описание которой имеет следующий вид:

WINADVAPI LONG APIENTRY RegFlushKey (HKEY hKey);

Другими словами, если вы не хотите, чтобы данные, которые вы изменили во время работы программы, были потеряны, перед закрытием ключа сбрасывайте на диск. С другой стороны, у программиста может появиться соблазн сбрасывать данные на диск достаточно часто. Так как RegFlushKey() использует огромное количество системных ресурсов, то эту функцию нужно вызывать только в том случае, когда действительно в этом есть необходимость.

ДОБАВЛЕНИЕ ДАННЫХ К КЛЮЧАМ И УДАЛЕНИЕ ДАННЫХ ИЗ КЛЮЧЕЙ

После того, как ключ создан, возникает необходимость добавить к ключу некоторые данные, которые будут использоваться программой. Для этого нужно вызвать функцию RegSetValueEx(). Описание этой функции, которое приведено ниже, взято из файла winreg.h:

```
WINADVAPI LONG APIENTRY RegSetValueExA (HKEY hKey,
                                        LPCSTR lpValueName,
                                        DWORD Reserved,
                                        DWORD dwType,
                                        CONST BYTE* lpData,
                                        DWORD cbData);
WINADVAPI LONG APIENTRY RegSetValueExW (HKEY hKey,
                                        LPCWSTR lpValueName,
                                        DWORD Reserved,
                                        DWORD dwType,
                                        CONST BYTE* lpData,
                                        DWORD cbData);
#ifdef UNICODE
#define RegSetValueEx  RegSetValueExW
#else
#define RegSetValueEx  RegSetValueExA
#endif // !UNICODE
```

Первый аргумент - хэндл ключа, к которому добавляются данные. Второй аргумент - указатель на строку, содержащую имя добавляемых данных. Третий аргумент зарезервирован. Четвертый аргумент определяет тип информации, который будет сохранен в качестве данных. Этот параметр может принимать одно из значений, приведенных в табл. 52.

Таблица 52. Типы сохраняемой в реестре информации

Параметр	Значение	Описание
REG_NONE	0	Тип данных не устанавливается
REG_SZ	1	Строка, оканчивающаяся нулем
REG_EXPAND_SZ	2	Строка со ссылками на переменные окружения (типа %PATH%)
REG_BINARY	3	Бинарные данные в любой форме
REG_DWORD	4	Двойное слово
REG_LINK	6	Символическая связь
REG_MULTI_SZ	7	Массив из нескольких строк, заканчивающихся нулями, который, в свою очередь, заканчивается двумя нулями
REG_RESOURCE_LIST	8	Список драйверов устройств
REG_FULL_RESOURCE_DESCRIPTOR	9	Список ресурсов в виде частей аппаратуры
REG_RESOURCE_REQUIREMENTS_LIST	10	
REG_DWORD_LITTLE_ENDIAN	4	То же, что и REG_DWORD
REG_DWORD_BIG_ENDIAN	5	То же, что и REG_DWORD, но наиболее значащим в слове является младший байт

Пятый аргумент является указателем непосредственно на данные, которые будут сохранены. И наконец, шестой аргумент определяет размер данных, на которые указывает пятый аргумент. Все легко и просто, не так ли?

А удалить данные можно с помощью обращения к функции RegDeleteValue(). Её описание приведено ниже:

```
WINADVAPI LONG APIENTRY RegDeleteValueA (HKEY hKey,
                                         LPCSTR lpValueName);
WINADVAPI LONG APIENTRY RegDeleteValueW (HKEY hKey,
                                         LPCWSTR lpValueName);
#ifdef UNICODE
#define RegDeleteValue  RegDeleteValueW
#else
```

```
#define RegDeleteValue  RegDeleteValueA
#endif // !UNICODE
```

Аргументы этой функции очевидны - хэндл ключа и указатель на строку с именем данных.

Но если данные записываются в реестр, то, наверное, их можно и нужно считывать из реестра. Поэтому сейчас мы рассмотрим вопрос о том, как происходит

ВЫБОРКА ДАННЫХ ИЗ РЕЕСТРА

Если прикладной программе нужно осуществить выборку данных из реестра, то для начала программа должна определить, из какой ветви дерева регистрации ей нужно выбрать данные. Естественно, что никаких функций для этого нет. При написании программы программист должен сам позаботиться об этом. После того как решение принято, начинается второй этап. Программа должна перебирать все ключи в этой ветви до тех пор, пока не найдет нужный ключ. Для этого приложение может воспользоваться функцией RegEnumKeyEx(). Как и всегда, обратимся к заголовочному файлу winreg.h для того, чтобы найти описание этой функции. Оно приведено ниже:

```
WINADVAPI LONG APIENTRY RegEnumKeyExA (HKEY hKey,
                                       DWORD dwIndex,
                                       LPSTR lpName,
                                       LPDWORD lpcbName,
                                       LPDWORD lpReserved,
                                       LPSTR lpClass,
                                       LPDWORD lpcbClass,
                                       PFLETIME lpftLastWriteTime);
WINADVAPI LONG APIENTRY RegEnumKeyExW (HKEY hKey,
                                       DWORD dwIndex,
                                       LPWSTR lpName,
                                       LPDWORD lpcbName,
                                       LPDWORD lpReserved,
                                       LPWSTR lpClass,
                                       LPDWORD lpcbClass,
                                       PFLETIME lpftLastWriteTime);
#ifdef UNICODE
#define RegEnumKeyEx  RegEnumKeyExW
#else
#define RegEnumKeyEx  RegEnumKeyExA
#endif // !UNICODE
```

Функция перебора объектов нам встречается впервые. Давайте сначала рассмотрим аргументы этой функции, а потом поговорим о том,

что происходит при переборе ключей. Многие аргументы этой функции уже должны быть знакомы читателю. Первый аргумент - это хэндл ключа, подчиненные ключи которого будут перебираться в поисках нужного ключа. Второй аргумент - dwIndex - является индексом требуемого подключа. Третий аргумент - lpName - указывает на буфер, в который будет записано имя ключа. Четвертый аргумент - lpcbName - определяет размер этого буфера в байтах. Пятый аргумент, как следует из его названия - lpReserved - зарезервирован для использования в будущем и должен быть равным NULL. Шестой аргумент - lpClass - должен указывать на буфер, в котором после завершения работы функции будет содержать имя класса подключа. Если это имя программе не требуется, то этот аргумент должен быть равным NULL. Размер этого буфера определяется седьмым аргументом - lpcbClass. И последний, восьмой аргумент - lpftLastWriteTime - после завершения работы функции содержит время последнего обновления данного подключа.

Знать функцию и ее аргументы - это хорошо. Но какой от функции и аргументов прок, если мы не умеем пользоваться функцией? Для того чтобы перебрать подключи, приложение должно сначала вызвать функцию RegEnumKeyEx() со вторым аргументом (dwIndex), равным нулю (поиск начинается с начала дерева). Если искомый ключ найден с первой попытки, то приложению повезло. В противном случае необходимо dwIndex увеличить на единицу и снова обратиться к функции. Так необходимо делать до тех пор, пока не будет найден искомый ключ или функция не вернет значение ERROR_NO_MORE_ITEMS. Естественно, что поиск можно производить и в обратном порядке. Для того чтобы поиск мог быть нормально осуществлен, ключ, хэндл которого указан первым аргументом, должен быть открыт с правом доступа KEY_ENUMERATE_SUB_KEYS. Если функция выполнена успешно, то она возвращает значение ERROR_SUCCESS. Любое другое возвращенное значение является кодом ошибки. Кстати, получить полную информацию о подключе можно с помощью функции RegQueryInfoKey().

Давайте считать, что с помощью способа, описанного выше, мы перебрали подключи и нашли нужный нам подключ. Теперь в этом подключе нам необходимо найти нужные данные. Способ поиска точно такой же, как и в предыдущей функции. Для поиска необходимо перебрать все данные, связанные с подключом. Чтобы произвести этот перебор, обычно используется функция RegEnumValue(), описание которой, приведенное ниже, можно найти в файле winreg.h:

```
WINADVAPI LONG APIENTRY RegEnumValueA (HKEY hKey,
                                       DWORD dwIndex,
```

```
                                                    LPSTR lpValueName,
                                                    LPDWORD lpcbValueName,
                                                    LPDWORD lpReserved,
                                                    LPDWORD lpType,
                                                    LPBYTE lpData,
                                                    LPDWORD lpcbData);
WINADVAPI LONG APIENTRY RegEnumValueW (HKEY hKey,
                                                    DWORD dwIndex,
                                                    LPWSTR lpValueName,
                                                    LPDWORD lpcbValueName,
                                                    LPDWORD lpReserved,
                                                    LPDWORD lpType,
                                                    LPBYTE lpData,
                                                    LPDWORD lpcbData);
#ifdef UNICODE
#define RegEnumValue  RegEnumValueW
#else
#define RegEnumValue  RegEnumValueA
#endif // !UNICODE
```

Порядок использования этой функции полностью совпадает с порядком использования функции RegEnumKeyEx(), поэтому я не стану на нем останавливаться. Опишу только аргументы этой функции. Понятно, что hKey - это хэндл ключа, которому принадлежит подключ, индекс которого представлен вторым аргументом - dwIndex. Следующий аргумент - указатель на буфер, в который будет записано имя подключа. lpcbValueName определяет размер этого буфера. Аргумент lpReserved зарезервирован и должен быть равным NULL. Последние три аргумента определяют класс подключа, указатель на буфер, в который будут записаны эти данные и размер буфера. После возврата функции предпоследний аргумент содержит число записанных данных.

Теперь я, наконец, могу сказать, что у читателя есть полное представление о том, как использовать реестр. Как всегда, рассмотрение темы заканчивается демонстрационной программой:

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

#define hKeyMin 0x80000000
#define hKeyMax 0x80000006
HINSTANCE hInst;
HWND hTreeChild;
TV_INSERTSTRUCT InsertStruct;

LRESULT CALLBACK RegistryWndProc ( HWND, UINT, UINT, LONG );
void FillTree(HWND, HTREEITEM);
```

```
void FillBranch(ULONG, DWORD, HWND, HTREEITEM);
void FillSubBranch(HKEY, char*, HWND, HTREEITEM);

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow )

 {
  HWND hWnd;
  WNDCLASS WndClass ;
  MSG Msg;
  char szClassName[] = "Registry";
  hInst = hInstance;
/* Registering our window class */
/* Fill WNDCLASS structure */

  WndClass.style = CS_HREDRAW | CS_VREDRAW;
  WndClass.lpfnWndProc = RegistryWndProc;
  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 = "";
  WndClass.lpszClassName = szClassName;

  if ( !RegisterClass(&WndClass) )
    {
     MessageBox(NULL,"Cannot register class","Error",MB_OK);
     return 0;
    }

  hWnd = CreateWindow(szClassName, "Registry Demo Program",
                      WS_POPUPWINDOW | WS_VISIBLE | WS_CAPTION,
                      100, 100, 300, 400,
                      NULL, NULL,
                      hInstance,NULL);
  if(!hWnd)
    {
     MessageBox(NULL,"Cannot create window" ,"Error", MB_OK);
     return 0;
    }

  InitCommonControls();
/* Show our window */
  ShowWindow(hWnd, nCmdShow);
  UpdateWindow(hWnd);

/* Beginning of messages cycle */
  while(GetMessage(&Msg, NULL, 0, 0))
```

```
    {
     TranslateMessage(&Msg);
     DispatchMessage(&Msg);
    }
   return Msg.wParam;

}

LRESULT CALLBACK RegistryWndProc (HWND hWnd, UINT Message,
                                  UINT wParam, LONG lParam )
{
 RECT Rect;
 static HWND hTreeChild;
 static HTREEITEM hParentItem;

 switch(Message)
  {
   case WM_CREATE:
    GetClientRect(hWnd, &Rect);
    hTreeChild = CreateWindow(WC_TREEVIEW, "",
                              WS_VISIBLE | WS_TABSTOP | WS_CHILD |
                              TVS_HASLINES | TVS_LINESATROOT |
                              TVS_HASBUTTONS | WS_DLGFRAME,
                              0, 0, Rect.right , Rect.bottom,
                              hWnd,
                              NULL,
                              hInst,
                              NULL);
    InsertStruct.item.mask = TVIF_TEXT;
    InsertStruct.item.hItem = NULL;
    InsertStruct.item.pszText = "Registry Keys";
    InsertStruct.item.cchTextMax = 14;
    InsertStruct.hParent = TVI_ROOT;
    InsertStruct.hInsertAfter = TVI_LAST;
    hParentItem = TreeView_InsertItem(hTreeChild, &InsertStruct);
    FillTree(hTreeChild, hParentItem);
    TreeView_Expand(hTreeChild, hParentItem, TVE_EXPAND);
    TreeView_SelectItem(hTreeChild, hParentItem);
    return 0;
   case WM_SIZE:
    MoveWindow(hTreeChild, 0, 0, LOWORD(lParam), HIWORD(lParam),
               TRUE);
    return 0;
   case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
  }
 return DefWindowProc(hWnd, Message, wParam, lParam);
}
```

```
void FillTree(HWND hTreeWnd, HTREEITEM hParentItem)
 {
  ULONG i;
  TV_ITEM Item;
  TV_INSERTSTRUCT InsertStruct;
  LPSTR lpszKeys[7] = {"HKEY_CLASSES_ROOT",
                       "HKEY_CURRENT_USER",
                       "HKEY_LOCAL_MACHINE",
                       "HKEY_USERS",
                       "HKEY_PERFORMANCE_DATA",
                       "HKEY_CURRENT_CONFIG",
                       "HKEY_DYN_DATA"};
  char cClass[80] = "";
  DWORD dwSize = 80, dwSubKeys, dwMaxLength, dwMaxClass, dwValues,
        dwMaxValue, dwMaxData, dwSec;
  FILETIME ftFileTime;
  HTREEITEM hNewParentItem;

  for(i = hKeyMin; i <= hKeyMax; i++)
   {
// Add the highest items
    if(ERROR_SUCCESS == RegQueryInfoKey((HKEY) i, cClass, &dwSize,
                                        NULL,
                                        &dwSubKeys, &dwMaxLength,
                                        &dwMaxClass, &dwValues,
                                        &dwMaxValue, &dwMaxData,
                                        &dwSec, &ftFileTime))
      {
       Item.mask = TVIF_TEXT;
       Item.pszText = lpszKeys[i - hKeyMin];
       InsertStruct.item = Item;
       InsertStruct.hParent = hParentItem;
       hNewParentItem = TreeView_InsertItem(hTreeWnd, &InsertStruct);
       FillBranch(i, dwSubKeys, hTreeWnd, hNewParentItem);
      }
   }
 }

void FillBranch(ULONG i, DWORD dwSubKeys, HWND hTreeWnd,
                HTREEITEM hNewParentItem)
 {
  int j;
  DWORD dwClassNameSize = 80;
  char cClassName[80] = "";

  if (dwSubKeys == 0)
    return;
   else
    {
     for(j = 0; j < dwSubKeys; j++)
```

```
        {
         RegEnumKey((HKEY) i, (DWORD) j, cClassName, dwClassNameSize);
         dwClassNameSize = 80;
         FillSubBranch((HKEY) i, cClassName, hTreeWnd, hNewParentItem);
        }
      }
  }

void FillSubBranch(HKEY hKey, char* cClassName, HWND hTreeWnd,
                   HTREEITEM hParentItem)
 {
  HKEY hNewKey;
  char cClass[80], cNewClass[80];
  DWORD dwClassSize = 80, dwSK, j;
  TV_ITEM Item;
  TV_INSERTSTRUCT InsertStruct;
  HTREEITEM hNewParentItem;

  Item.mask = TVIF_TEXT;
  Item.pszText = cClassName;
  InsertStruct.hParent = hParentItem;
  InsertStruct.hInsertAfter = TVI_SORT;
  InsertStruct.item = Item;
  hNewParentItem = TreeView_InsertItem(hTreeWnd, &InsertStruct);
  RegOpenKey(hKey, cClassName, &hNewKey);
  RegQueryInfoKey(hNewKey, cClass, &dwClassSize, NULL, &dwSK, NULL,
                  NULL, NULL, NULL, NULL, NULL, NULL);
  dwClassSize = 80;
  if(dwSK != 0)
    for(j = 0; j < dwSK; j++)
     {
       RegEnumKey(hNewKey, j, cNewClass, dwClassSize);
       FillSubBranch(hNewKey, cNewClass, hTreeWnd, hNewParentItem);
     }
  RegCloseKey(hNewKey);
 }
```

Вид, создаваемого программой окна, показан на рис. 20.

Я не большой специалист в рисовании (☺), поэтому подключение изображений, как всегда, оставляю на долю читателя.

Эта программа просто перебирает ключи и позволяет просмотреть все «дерево» реестра. Обращаю внимание читателя на тот факт, что предопределенные ключи (их имена начинаются с HKEY_) всегда открыты. Открывать следует только ключи, находящиеся ниже предопределенных в иерархии. Если читатель будет разрабатывать программу, хранящую конфигурационные данные на диске, я настоятельно рекомендую использовать реестр, а не пользоваться произвольными файлами.

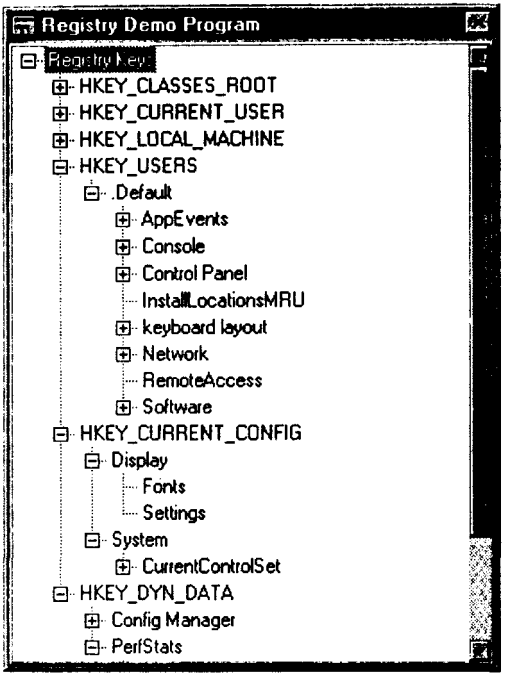


Рис. 20. Окно с деревом реестра, созданное программой

КОЕ-ЧТО О МНОГОЗАДАЧНОСТИ В WINDOWS

Одним из основных отличий Win32 от его предшественников явилась многозадачность. "Как это - разве в Windows 3.x не была реализована истинная многозадачность?" - может спросить кто-нибудь из неискушенных пользователей. "НЕ БЫЛА!" - отвечу я. И вот почему.

Мне Windows 3.x представляется чем-то вроде однорукого натуралиста. У этого натуралиста есть зверинец, в каждой клетке которого сидит хищник - программа. Каждому хищнику (программе) натуралист подает корм (сообщение) рукой. Как только хищник (программа) съест корм (обработает сообщение), корм (сообщение) получает очередной хищник. И так далее по кругу. Но иногда один из хищников (программ) мертвой хваткой вцепляется в руку (зависает и не возвращает управление) и натуралист умирает (Windows зависает), после чего на смену умершему смотрителю зверинца приходит новый однорукий натуралист (производится перезагрузка системы). И все начинается сначала. Разве

вам не знакома эта ситуация, уважаемый читатель? В Windows 3.x была реализована ПСЕВДОМНОГОЗАДАЧНОСТЬ, т. е. управление передавалось программе, возвращалось системе и передавалось следующей программе, т. е. фактически программы работали последовательно, друг за другом. В случае зависания одной из задач средств завершить ее, не перезагружая систему, практически не было. Кроме этого, фактически все задачи разделяли одни и те же системные ресурсы, например, память. Не было никаких проблем для одной задачи затереть содержимое памяти, выделенной другой.

В Windows'95 и Windows NT дело обстоит не так. В этой системе реализована истинная многозадачность, т. е. каждой программе системой выделяется квант времени, в течение которого программа обрабатывает поступившие в ее адрес сообщения. Вне зависимости от состояния программы СИСТЕМА забирает управление у программы и передает его другой программе. Если программа зависла, то система от этого не пострадает. Управление в любом случае будет передано другой программе. Кроме этого, в Windows'95 и Windows NT введено понятие процесса. Грубо говоря, процесс - это совокупность выполняющейся программы и выделенных ей системных ресурсов. Случай, при котором программа может вырваться из рамок своего процесса и повредить еще чьи-то ресурсы, практически не возможен.

Рассуждаем дальше. Раз программа получает управление на время, то почему бы этой программе не распараллелить свою работу и не запустить несколько одновременно выполняющихся программ под своим управлением? В Win32 эти программы называются потоками.

Таким образом, в системе Windows реализованы два типа многозадачности - процессная и потоковая. Рассмотрим оба типа многозадачности, после чего отдельно остановимся на вопросе синхронизации работы в многозадачной среде.

Остановимся на одной детали. Оттого, что мы назвали Windows многозадачной системой, физический смысл этой многозадачности не изменился. На однопроцессорном компьютере в каждый конкретный момент выполняется одна задача. Если при запуске двух-трех маленьких программ временная задержка субъективно не заметна, то при запуске нескольких программ, требующих колоссальных ресурсов (к примеру, WinWord или Borland C++ 5.0), задержка при выполнении программ становится достаточно заметной. На многопроцессорных системах за каждым процессором может быть закреплен свой поток, поэтому на таких системах выполнение программ осуществляется действительно в многозадачном режиме.

ЗАПУСК ПРОЦЕССА

Давайте, уважаемый читатель, все же более точно определим, что есть процесс. В Windows 3.x, да иногда и в Win32 процесс определяют как копию (экземпляр) выполняющейся программы. Так оно и есть, но при этом забывают, что копия - понятие статическое. Другими словами, процесс в Win32 - это объект, который не выполняется, а просто «владеет» выделенным ей адресным пространством, другими словами, процесс является структурой в памяти. А вот в адресном пространстве процесса находятся не только код и данные, но и потоки - выполняющиеся объекты. При запуске процесса автоматически запускается поток (он называется главным). При остановке главного потока автоматически останавливается и процесс. А так как процесс без потока просто бесцельно занимает ресурсы, то система автоматически уничтожает ставший ненужным процесс. Первичный процесс создается системой при запуске, точно так же при создании первичного процесса в нем создается и поток.

Приложение тоже может создать процесс с главным потоком, используя для этой цели функцию CreateProcess(). Её прототип, находящийся в файле winbase.h, при первой встрече с ним внушает легкий ужас:

```
WINBASEAPI BOOL WINAPI CreateProcessA(LPCSTR lpApplicationName,
                    LPSTR lpCommandLine,
                    LPSECURITY_ATTRIBUTES lpProcessAttributes,
                    LPSECURITY_ATTRIBUTES lpThreadAttributes,
                    BOOL bInheritHandles,
                    DWORD dwCreationFlags,
                    LPVOID lpEnvironment,
                    LPCSTR lpCurrentDirectory,
                    LPSTARTUPINFOA lpStartupInfo,
                    LPPROCESS_INFORMATION lpProcessInformation);
WINBASEAPI BOOL WINAPI CreateProcessW(LPCWSTR lpApplicationName,
                    LPWSTR lpCommandLine,
                    LPSECURITY_ATTRIBUTES lpProcessAttributes,
                    LPSECURITY_ATTRIBUTES lpThreadAttributes,
                    BOOL bInheritHandles,
                    DWORD dwCreationFlags,
                    LPVOID lpEnvironment,
                    LPCWSTR lpCurrentDirectory,
                    LPSTARTUPINFOW lpStartupInfo,
                    LPPROCESS_INFORMATION lpProcessInformation);
#ifdef UNICODE
#define CreateProcess  CreateProcessW
#else
#define CreateProcess  CreateProcessA
#endif // !UNICODE
```

Так как понимание сущности процессов и потоков крайне важно для программирующих в Win32, в этом месте я чуть отступлю от принятого стиля изложения и более подробно расскажу о том, что происходит при вызове этой функции.

Я уже говорил, что процесс - это структура в памяти. Таким образом, в начале работы функция выделяет память для этой структуры, а потом выделяет память (виртуальную, естественно) для адресного пространства процесса. Если выделение памяти прошло без ошибок, в адресное пространство процесса загружается код исполняемой программы и используемых программой динамических библиотек. Только после этого создается главный поток процесса. Если функции удастся произвести все эти действия без ошибок, то возвращаемое значение будет равно TRUE. FALSE явится индикатором того, что по каким-то причинам процесс не создан.

Перейдем к рассмотрению аргументов функции CreateProcess().

АРГУМЕНТЫ ФУНКЦИИ CREATEPROCESS()

С моей точки зрения, взаимодействие первых двух аргументов не совсем продумано.

Первый аргумент - lpApplicationName - определяет имя исполняемого файла (обязательно указывать имя и расширение файла, автоматически расширение .exe не подставляется), для которого создается процесс.

Второй аргумент - lpCommandLine определяет передаваемую этому файлу командную строку. Если lpApplicationName равен NULL, то первый (до первого пробела) элемент lpCommandLine считается именем исполняемого файла.

Таким образом, имя исполняемого файла можно передавать как в первом, так и во втором аргументе. Но здесь нужно быть внимательным и не допустить, скажем, такой ситуации, когда lpApplicationName равен «Wordpad.exe», а lpCommandLine - «Wordpad.exe MyFile.doc». Нетрудно догадаться к чему это приведет.

Третий и четвертый аргументы определяют атрибуты доступа к процессу и потоку соответственно. Я намеренно употребил слово «должны», ибо Windows'95 системы разграничения доступа не имеет. В Windows'95 эти значения, как правило, равны NULL.

Очередное поле - dwCreationFlag - является комбинацией битовых флагов. Одна группа битовых флагов определяет способ создания процесса.

Флаги способа создания процесса

Флаг DEBUG_PROCESS (0x00000001) устанавливается в тех случаях, когда родительский процесс должен осуществлять отладку порождаемого процесса и всех его потомков. Система будет оповещать родительский процесс о возникновении определенных событий в порождаемом процессе и его потомках.

Флаг DEBUG_ONLY_THIS_PROCESS (0x00000002) почти эквивалентен предыдущему, разница состоит в том, что система будет оповещать о событиях только в порождаемом процессе, но не в его потомках.

Флаг CREATE_SUSPENDED (0x00000004) указывает, что главный поток порождаемого процесса создается, но не выполняется до вызова функции ResumeThread(). Этот флаг обычно используется в отладчиках.

Флаг DETACHED_PROCESS (0x00000008) запрещает создаваемому консольному процессу использовать консоль родительского процесса. Порождаемый процесс вынужден будет вызвать функцию AllocConsole() для получения собственной консоли.

Флаг CREATE_NEW_CONSOLE (0x00000010) указывает на необходимость создания новой консоли для порождаемого процесса. Этот флаг не может использоваться вместе с предыдущим.

Флаг CREATE_NEW_PROCESS_GROUP (0x00000200) создает группу консольных процессов, которые будут одновременно реагировать на нажатие клавиш Ctrl-C и Ctrl-Break.

Флаг CREATE_UNICODE_ENVIRONMENT (0x00000400) означает, что данные, на которые указывает lpEnvironment, используют символы Unicode. По умолчанию считается, что используется ANSI-кодировка.

Флаг CREATE_SEPARATE_WOW_VDM (0x00000800) используется только при запуске 16-битовых Windows-приложений и указывает, что приложению необходимо выделить отдельную виртуальную машину (Virtual DOS Machine, VDM) (по умолчанию, все 16-битовые Windows-приложения используют одну разделяемую виртуальную машину). Преимуществом выделения отдельной машины является то, что приложение почти не влияет на остальные. Даже зависнув, оно приведет к краху только своей VDM. Недостаток - каждая виртуальная машина требует большого объема памяти.

Флаг CREATE_SHARED_WOW_VDM (0x00001000) используется при запуске 16-битовых Windows-приложений и указывает на необходимость создания для процесса разделяемой VDM.

Таблица 53. Флаги класса приоритета процесса

Флаг	Значение	Эффект
NORMAL_PRIORITY_CLASS	0x00000020	Нормальный приоритет
IDLE_PRIORITY_CLASS	0x00000040	Потоки этого процесса выполняются только тогда, когда система простаивает
HIGH_PRIORITY_CLASS	0x00000080	Приоритет выше нормального, но ниже приоритета реального времени
REALTIME_PRIORITY_CLASS	0x00000100	Самый высокий возможный приоритет

Флаг CREATE_DEFAULT_ERROR_MODE (0x040000) указывает, что порождаемый процесс не наследует режим обработки ошибок своего родителя. Ему при создании устанавливается режим обработки ошибок, принятый по умолчанию.

На этом флаги, определяющие способ создания процесса, исчерпаны. Вторая группа битовых флагов определяет класс приоритета создаваемого процесса.

Флаги класса приоритета процесса

При создании процесса можно указать и класс его приоритета (табл. 53). Если при создании процесса не указан ни один из флагов, приведенных ниже, класс приоритета порождаемого процесса по умолчанию устанавливается равным IDLE_PRIORITY_CLASS, если этот класс установлен у процесса родителя, и NORMAL_PRIORITY_CLASS во всех остальных случаях.

Тем не менее, присвоение класса приоритета вновь создаваемому потоку не рекомендуется - Windows сама присвоит потоку класс приоритета по умолчанию.

Следующий аргумент функции CreateProcess() - lpEnvironment - обычно равен NULL. Это означает, что порождаемый процесс наследует переменные окружения родительского процесса. Если этот аргумент не равен NULL, то он должен содержать указатель на блок памяти, содержащий те переменные окружения, которыми будет пользоваться порождаемый процесс.

Наименование следующего аргумента функции - lpCurrentDirectory - говорит само за себя. Этот аргумент позволяет установить текущие диск и директорию для порождаемого процесса. Если этот аргумент равен NULL, порождаемый процесс наследует текущие диск и директорию

родительского процесса. В противном случае этот аргумент должен указывать на строку, в которой указан полный путь к устанавливаемой текущей директории, включающий и букву дисковода.

Очередной аргумент - указатель на структуру типа STARTUPINFO. Эта структура, служащая для описания свойств окна, создаваемого в новом процессе, описана в winbase.h следующим образом:

```
typedef struct _STARTUPINFOA {DWORD  cb; LPSTR  lpReserved;
                              LPSTR  lpDesktop; LPSTR  lpTitle;
                              DWORD  dwX; DWORD  dwY;
                              DWORD  dwXSize; DWORD  dwYSize;
                              DWORD  dwXCountChars;
                              DWORD  dwYCountChars;
                              DWORD  dwFillAttribute;
                              DWORD  dwFlags;
                              WORD   wShowWindow;
                              WORD   cbReserved2;
                              LPBYTE lpReserved2;
                              HANDLE hStdInput;
                              HANDLE hStdOutput;
                              HANDLE hStdError;
} STARTUPINFOA, *LPSTARTUPINFOA;
typedef struct _STARTUPINFOW {DWORD  cb; LPWSTR  lpReserved;
                              LPWSTR  lpDesktop; LPWSTR  lpTitle;
                              DWORD  dwX; DWORD  dwY;
                              DWORD  dwXSize; DWORD  dwYSize;
                              DWORD  dwXCountChars;
                              DWORD  dwYCountChars;
                              DWORD  dwFillAttribute;
                              DWORD  dwFlags;
                              WORD   wShowWindow;
                              WORD   cbReserved2;
                              LPBYTE lpReserved2;
                              HANDLE hStdInput;
                              HANDLE hStdOutput;
                              HANDLE hStdError;
} STARTUPINFOW, *LPSTARTUPINFOW;
#ifdef UNICODE
typedef STARTUPINFOW STARTUPINFO;
typedef LPSTARTUPINFOW LPSTARTUPINFO;
#else
typedef STARTUPINFOA STARTUPINFO;
typedef LPSTARTUPINFOA LPSTARTUPINFO;
#endif // UNICODE
```

Рассмотрим поля этой структуры.

Поля структуры типа STARTUPINFO

Первое поле - cb - размер этой структуры. Оно должно быть равно sizeof(STARTUPINFO).

Второе поле - lpReserved - зарезервировано и должно быть равно NULL.

Третье поле - lpDesktop - в Windows'95 просто игнорируется.

Четвертое поле - lpTitle - определяет заголовок консольного приложения. Для GUI или приложений, не создающих новой консоли, должен быть равным NULL.

Поля с пятого по восьмое включительно определяют положение окна и его размеры (dwX, dwY - координаты верхнего левого угла окна в пикселах, dwXSize, dwYSize - ширина и высота окна в пикселах).

Таблица 54. Флаги, определяющие, в каких полях структуры типа STARTUPINFO содержится информация

Флаг	Значение	Эффект
STARTF_USESHOWWINDOW	0x00000001	Если флаг не установлен, поле wShowWindow игнорируется
STARTF_USESIZE	0x00000002	Если флаг не установлен, поля dwXSize и dwYSize игнорируются
STARTF_USEPOSITION	0x00000004	Если флаг не установлен, поля dwX и dwY игнорируются
STARTF_USECOUNTCHARS	0x00000008	Если флаг не установлен, поля dwXCountChars и dwYCounChars игнорируются
STARTF_USEFILLATTRIBUTE	0x00000010	Если флаг не установлен, поле dwFillAttribute игнорируется
STARTF_RUNFULLSCREEN	0x00000020	
STARTF_FORCEONFEEDBACK	0x00000040	Курсор становится «песочными часами» на две секунды, за которые должно произойти обращение к GUI, после чего за 5 секунд должно быть создано окно и еще за 5 секунд оно должно перерисоваться
STARTF_FORCEOFFFEEDBACK	0x00000080	При создании процесса форма курсора не меняется
STARTF_USESTDHANDLES	0x00000100	Если флаг установлен, то используются потоки, хэндлы которых определены полями hStdInput, hStdOutput, hStdError
STARTF_USEHOTKEY	0x00000200	

Девятое и десятое поля (dwXCountChars, dwYCountChars) определяют ширину и высоту окна консоли в символах (не пикселах!). Одиннадцатое поле - dwFillAttribute - определяет атрибуты консольного окна.

Двенадцатое поле - dwFlags - используется для того, чтобы определить, какие поля структуры типа STARTUPINFO будут использоваться при создании окна порождаемым процессом. Это поле представляет собой комбинацию битовых флагов (табл. 54).

Тринадцатое поле - wShowWindow - определяет, каким образом окно будет отображено (помните функцию ShowWindow())? Значение этого поля игнорируется, если только в dwFlags не установлен флаг STARTF_USESHOWWINDOW. Возможные значения этого поля - те же константы, начинающиеся с SW_, которые используются в функции ShowWindow().

Четырнадцатое и пятнадцатое поля, - cbReserved2 и lpReserved2 зарезервированы. Должны инициализироваться нулем и NULL соответственно.

Шестнадцатое, семнадцатое и восемнадцатое поля - hStdInput, hStdOutput и hStdError - определяют хэндлы стандартных потоков ввода-вывода.

Нерассмотренным остался только один аргумент функции CreateProcess() - lpProcessInformation, указывающий на структуру типа PROCESS_INFORMATION, в которую записывается информация о порожденном процессе после его создания. Структура описана в файле winbase.h следующим образом:

```
typedef struct _PROCESS_INFORMATION
 {
   HANDLE hProcess;
   HANDLE hThread;
   DWORD dwProcessId;
   DWORD dwThreadId;
}
 PROCESS_INFORMATION, *PPROCESS_INFORMATION,
  *LPPROCESS_INFORMATION;
```

В первое поле - hProcess - система записывает хэндл созданного процесса, во второе - hThread - хэндл потока. Поля dwProcessId и dwThreadId являются уникальными идентификаторами процесса и потока соответственно. Рекомендую обратить особое внимание на последние два поля. Дело в том, что Win32, если идентификатор освобожден, может повторно использовать его. К примеру, пусть процессу присвоен идентификатор 0x00001111. После завершения процесса идентификатор освобождается и какому-нибудь новому процессу может опять быть присвоен тот же

идентификатор 0x00001111. Это необходимо учитывать при написании программ.

Итак, аргументы функции рассмотрены. Основные результаты - хэндлы и идентификаторы процесса и потока - получены. А какое значение возвращает функция? Возвращаемое функцией значение TRUE говорит о том, что процесс создан и функция завершилась нормально. А при получении значения FALSE программисту придется искать ошибку.

Итак, мы подробно рассмотрели вопрос о запуске процесса. Теперь, очевидно, процесс необходимо завершить.

ЗАВЕРШЕНИЕ ПРОЦЕССА

Процесс может быть завершен вызовом одной из двух функций - ExitProcess() или TerminateProcess(). Рассмотрим более подробно каждую из этих функций.

ФУНКЦИЯ EXITPROCESS()

В обычных условиях процесс завершается тогда, когда один из принадлежащих ему потоков вызывает функцию ExitProcess(), которая описана в файле winbase.h следующим образом:

```
WINBASEAPI VOID WINAPI ExitProcess(UINT uExitCode);
```

Читателю следует обратить внимание на тот факт, что завершение процесса начинается изнутри процесса. Почему так сделано? Во-первых, только поток процесса знает, когда он выполнил свою работу и когда ему необходимо завершиться. Во-вторых, только процесс в тот момент, когда он узнает о необходимости завершения, может оповестить об этом все принадлежащие ему потоки и произвести нормальное завершение. Извне процесса эти действия произвести почти невозможно.

Если говорить более конкретно, то при завершении процесса производятся следующие действия:

- вызываются функции деинициализации всех подключенных библиотек динамической компоновки, т. е. происходит нормальное завершение всех подключенных DLL;
- закрываются и/или уничтожаются все объекты, открытые и/или созданные процессом;
- состояние процесса изменяется на «освобожденный» (signaled), что является сигналом для всех потоков, ожидающих завершения процесса;
- состояние всех потоков изменяется на «освобожденный» (signaled), что является сигналом для всех потоков других процессов, которые ожидают завершения потоков текущего процесса;

- код завершения меняется со STILL_ACTIVE на код, записываемый в uExitCode;
- счетчик числа пользователей процесса уменьшается на единицу (заметим, что данные процесса удаляются из памяти, но сам объект остается в памяти до того момента, пока счетчик пользователей не достигнет нулевого значения, или, другими словами, пока не будут закрыты все хэндлы процесса. Определить, завершен ли процесс можно с помощью функции GetExitProcessCode(), которая в случае незавершенности процесса возвращает STILL_ACTIVE).

Необходимо отметить, что завершение процесса не приводит к завершению порожденных им процессов.

Сразу после деинициализации и выгрузки библиотек из памяти, но до своего завершения, функция заносит в параметр uExitCode код завершения. После этого процесс можно считать полностью завершенным.

ФУНКЦИЯ TERMINATEPROCESS()

Эта функция является аварийным средством завершения процесса и её рекомендуется использовать только в крайнем случае. Она описана в том же winbase.h:

```
WINBASEAPI BOOL WINAPI TerminateProcess(HANDLE hProcess,
                                        UINT uExitCode);
```

Функция используется только тогда, когда иными средствами завершить процесс не удается. С этой целью извне (!), а не изнутри процесса вызывается функция TerminateProcess(), которая и завершает процесс. Но в данном случае не освобождаются используемые процессом DLL, хотя все используемые объекты освобождаются. Освобождается также и память, занимаемая процессом. Число пользователей процесса также уменьшается.

Отметим один интересный факт. Обычно один процесс запускает другой как обособленный и после запуска забывает о нем. Для того чтобы порожденный процесс мог быть завершен, сразу после создания процесса порождающий процесс должен закрыть хэндл порожденного процесса и его потока. Делается это примерно следующим образом:

```
PROCESS_INFORMATION ProcessInformation;
BOOL hMyProcess;
if ( (hMyProcess = CreateProcess( ....., &ProcessInformation) )
   {
    CloseHandle(ProcessInformation.hThread);
    CloseHandle(ProcessInformation.hProcess);
   }
```

О процессах можно рассказать намного больше, но надеюсь, что написанного хватит для того, чтобы приступить к программированию. Для того чтобы проиллюстрировать все эти длинные рассуждения, приведу в качестве примера небольшую программу.

ДЕМОНСТРАЦИОННАЯ ПРОГРАММА

В этой программе не происходит ничего интересного. Просто при выборе элемента меню «Создать процесс» создается процесс, в котором запускается обычный Notepad (надеюсь, он у всех в доступной директории? Если нет, то вы можете заменить Notepad любой другой программой). Максимум может быть запущено 10 процессов. По команде «Kill process» процессы уничтожаются в порядке, обратном их созданию. Предлагаю читателю обратить внимание на то, что я завершаю процесс посредством вызова TerminateProcess(), а не ExitProcess(). Для того чтобы завершить процесс обычным способом, пришлось бы писать программу, которая вызывала бы функцию ExitProcess() изнутри процесса, а мне бы не хотелось рассеивать внимание читателя. Результаты создания процесса, взятые из структуры типа PROCESS_INFORMATION, отображаются в окне сообщений. Если кого-то раздражает необходимость постоянно убирать окно сообщений с отображения, рекомендую воспользоваться программой pview95.exe, которая поставляется с SDK.

Текст демонстрационной программы приведен ниже:

```
#include <windows.h>
#include <stdio.h>
#include "proc.h"

LRESULT CALLBACK ProcessesWndProc ( HWND, UINT, UINT, LONG );

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow )

 {
  HWND hWnd ;
  WNDCLASS WndClass ;
  MSG Msg;
  char szClassName[] = "Processes";
/* Registering our window class */
/* Fill WNDCLASS structure */

  WndClass.style = CS_HREDRAW | CS_VREDRAW;
  WndClass.lpfnWndProc = ProcessesWndProc;
  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 = "ProcessesMenu";
  WndClass.lpszClassName = szClassName;

  if ( !RegisterClass(&WndClass) )
    {
     MessageBox(NULL,"Cannot register class","Error",MB_OK);
     return 0;
    }

  hWnd = CreateWindow(szClassName, "Processes Demo",
                      WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                      CW_USEDEFAULT, CW_USEDEFAULT,
                      CW_USEDEFAULT, NULL, NULL,
                      hInstance,NULL);
  if(!hWnd)
    {
     MessageBox(NULL,"Cannot create window","Error",MB_OK);
     return 0;
    }

  /* Show our window */
  ShowWindow(hWnd,nCmdShow);
  UpdateWindow(hWnd);

/* Beginning of messages cycle */
  while(GetMessage(&Msg, NULL, 0, 0))
   {
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
   }
  return Msg.wParam;
 }

LRESULT CALLBACK ProcessesWndProc (HWND hWnd, UINT Message,
                                   UINT wParam, LONG lParam )
 {
  const Max = 10;
  STARTUPINFO StartupInfo;
  static int ProcessNumber = 0;
  static PROCESS_INFORMATION ProcessInformation[Max];

  static char cMyMessage[80];
  static HMENU hSubMenu;

  switch(Message)
   {
    case WM_CREATE:
     hSubMenu = GetSubMenu(GetMenu(hWnd),0);
```

```
 return 0;
case WM_COMMAND:
 switch( LOWORD(wParam) )
  {
   case IDM_New_Process:
    if(ProcessNumber < Max)
      {
       StartupInfo.cb = sizeof(STARTUPINFO);
       StartupInfo.lpReserved = NULL;
       StartupInfo.lpDesktop = NULL;
       StartupInfo.lpTitle = NULL;
       StartupInfo.dwFlags = STARTF_USESHOWWINDOW;
       StartupInfo.wShowWindow = SW_SHOWNORMAL;
       StartupInfo.cbReserved2 = 0;
       StartupInfo.lpReserved2 = NULL;
       if(CreateProcess(NULL,"Notepad.exe",
                        NULL, NULL, FALSE, 0,
                        NULL, NULL, &StartupInfo,
                        &(ProcessInformation[ProcessNumber])))
         {
          ProcessNumber++;
          wsprintf(cMyMessage,"hProcess is %x.\nhThread is
                   %x.\ndwProcessId is %x.\ndwThreadId is %x.",
                   ProcessInformation[ProcessNumber - 1].hProcess,
                   ProcessInformation[ProcessNumber - 1].hThread,
                   ProcessInformation[ProcessNumber - 1].dwProcessId,
                   ProcessInformation[ProcessNumber - 1].dwThreadId);
          MessageBox(hWnd, cMyMessage, "Process is created", MB_OK);
          EnableMenuItem(hSubMenu, IDM_Kill_Process,
                         MF_BYCOMMAND | MF_ENABLED);
         }
        else
         MessageBox(hWnd, "Cannot create process", "Process creation", MB_OK);
      }
     else
      {
       MessageBox(hWnd, "Too many created processes...",
                  "Process creation", MB_OK);
      }
    break;
   case IDM_Kill_Process:
    if(ProcessNumber > 0)
      {
       if(TerminateProcess(ProcessInformation[ProcessNumber-1].hProcess,  0))
         {
          ProcessNumber--;
          if(!ProcessNumber)
           EnableMenuItem(hSubMenu, IDM_Kill_Process,
                          MF_BYCOMMAND | MF_GRAYED);
         }
```

```
            else
              MessageBox(hWnd, "Cannot terminate process",
                         "Process termination", MB_OK);
          }
          else
            MessageBox(hWnd, "No more processes", "Process termination",
                       MB_OK);
          break;
        case IDM_Exit:
          SendMessage(hWnd, WM_CLOSE, 0, 0);
          break;
        }
      return 0;
    case WM_DESTROY:
      PostQuitMessage(0);
      return 0;
    }
  return DefWindowProc(hWnd,Message,wParam, lParam);
}
```

В этой программе используется файл описаний:

```
#define IDM_About          104
#define IDM_Exit           103
#define IDM_Kill_Process   102
#define IDM_New_Process    101
```

Кроме этого, в программе используется файл ресурсов:

```
#include "proc.h"

ProcessesMenu MENU
{
 POPUP "&Processes"
 {
 MENUITEM "&New process", IDM_New_Process
 MENUITEM "&Kill process", IDM_Kill_Process, GRAYED
 MENUITEM SEPARATOR
 MENUITEM "E&xit", IDM_Exit
 }

 POPUP "&Help"
 {
 MENUITEM "&About", IDM_About
 }

}
```

СОЗДАНИЕ ПОТОКА

Создание потока в большей степени (внешне, конечно) напоминает программу для Windows, чем создание процесса. Дело в том, что для создания потока используется функция CreateThread() (аналог WinMain()), одним из аргументов которой является указатель на функцию потока (аналог оконной функции). Но давайте обо всем по порядку.

Итак, начнем по уже сложившейся традиции, с прототипа функции. Она описана в файле winbase.h:

```
WINBASEAPI HANDLE WINAPI CreateThread(
                    LPSECURITY_ATTRIBUTES lpThreadAttributes,
                    DWORD dwStackSize,
                    LPTHREAD_START_ROUTINE lpStartAddress,
                    LPVOID lpParameter,
                    DWORD dwCreationFlags,
                    LPDWORD lpThreadId);
```

При вызове этой функции происходит следующее:

в памяти создаются все необходимые для управления потоком структуры (назовем их объектом «поток»);

код завершения потока инициализируется значением STILL_ACTIVE;

создается структура типа CONTEXT для потока (к сожалению, я не могу описать структуру в рамках этой книги - она слишком велика, но рекомендую читателю самостоятельно разобраться с ней по заголовочным файлам и файлам системы помощи);

создается стек потока;

инициализируется регистр - указатель стека в структуре типа CONTEXT так, чтобы он указывал на верхнюю границу стека, а регистр - указатель команд - на точку входа функции потока.

Рассмотрим аргументы этой функции.

АРГУМЕНТЫ ФУНКЦИИ CREATETHREAD()

Первый аргумент - lpThreadAttributes - является указателем на структуру типа SECURITY_ATTRIBUTES. Так как в Windows'95 атрибуты безопасности не используются, то обычно этот аргумент равен NULL.

Второй аргумент - dwStackSize - определяет размер выделяемого потоку стека. Если в качестве этого параметра указан 0, то поток будет иметь стек такого же размера, как и у породившего его потока.

Третий аргумент этой функции - lpStartAddress - собственно и определяет поток, так как является адресом точки входа функции потока. Функ-

ция потока может иметь имя, определяемое программистом, но должна иметь следующий прототип:

```
DWORD WINAPI ThreadFunction(LPVOID lpParameter);
```

Я не случайно дал аргументу этой функции и четвертому аргументу функции CreateThread() одинаковые имена. Четвертый аргумент функции CreateThread() - это параметр, передаваемый функции потока. Что и каким образом передается в этом параметре, совершенно неважно. Это могут быть всевозможные данные, которые функция потока может использовать для своей работы.

Если следующий аргумент - dwCreationFlags - равен нулю, то выполнение потока начнется немедленно. Если этот аргумент будет равен CREATE_SUSPENDED, то начало выполнение потока будет задержано до определенных событий, например, до вызова функции ResumeThread().

И наконец, в значение, определяемое последним аргументом, lpThreadId, записывается идентификатор созданного потока. А значение, возвращаемое функцией, является хэндлом этого потока.

Раз у потока есть начало, то должно быть и

ЗАВЕРШЕНИЕ ПОТОКА

Как и процесс, поток может быть завершен двумя способами - вызовом функции ExitThread() и обращением к функции TerminateThread(). Отличаются они друг от друга примерно тем же, что и функции ExitProcess() и TerminateProcess(). Первая функция, ExitThread(), используется для нормального завершения потока. Естественно, что она вызывается изнутри потока. Она описана в файле winbase.h:

```
WINBASEAPI VOID WINAPI ExitThread(DWORD dwExitCode);
```

Единственным ее аргументом является двойное слово, в которое будет помещен код возврата этой функции.

Функцию TerminateProcess(), описанную в том же файле winbase.h следующим образом,

```
WINBASEAPI BOOL WINAPI TerminateThread(HANDLE hThread,
                                       DWORD dwExitCode);
```

следует вызывать только в крайних случаях, когда поток завис, и ни на какие действия пользователя не реагирует. Функция вызывается из какого-либо внешнего (по отношению к завершаемому) потока, а ее

аргументами являются хэндл завершаемого потока и двойное слово, в которое будет записан код завершения потока.

Осталось только узнать, что происходит при завершении потока. Во-первых, освобождаются или удаляются все занятые или созданные объекты. Это действие является стандартным и ничего особенного собой не представляет. Во-вторых, поток получает статус незанятого (signaled). В-третьих, код завершения процесса меняется со STILL_ACTIVE на указанный при вызове завершающей поток функции. В-четвертых, уменьшается счетчик пользователей потока. Если пользователей потока больше не осталось, и поток является единственным потоком процесса, то завершается и процесс. Все легко, просто и логично.

СИНХРОНИЗАЦИЯ

К этому моменту читатель уже знает, что в программе один поток, главный, запускается автоматически при запуске. Следовательно, создавая новые потоки, мы тем самым делаем программу многопотоковой. Хорошо, конечно, если эти потоки работают независимо. А как быть тем потокам, которые зависят друг от друга? Например, осуществляют доступ к одному и тому же файлу или продолжение работы одного зависит от выполнения какого-то условия в другом? Для решения этих проблем в Win32 предусмотрен механизм синхронизации, который позволяет, что следует из его названия, синхронизировать работу потоков.

Обычно поток, работа которого зависит каким-то образом от другого потока, сообщает системе о том, какое событие он ожидает. После этого выполнение этого потока приостанавливается до наступления ожидаемого события. Обычно для синхронизации используются четыре типа объектов - семафоры, исключающие семафоры (объекты типа mutex), события и критические секции. Далее мы поговорим об этих объектах.

СЕМАФОРЫ

Семафор действует как обычный флажок, и используется для того, чтобы определить, свободен или нет в настоящее время требующийся потоку ресурс. Тем не менее, фактически семафор является не просто флагом. Для того чтобы создать семафор, приложение должно вызвать функцию CreateSemaphore(), описание которой находится в файле winbase.h и выглядит следующим образом:

```
WINBASEAPI HANDLE WINAPI CreateSemaphoreA(
        LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
        LONG lInitialCount, LONG lMaximumCount,
        LPCSTR lpName);
```

```
WINBASEAPI HANDLE WINAPI CreateSemaphoreW(
            LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
            LONG lInitialCount,
            LONG lMaximumCount,
            LPCWSTR lpName);
#ifdef UNICODE
#define CreateSemaphore  CreateSemaphoreW
#else
#define CreateSemaphore  CreateSemaphoreA
#endif // !UNICODE
```

Первый аргумент, что и следует из его типа, является указателем на структуру, содержащую атрибуты доступа к семафору. Он может также принимать значение NULL в том случае, если эти атрибуты не используются, как, например, в Windows'95.

Второй аргумент - начальное значение счетчика учета ресурсов. Этот аргумент определяет, сколько потоков может получить доступ к ресурсам в момент вызова функции. К примеру, компьютер имеет три порта, к которым обращается программа. В этом случае значение счетчика учета ресурсов может быть в пределах от 0 (нет свободных портов) до трех (все порты свободны). При обращении потока к ресурсу система проверяет, свободен ли ресурс, т. е. не установлено ли максимальное значение счетчика учета ресурсов (третий аргумент функции), после чего разрешает или запрещает доступ к ресурсу. Если для потока ресурсы недоступны, то он будет ждать освобождения ресурсов.

Последний, четвертый аргумент, - это указатель на строку, содержащую имя семафора.

При успешном завершении функция возвращает хэндл созданного семафора. Возвращение NULL сигнализирует о том, что произошла ошибка.

Если два процесса используют семафор с одним и тем же именем, то в этом случае используется один и тот же семафор. Использование этого семафора и является способом синхронизации потоков.

Перед завершением потока, использующего семафор, последний должен быть освобожден. Это делается с помощью функции ReleaseSemaphore(), описание которой, взятое из файла winbase.h, приведено ниже:

```
WINBASEAPI BOOL WINAPI ReleaseSemaphore(HANDLE hSemaphore,
                                        LONG lReleaseCount,
                                        LPLONG lpPreviousCount);
```

Первый аргумент - это хэндл семафора, полученный с помощью функции CreateSemaphore(). Второй аргумент определяет, какое значение

должно быть установлено в счетчике ресурсов семафора при его освобождении. В двойное слово, адрес которого определяется третьим аргументом, записывается предыдущее значение счетчика.

Таблица 55. Флаги доступа к семафору

Параметр	Описание
SEMAPHORE_ALL_ACCESS	Устанавливает все возможные флаги доступа к семафору
SEMAPHORE_MODIFY_STATE	Разрешается изменение счетчика ресурсов в функции ReleaseSemaphore()
SYNCHRONIZE	Разрешается использование в любой из ожидающих функций сигнала об изменении состояния семафора

Поток, которому заведомо известно, что семафор уже создан, может не создавать семафор, а открыть его с помощью функции OpenSemaphore(). Ниже приводено описание этой функции, взятое из файла winbase.h:

```
WINBASEAPI HANDLE WINAPI OpenSemaphoreA(DWORD dwDesiredAccess,
                                        BOOL bInheritHandle,
                                        LPCSTR lpName);
WINBASEAPI HANDLE WINAPI OpenSemaphoreW(DWORD dwDesiredAccess,
                                        BOOL bInheritHandle,
                                        LPCWSTR lpName);
#ifdef UNICODE
#define OpenSemaphore  OpenSemaphoreW
#else
#define OpenSemaphore  OpenSemaphoreA
#endif // !UNICODE
```

Первый аргумент определяет уровень доступа к семафору и может принимать значения, приведенные в табл. 55.

Второй аргумент определяет, наследуют ли этот семафор другие процессы, создаваемые функцией CreateProcess(). Значение TRUE говорит о том, что семафор является наследуемым.

Главным аргументом в этой функции является третий аргумент, определяющий имя открываемого семафора. Если функция выполняется успешно, то она возвращает хэндл открытого семафора.

Созданный или открытый семафор можно использовать с помощью функции WaitForSingleObject(), описание которой, приведенное ниже, можно найти в файле winbase.h:

WINBASEAPI DWORD WINAPI WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);

Первый аргумент функции очевиден - хэндл семафора. Второй аргумент определяет время ожидания наступления события в миллисекундах. Если это значение равно 0, то функция сразу же прекращает ожидание и возвращает управление. Если время ожидания определено как INFINITE, то ожидание наступления события не прекращается. Функция может вернуть значения, приведенные в табл. 56.

Алгоритм работы с семафорами выглядит следующим образом:

- поток создает или открывает семафор с помощью функций CreateSemaphore() или OpenSemaphore() соответственно;
- поток вызывает функцию WaitForSingleObject() (или WaitForMultipleObjects()) для того, чтобы определить, свободен ли требующийся потоку ресурс. В зависимости от результата, возвращаемого этой функцией, определяются дальнейшие действия;
- при завершении поток вызывает функцию ReleaseSemaphore(), освобождающую семафор.

СОБЫТИЯ

События являются самой примитивной разновидностью объектов синхронизации. Они используются для того, чтобы оповестить поток о том, что наступило ожидаемое событие. Эти объекты обычно используются для того, чтобы синхронизировать потоки, которые работают по принципу конвейера. К примеру, один поток опрашивает датчики и загружает считанные значения в буфер. Другой поток считывает эти данные из буфера и производит их обработку. Первый поток может сигнализировать второму о том, что событие - заполнение буфера - наступило. Второй поток может сигнализировать первому о том, что наступило другое событие - данные из буфера считаны, ожидается новая порция данных. Событие может иметь два состояния - занятое (nonsignaled) и свободное (signaled).

Таблица 56. Значения, возвращаемые функцией WaitForSingleObject()

Параметр	Значение	Описание
WAIT_OBJECT_0	0x00000000	Объект перешел в состояние свободного
WAIT_TIMEOUT	0x00000102	Объект за указанное время не перешел в состояние свободного
WAIT_ABANDONED	0x00000080	Объект mutex стал свободным из-за отказа от него
WAIT_FAILED	0xFFFFFFFF	Произошла ошибка

Для того чтобы использовать событие, его нужно создать. Делается это посредством функции CreateEvent(), описание которой, приведенное ниже, взято из файла winbase.h:

```
WINBASEAPI HANDLE WINAPI CreateEventA(
                    LPSECURITY_ATTRIBUTES lpEventAttributes,
                    BOOL bManualReset,
                    BOOL bInitialState,
                    LPCSTR lpName);
WINBASEAPI HANDLE WINAPI CreateEventW(
                    LPSECURITY_ATTRIBUTES lpEventAttributes,
                    BOOL bManualReset,
                    BOOL bInitialState,
                    LPCWSTR lpName);
#ifdef UNICODE
#define CreateEvent  CreateEventW
#else
#define CreateEvent  CreateEventA
#endif // !UNICODE
```

С первым аргументом этой функции - указателем на структуру, содержащую атрибуты доступа, мы уже знакомы.

Второй аргумент определяет тип создаваемого события. Если значение этого параметра равно TRUE, то создается объект, для сброса которого в свободное состояние необходимо использовать функцию ResetEvent(). При значении FALSE создается событие, автоматически сбрасывающееся в свободное состояние.

Третий аргумент определяет начальное состояние создаваемого события. Значение TRUE определяет, что создается событие в СВОБОДНОМ состоянии. Если поток намерен создать событие в занятом состоянии, то он должен установить этот параметр в FALSE.

И наконец, последний, четвертый аргумент, определяет имя создаваемого объекта - события.

При успешном выполнении функция возвращает хэндл созданного объекта-события. Если при выполнении функции встретилась ошибка, то возвращается значение NULL.

Для того чтобы сигнализировать о наступлении события, в потоке должна присутствовать функция SetEvent(), переводящая событие в свободное состояние, описанная в winbase.h следующим образом:

```
WINBASEAPI BOOL WINAPI SetEvent(HANDLE hEvent);
```

Единственный аргумент этой функции очевиден - хэндл события, созданного посредством CreateEvent().

Каким образом прикладная программа может узнать о наступлении события? Да с помощью уже знакомой нам функции WaitForSingleObject().

Для того чтобы сбросить событие в занятое состояние, необходимо вызвать функцию ResetEvent(), описанную в том же winbase.h следующим образом:

```
WINBASEAPI BOOL WINAPI ResetEvent(HANDLE hEvent);
```

В качестве аргумента функции передается хэндл события.

Достаточно часто встречаются случаи, когда после установки события с помощью SetEvent() тут же следует вызов ResetEvent(). Для этих целей предусмотрена функция PulseEvent(), описанная так:

```
WINBASEAPI BOOL WINAPI PulseEvent(HANDLE hEvent);
```

Аргументом является хэндл события. После выполнения функции объект-событие остается в занятом состоянии.

Алгоритм использования объекта-события полностью аналогичен алгоритму использования семафора.

Надеюсь, что после того, что сейчас узнал читатель, разобрать критические секции и объекты типа mutex ему не составит труда.

ДИНАМИЧЕСКИ ПОДКЛЮЧАЕМЫЕ БИБЛИОТЕКИ

Думаю, что, посмотрев на размер исполняемого файла, полученного после компиляции нашей первой программы «Helloworld», многие были поражены. Как! Столько всего умеет делать программа при таком малом размере! Даже на ассемблере невозможно написать библиотеку такого размера и обладающую такими возможностями. Как же это сделано? Ответ прост - большая часть кода, обеспечивающего возможности программы, находится вне самое программы, в библиотеках. Это естественно и понятно. Но с исполняемым файлом эти библиотеки соединяются не на стадии линкования, как обычные библиотеки, а НА СТАДИИ ВЫПОЛНЕНИц! Это одно из принципиальных положений, отличающих все версии Windows от ее главного в прошлом конкурента MS DOS.

Библиотеки динамической компоновки представляют собою одного из тех китов, на которых базировались Windows всех версий, в том числе и Windows'95. Все функции API, с которыми мы работаем, находятся в библиотеках динамической компоновки - DLL (dynamic link libraries). Основу Windows составляют три библиотеки: kernel32dll, user32.dll и gdi32.dll. Первая отвечает за управление памятью, процессами и потоками, вторая - за систему окон с подсистемой сообщений, третья - за графику и вывод текста (само название - GDI - является аббревиатурой выражения Graphical User Interface - графический интерфейс пользователя).

Это, так сказать, самое Windows. Многочисленные DLL, которые можно найти в директории Windows, являются ее расширениями. Но, естественно, пользователи об этом и не догадываются.

Как правило, написать DLL значительно легче, чем разработать программу, ибо DLL - это всего-навсего набор автономных функций, предназначенных для вызова из других приложений или DLL. Естественно, что при этом условии в DLL совершенно не нужен цикл обработки сообщений или функция создания окна, что, как правило, присутствует в вызывающей программе. DLL компилируются и линкуются точно так же, как и обычные программы, единственное, необходимо указать линкеру, что необходимо создать DLL, а не программу. В этом случае линкование проходит несколько другим способом, в результате чего в конечный файл записывается другая информация, и загрузчик Windows с легкостью отличает DLL от обычной программы.

СПОСОБЫ ПРИСОЕДИНЕНИЯ DLL К ПРОГРАММЕ

Для того чтобы программа смогла выполнить код, находящийся в библиотеке, необходимо этот код сначала загрузить в память, выделенную вызывающему процессу, после чего оповестить вызывающий процесс о том, где находится загруженный код.

Решить эту задачу можно двумя способами. Первый способ - это неявное линкование с DLL. Второй способ - явная загрузка DLL.

НЕЯВНОЕ ЛИНКОВАНИЕ С DLL

При неявном линковании программа не знает о том, какую библиотеку ей необходимо присоединить. Для того чтобы неявно прилинковать библиотеку, необходимо на этапе подготовки проекта произвести некоторые действия: создать файл с расширением .lib, содержащий ссылку на DLL и перечень находящихся в ней функций. Делается это с помощью утилиты implib. Вызывается она следующим образом:

```
implib FileName1.lib FileName2.dll
```

где FileName1.lib - это имя создаваемого файла; а FileName2.dll - DLL. Полученный lib-файл можно прилинковать к вашей программе точно так же, как и любую другую библиотеку.

Для того чтобы проиллюстрировать процесс вызова DLL при неявном линковании, я напишу библиотеку, в которой будет находиться всего одна функция. При обращении к этой функции будет выдаваться окно с сообщением (MessageBox) «Сейчас мы в DLL!». Наверное, трудно при-

думать что-то более простое, а? Приложение, которое будет обращаться к этой библиотеке, всего-навсего будет вызывать эту функцию, не создавая никаких окон, циклов обработки сообщений и т. д. Листинги программы и библиотеки (я назвал ее dll.dll) приведены ниже:

```
#ifdef _MYDLL_
  #define MYAPI __declspec(dllexport)
 #else
  #define MYAPI __declspec(dllimport)
#endif

MYAPI void CALLBACK MyMessage();
```

Листинг № 7. Файл заголовков библиотеки dll.h:

```
#include <windows.h>

#define _MYDLL_

MYAPI void CALLBACK MyMessage()
 {
  MessageBox(NULL, "Now we are in DLL!", "Hurray!" , MB_OK);
 }
```

Листинг № 8. Основной файл библиотеки dll.cpp:

```
LIBRARY      MyMessage
DESCRIPTION 'Program'
EXETYPE      WINDOWS
CODE          PRELOAD MOVEABLE DISCARDABLE
DATA          PRELOAD MOVEABLE SINGLE
```

Листинг № 9. Файл определения модуля dll.def:

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

int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,
                   LPSTR lpszCmdLine, int nCmdShow)
 {
  MyMessage();
  return 1;
 }
```

Листинг № 10. Основной файл программы, осуществляющей вызов библиотечной функции, app.cpp посредством неявной компоновки:

```
NAME        MyApp
DESCRIPTION 'Program'
EXETYPE     WINDOWS
CODE        PRELOAD MOVEABLE DISCARDABLE
DATA        PRELOAD MOVEABLE MULTIPLE
```

Листинг № 11. Файл определения модуля app.def.

Для того чтобы эта связка программа-DLL заработала, проделайте следующие действия:

создайте DLL;

с помощью утилиты IMPLIB создайте lib-файл;

при создании exe-файла прилинкуйте файл, полученный с помощью implib, к вашей программе;

запустите exe-файл.

Надеюсь, что после всех этих действий вы увидите на экране сообщение о том, что произошло обращение к DLL.

В данном случае при загрузке выполняемого файла система просматривает его для того, чтобы определить, какие DLL используются при его работе, после чего пытается загрузить требующиеся DLL. Поиск DLL осуществляется в следующих каталогах:

каталоге, содержащем exe-файл;

текущем каталоге процесса;

системном каталоге Windows;

каталоге Windows;

каталогах, указанных в PATH.

Попробуйте изменить имя DLL-файла. Вы заметите, что если файл DLL не найден, то система выдает сообщение об этом и немедленно завершает процесс.

У неявной компоновки есть свои преимущества и недостатки. По моему мнению, к преимуществам нужно отнести следующее:

программа может ничего не знать о том, что она использует DLL;

проверка доступности DLL производится еще до загрузки программы, т. е. в случае отсутствия DLL программа просто не запустится.

Недостатком такого способа можно считать то, что DLL загружается в память до программы и выгружается после окончания программы, т. е. программист не может управлять загрузкой и выгрузкой библиотек. Если программа использует несколько библиотек, то придется все библиотеки хранить в памяти от запуска до завершения программы. Наверное, иногда

неплохо было бы обратиться и к другому способу подключения библиотек к программе, который называется

ЯВНАЯ ЗАГРУЗКА DLL

В этом случае все манипуляции с DLL производит вызывающая программа. Для того чтобы библиотека загрузилась в память, должна быть вызвана одна из функций - LoadLibrary() или LoadLibraryEx().

В winbase.h эти функции описаны следующим образом:

```
WINBASEAPI HMODULE WINAPI LoadLibraryA(LPCSTR lpLibFileName);
WINBASEAPI HMODULE WINAPI LoadLibraryW(LPCWSTR lpLibFileName);
#ifdef UNICODE
#define LoadLibrary  LoadLibraryW
#else
#define LoadLibrary  LoadLibraryA
#endif // !UNICODE

WINBASEAPI HMODULE WINAPI LoadLibraryExA(LPCSTR lpLibFileName,
                                         HANDLE hFile,
                                         DWORD dwFlags);
WINBASEAPI HMODULE WINAPI LoadLibraryExW(LPCWSTR lpLibFileName,
                                         HANDLE hFile,
                                         DWORD dwFlags);
#ifdef UNICODE
#define LoadLibraryEx  LoadLibraryExW
#else
#define LoadLibraryEx  LoadLibraryExA
#endif // !UNICODE
```

Аргументом первой функции является имя загружаемой DLL. Другая же при вызове должна получить три аргумента. Первый - то же имя загружаемой DLL. Второй аргумент зарезервирован и должен быть равен NULL. Третий аргумент должен представлять собой либо нуль, либо комбинацию из трех флагов: DONT_RESOLVE_DLL_REFERENCES, LOAD_LIBRARY_AS_DATAFILE, LOAD_WITH_ALTERED_SEARCH_PATH.

DONT_RESOLVE_DLL_REFERENCES

Чуть позже мы рассмотрим функцию, которая производит инициализацию и деинициализацию DLL автоматически при загрузке. Данный флаг заставляет систему не вызывать функцию инициализации. Кроме этого, при загрузке библиотеки система проверяет, не используются ли данной DLL функции из других DLL. Если используются, то загружаются и они. Если данный флаг установлен, то дополнительные библиотеки не загружаются.

LOAD_LIBRARY_AS_DATAFILE

Этот флаг может использоваться в нескольких случаях. Во-первых, можно загружать библиотеку, не содержащую никакого кода и содержащую только ресурсы. Полученное значение HINSTANCE можно использовать при вызове функций, использующих ресурсы. Во-вторых, если мы загружаем exe-файл обычным способом, то это приводит к запуску нового процесса. А как быть в том случае, если мы хотим получить доступ к ресурсам exe-файла, не запуская его? При загрузке exe-файла с помощью функции LoadLibraryEx() с установленным флагом LOAD_LIBRARY_AS_DATAFILE, возможно получить доступ к ресурсам exe-файла.

LOAD_WITH_ALTERED_SEARCH_PATH

Ранее мы рассмотрели, какие каталоги и в какой последовательности просматриваются системой при загрузке DLL. Если установлен флаг LOAD_WITH_ALTERED_SEARCH_PATH, то просмотр каталогов начинается с каталога, указанного в первом аргументе функции LoadLibraryEx(). Далее просмотр продолжается в обычном порядке.

После загрузки DLL программа не может вызывать требующиеся ей функции. Для того чтобы вызвать какую-либо функцию, ей необходимо сначала определить адрес этой функции с помощью GetProcAddress(), а затем вызывать функцию через полученный адрес. После того, как надобность в присутствии DLL в памяти отпала, программа должна выгрузить ее с помощью функций FreeLibrary() или FreeLibraryAndExitThread(). Но сейчас разговор не об этом. Давайте попробуем рассмотреть предыдущий пример, измененный таким образом, чтобы DLL загружалась явно.

Само собой разумеется, что все, что касается DLL, никаким изменениям не подвергалось. Иначе какой смысл писать DLL, если в зависимости от потребностей программиста ее нужно было бы каждый раз переписывать? Изменился только основной файл программы, которая вызывает функцию из DLL. Итак...

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

int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,
                   LPSTR lpszCmdLine, int nCmdShow)
 {
  HINSTANCE hDll;
  FARPROC MyProcAddr;

  if( (hDll = LoadLibrary("dll.dll")) != NULL)
```

```
    MyProcAddr = GetProcAddress(hDll, "MyMessage");
  else
    {
     MessageBox(NULL, "Sorry, cannot find requested DLL", "Sorry", MB_OK);
     return 0;
    }
 (MyProcAddr)();
 FreeLibrary(hDll);
 return 1;
}
```

Листинг № 12. Основной файл программы, осуществляющей вызов библиотечной функции, app.cpp посредством явной загрузки DLL.

С точки зрения пользователя заметить какие-либо отличия в работе программ, использующих для вызова функций из DLL неявную компоновку и явную загрузку, невозможно. С точки зрения программиста два отличия прямо-таки бросаются в глаза. Первое - если при неявной компоновке в случае отсутствия DLL программа просто не запускается, то в случае явной загрузки, возможно перехватить такую ситуацию и предпринять какие-либо действия. И второе - у программиста прибавилось головной боли. Вместо обычного обращения к функции он должен вызвать еще три вспомогательные функции, да и требующаяся функция из DLL вызывается не напрямую, а косвенно, посредством использования ее адреса. Еще раз повторю - программист должен сам решить, стоит ли овчинка выделки.

ВЫВЕРНЕМ ПРОГРАММЫ НАИЗНАНКУ

Давайте попробуем разобраться в том, что все это означает и к чему может привести.

Начнем с файла, который используется как библиотекой, так и приложением - файла заголовков. Чтобы exe-файл мог вызывать функции из DLL, то, с одной стороны, библиотека должна объявить их как доступные, или, как говорят, экспортируемые. С другой стороны, сам exe-файл должен определить эти функции как находящиеся в DLL, т. е. как импортируемые. Если мы объявим эти функции по-разному в заголовочном файле и непосредственно в тексте, мы не оберемся ошибок при компиляции. Следовательно, выход один - условная компиляция. Если мы посмотрим на распечатку заголовочного файла dll.h, то увидим, что я определяю макро MYAPI, которое принимает одно из двух значений (__declspec(dllexport)) или (__declspec(dllimport)) в зависимости от факта определения другого макро, _MYDLL_ . Теперь понятно, что и в заголовочном, и в исходных файлах можно описать функцию, находящуюся в

DLL, как MYAPI, но при этом в исходном файле библиотеки мы должны определить макро _MYDLL_ , а в исходном файле приложения ни в коем случае это макро не определять. Что и сделано. Проверьте - работает! Посмотрите в заголовочные файлы Win32. По-моему, в них используется эта техника.

Описание функций как экспортируемых требуется линкеру для того, чтобы правильно построить таблицу экспортируемых функций в dll-файле. Каждый элемент в этой таблице содержит имя экспортируемой функции, а также ее адрес. Немаловажно, что список функций сортируется по алфавиту. Таким образом, если функция должна работать быстро, то какое-то преимущество можно получить в том случае, если имя функции будет начинаться с первых букв алфавита. Но это преимущество сработает только в момент выполнения GetProcAddress(), а не (увы!) при обращении к функции. Для того чтобы разобраться во внутренностях DLL, воспользуемся утилитой TDUMP, поставляемой с Borland C++ 5.0 (аналогичные утилиты есть и в других системах программирования, например, в Microsoft Visual C++ подобная утилита называется DUMPBIN). Часть распечатки таблицы экспорта для kernel32.dll приведена ниже:

```
Turbo Dump  Version 4.2.15.2 Copyright (c) 1988, 1996 Borland International
            Display of File KERNEL32.DLL

Exports from KERNEL32.dll
  680 exported name(s), 780 export addresse(s).  Ordinal base is 1.
    Ordinal RVA      Name


    ------- -------- ----
    0049    0002d900 AddAtomA
    0101    00034c99 AddAtomW
    0102    0002f44b AllocConsole
    0103    00021b22 AllocLSCallback
    0104    00021b55 AllocSLCallback
    0105    0002e75b AreFileApisANSI
    0106    00034d20 BackupRead
    .............
    .............
    .............
    0774    000071da lstrcpyn
    0775    000071da lstrcpynA
    0776    00034ccf lstrcpynW
    0777    00007251 lstrlen
    0778    00007251 lstrlenA
    0779    0002b83c lstrlenW
```

Как можно убедиться, все функции рассортированы в алфавитном порядке? В первой колонке указаны ordinals - порядковые номера функций. В более ранних версиях Windows функции могли экспортироваться не только по именам, но и по порядковым номерам. Сейчас Microsoft рекомендует пользоваться только именами функций (именно поэтому я не описываю способ экспорта и импорта функций посредством указания порядковых номеров). Для справки - средняя колонка содержит Real Virtual Addresses - адреса функций внутри DLL.

Аналогично решается и вопрос при импорте функций. Функция объявляется как импортируемая, после чего линкер строит специальный код и таблицу импортируемых функций.

Кстати, помимо TDUMP можно воспользоваться утилитой IMPDEF, которая выдает список присутствующих в DLL функций. Что же касается моего личного мнения, то я рекомендую читателю изучить форматы файлов Windows и написать самостоятельно несколько утилит, которые будут использоваться для «выворачивания программ наизнанку» и показывать все те данные и в таком виде, который удобен программисту.

ИНИЦИАЛИЗАЦИЯ И ДЕИНИЦИАЛИЗАЦИЯ DLL

Мы разобрались, каким образом можно написать DLL и вызвать функцию из DLL. Но, как правило, во всех нормальных (не демонстрационных) примерах существуют блоки, отвечающие за инициализацию и деинициализацию программы. Возникает вопрос: как это сделать в DLL? Есть ли такая возможность? Да, есть! В каждой библиотеке может быть функция, которая вызывается строго в определенных обстоятельствах и обычно используется библиотекой для инициализации и деинициализации. В нашей микро-DLL эта функция не использовалась, однако, я счел бы тему рассмотренной не полностью, если бы обошел этот вопрос стороной.

В Borland C++ v. 5.0 эта функция по умолчанию называется DllEntryPoint() и в некотором смысле является аналогом связки LibMain() + WEP() в Windows 3.x. Вызывается эта функция всего в четырех случаях и имеет следующий вид:

```
BOOL WINAPI DllEntryPoint(HINSTANCE hinstDll, DWORD
fdwReason, LPVOID lpvReserved)
  {
   switch(fdwReason)
    {
     case DLL_PROCESS_ATTACH:
```

```
    /* Операторы */
  case DLL_THREAD_ATTACH:
    /* Операторы */
  case DLL_THREAD_DETACH:
    /* Операторы */
  case DLL_PROCESS_DETACH:
    /* Операторы */
  }
 return(TRUE);
}
```

Лично мне именно такой способ инициализации и деинициализации библиотек динамической компоновки очень импонирует. Дело в том, что он использует оператор switch - case, который применяется при обработке сообщений. Именно из-за этого вся конструкция зрительно воспринимается так же, как и оконная процедура. Мне кажется, что с такой функцией намного приятнее иметь дело, чем со связкой LibMain() - WEP в Windows 3.x.

Первый аргумент этой функции - это хэндл библиотеки, присваиваемый системой.

Второй аргумент указывает причину вызова этой библиотеки системой.

Третий аргумент, как понятно из его названия, пока зарезервирован и обычно должен быть равным NULL.

Теперь нам необходимо до конца разобраться с причинами вызова библиотеки.

DLL_PROCESS_ATTACH

Система вызывает функцию инициализации с этим значением параметра fdwReason единственный раз при загрузке библиотеки. Другими словами, если один из потоков процесса, вызвавшего DLL, пытается вновь загрузить ее с помощью LoadLibrary(), то обращение к DLL с параметром DLL_PROCESS_ATTACH не произойдет. Система увеличит счетчик пользователей этой DLL.

Значение, возвращаемое функцией инициализации, после обработки DLL_PROCESS_ATTACH, уведомляет пользователя, была ли инициализация успешной. В случае неуспешной инициализации функция должна возвратить FALSE, при успехе - TRUE. Это значение используется как при неявной, так и при явной загрузке DLL в память.

DLL_PROCESS_DETACH

Вызов функции инициализации для обработки DLL_PROCESS_DETACH означает, что библиотека из памяти выгружается и должна произвести действия по деинициализации самое себя. Помимо того, что необходимо освободить память и другие ресурсы, хорошим тоном считается оставить систему точно в том же состоянии, в каком ее приняла библиотека (если, конечно, изменение параметров системы не является задачей одной из функций DLL). При выгрузке библиотеки есть одна тонкость, связанная с причиной завершениия процесса, обратившегося к ней. Если DLL выгружается в связи с вызовом функции ExitProcess() (или FreeLibrary(), хотя это и не связано с процессом), вызов функции инициализации проходит нормально. Но если процесс завершается благодаря функции TerminateProcess(), функция инициализации НЕ ВЫЗЫВАЕТСЯ! Таким образом, попутно можно сделать еще один вывод - функцией TerminateProcess() можно и нужно пользоваться только в самых крайних случаях!

Ниже приведен чуть измененный листинг библиотеки dll.c; попробуйте загрузить ее и понаблюдать за тем, как она работает:

```
#include <windows.h>

#define _MYDLL_

#include "dll.h"

BOOL WINAPI DllEntryPoint(HINSTANCE hinstDll, DWORD fdwReason,
                          LPVOID lpvReserved)
 {
  switch(fdwReason)
   {
    case DLL_PROCESS_ATTACH:
     MessageBox(NULL, "We are in DLL_PROCESS_ATTACH!", "Hurray!",
                MB_OK);
     break;
    case DLL_THREAD_ATTACH:
     MessageBox(NULL, "We are in DLL_THREAD_ATTACH!", "Hurray!",
                MB_OK);
     break;
    case DLL_THREAD_DETACH:
     MessageBox(NULL, "We are in DLL_THREAD_DETACH!", "Hurray!",
                MB_OK);
     break;
    case DLL_PROCESS_DETACH:
     MessageBox(NULL, "We are in DLL_PROCESS_DETACH!", "Hurray!",
                MB_OK);
```

```
        break;
      }
    return(TRUE);
  }

MYAPI void CALLBACK MyMessage()
  {
   MessageBox(GetDesktopWindow(),"DLL is called!", "Hurray!" , MB_OK);
  }
```

Листинг № 13. Библиотека dll.c, включающая код функции инициализации.

Очередной раз - ура! Мы научились писать библиотеки динамической компоновки, добавлять в них функции инициализации и деинициализации, подключать DLL к нашим программам, используя как неявную, так и явную загрузку.

КОНСОЛИ

«Неужели для того, чтобы написать простейшую программу, которая выводит на экран несколько строк, мне необходимо городить этот огород с WinMain() и функцией окна? Неужели в каждой моей программе, предназначенной для вывода текста на экран, должно присутствовать «стандартное заклинание»? Или же я вынужден вечно быть ограниченным рамками DOS?» Я предвижу такие вопросы со стороны тех, которым при разработке их программ не только не нужен, но и мешает графический интерфейс.

Что ж, вопросы вполне понятны и закономерны. Наверное, именно эта закономерность и обусловила появление в Win32 новых функций, обеспечивающих эмуляцию текстового терминала. Эти функции называются функциями консоли.

ЧТО ТАКОЕ КОНСОЛЬ

Консоль - это интерфейс, обеспечивающий поддержку программ, работающих в текстовом режиме, т. е. программ, написанных в стиле MS DOS. Консоль состоит из буфера ввода и одного или нескольких экранных буферов. Буфер ввода включает в себя очередь, каждая запись в которой содержит информацию о вводных событиях. Под вводными событиями в данном случае подразумеваются нажатия и отжатия клавиш на клавиатуре, на мыши, движения мыши, а также действия пользователя, производимые с окном. Экранный буфер - это двумерный массив, кото-

рый содержит коды символов и цвета символов текстового экрана (аналог видеобуфера в текстовом режиме при работе в MS DOS).

Каждая программа, работающая в текстовом режиме, взаимодействует с Windows через консоль. Если одна программа запускается из конçоли, принадлежащей другой программе (скажем, aidstest запускается из консоли Norton Commander'a), то запускаемая программа работает в той же консоли. Если же программа запускается самостоятельно из Windows, то ей выделяется собственная консоль (фактически ей выделяется целая виртуальная машина). Другими словами, каждая программа может получить для себя эмулятор DOS-машины и считать, что весь компьютер принадлежит только ей.

Даже из этого краткого описания видно, что консоли могут оказать программисту довольно существенную помощь, состоящую, во-первых, в том, что обработка действий пользователя с мышью и клавиатурой производится средствами Windows, а, во-вторых, разрешают доступ к некоторым функциям API. В-третьих, каждая программа может работать в своей сессии. В-четвертых, программе доступны стандартные потоки ввода-вывода DOS. Наверное, даже этого краткого перечисления достаточно для того, чтобы убедить читателя в том, что разработку программ, не имеющих графического интерфейса, имеет смысл производить с учетом новых возможностей, предоставляемых Win32.

Мы долго говорили о том, что с консолями работают программы, написанные в стиле MS DOS. Но в языке С точкой входа для DOS'овских программ является функция main(), а не WinMain(), следовательно, и консольные программы должны точкой входа тоже иметь функцию main(), а не WinMain(). Таким образом, основными отличиями консольных программ от обычных программ для Windows являются:

- отсутствие графического интерфейса;
- использование в качестве точки входа функции main(), а не WinMain().

ТЕХНИКА РАЗРАБОТКИ КОНСОЛЬНОЙ ПРОГРАММЫ

СОЗДАНИЕ КОНСОЛИ

Как уже было сказано, консольная программа должна иметь точкой входа не WinMain(), а main(). При запуске программы она должна запросить для себя консоль, используя для этого функцию AllocConsole(). Ее прототип находится в файле wincon.h, к которому мы будем обращаться в этом разделе:

```
WINBASEAPI BOOL WINAPI AllocConsole(VOID);
```

Эта функция, возвращающая TRUE при успешном завершении, предоставляет вызвавшей ее программе консоль. Внешне консоль выглядит так же, как и обычное окно. У него есть заголовок, системное меню, кнопки максимизации и минимизации. Если программе необходима собственная консоль, а она работает в унаследованной, то программа может перед вызовом AllocConsole() произвести освобождение консоли, вызвав для этого функцию FreeConsole(), описание которой практически не отличается от описания предыдущей функции:

```
WINBASEAPI BOOL WINAPI FreeConsole( VOID );
```

Если программа запускается в самостоятельной консоли, то вызов FreeConsole() не повлечет за собой никаких неприятных последствий.

При завершении программы самостоятельная консоль автоматически уничтожается, унаследованная же продолжает свое существование до момента завершения породившей ее программы.

ПРИСВОЕНИЕ КОНСОЛИ ИМЕНИ

Очередным шагом после создания консоли будет присвоение консоли имени. Это имя будет отображено в заголовке консоли. Делается это с помощью функции SetConsoleTitle(). Из wincon.h извлекаем прототип этой функции:

```
WINBASEAPI BOOL WINAPI SetConsoleTitleA(LPCSTR lpConsoleTitle );
WINBASEAPI BOOL WINAPI SetConsoleTitleW(LPCWSTR lpConsoleTitle);
#ifdef UNICODE
#define SetConsoleTitle  SetConsoleTitleW
#else
#define SetConsoleTitle  SetConsoleTitleA
#endif // !UNICODE
```

Аргумент этой функции - указатель на строку символов, содержащую текст, который будет отображен в заголовке окна консоли.

ВВОД И ВЫВОД В КОНСОЛИ

Основные функции вывода в окно консоли

Когда я изучал этот вопрос, функции ввода и вывода в консоли напомнили мне вызов прерываний DOS (наверное, так и должно быть, ведь консоль эмулирует DOS-машину). Поэтому знакомые с прерываниями DOS программисты увидят в функциях ввода - вывода много «знакомого».

В DOS для операций ввода - вывода считалось, что стандартные потоки, такие, как поток ввода, поток вывода и поток ошибок, имеют стандартные, заранее определенные хэндлы. При работе в режиме консоли стандартные потоки предопределенных хэндлов не имеют, поэтому эти хэндлы необходимо получить, обратившись к функции GetStdHandle(). По ее описанию -

```
WINBASEAPI HANDLE WINAPI GetStdHandle(DWORD nStdHandle);
```

извлеченному в данном случае из файла winbase.h, мы видим, что для получения хэндла стандартного потока в консольной сессии мы в качестве аргумента функции должны указать номер того потока, хэндл которого нам нужен. Запоминать номера потоков не нужно, они определены в том же файле winbase.h как STD_INPUT_HANDLE, STD_OUTPUT_HANDLE и STD_ERROR_HANDLE. При успешном завершении функция возвращает хэндл требующегося потока, в противном случае возвращаемое значение равно INVALID_HANDLE_VALUE.

Определив хэндл стандартного потока, можно попытаться вывести текст в окно консоли с помощью функции WriteConsole():

```
WINBASEAPI BOOL WINAPI WriteConsoleA(HANDLE hConsoleOutput,
                                     CONST VOID *lpBuffer,
                                     DWORD nNumberOfCharsToWrite,
                                     LPDWORD lpNumberOfCharsWritten,
                                     LPVOID lpReserved);
WINBASEAPI BOOL WINAPI WriteConsoleW(HANDLE hConsoleOutput,
                                     CONST VOID *lpBuffer,
                                     DWORD nNumberOfCharsToWrite,
                                     LPDWORD lpNumberOfCharsWritten,
                                     LPVOID lpReserved);
#ifdef UNICODE
#define WriteConsole  WriteConsoleW
#else
#define WriteConsole  WriteConsoleA
#endif // !UNICODE
```

Аргументами этой функции являются:

- хэндл стандартного потока вывода;
- указатель на выводимую строку;
- длина выводимой строки в символах;
- указатель на двойное слово, в которое записывается действительное число выведенных символов;

• указатель на двойное слово, зарезервированное для использования в дальнейшем, который должен быть равным NULL.

Строка выводится, начиная с текущей позиции курсора, при этом используются текущие цветовые атрибуты текста и фона. Курсор устанавливается в позицию, следующую за последним символом строки.

В случае успешного завершения функция возвращает TRUE.

Для установки позиции курсора в консоли необходимо вызвать функцию SetConsoleCursorPosition(), прототип которой можно найти в wincon.h:

```
WINBASEAPI BOOL WINAPI SetConsoleCursorPosition(
                        HANDLE hConsoleOutput,
                        COORD dwCursorPosition);
```

hConsoleOutput - это хэндл стандартного вывода консоли, а структура типа COORD, содержащая координаты новой позиции курсора, определяется в wincon.h следующим образом:

```
typedef struct _COORD {
  SHORT X;
  SHORT Y;
} COORD, *PCOORD;
```

X и Y - координаты новой позиции курсора.

Если функция завершена успешно, она возвращает ненулевое значение.

Последнее, что нам осталось сделать для того, чтобы мы могли полностью управлять выводом, - это научиться устанавливать цветовые атрибуты выводимого текста. Учиться недолго - это делается с помощью функции SetConsoleTextAttribute(). Извлечем из wincon.h ее прототип:

```
WINBASEAPI BOOL WINAPI SetConsoleTextAttribute(HANDLE hConsoleOutput,
                                               WORD wAttributes);
```

hConsoleOutput - хэндл стандартного потока вывода консоли, а wAttribues определяет цвета тона и фона текста. wAttributes должен быть комбинацией нескольких флагов. Перечень флагов приведен в табл. 57.

Таблица 57. Атрибуты цветов фона и тона окна консоли

Флаг	Значение	Эффект
FOREGROUND_BLUE	0x0001	Тон содержит синюю составляющую
FOREGROUND_GREEN	0x0002	Тон содержит зеленую составляющую
FOREGROUND_RED	0x0004	Тон содержит красную составляющую
FOREGROUND_INTENSITY	0x0008	Тон имеет повышенную интенсивность
BACKGROUND_BLUE	0x0010	Фон имеет синюю составляющую
BACKGROUND_GREEN	0x0020	Фон имеет зеленую составляющую
BACKGROUND_RED	0x0040	Фон имеет красную составляющую
BACKGROUND_INTENSITY	0x0080	Фон имеет повышенную интенсивность или текст мигает (только в полноэкранном режиме)

В разделе, посвященном графике, упоминалось о том, что каждый пиксель на экране состоит из трех микроточек, при этом интенсивность свечения каждой точки может изменяться от нуля до 255. В текстовом режиме все проще. В обычных условиях (подчеркиваю - обычных условиях!) тон символа (не пикселя - символа!) тоже определяется как состоящий из трех компонентов, однако их интенсивности могут быть 0 и 127 (флаг интенсивности не установлен), 0 и 255 (флаг интенсивности установлен). Таким образом, всего возможно 16 цветов тона символов.

Точно так же обстоит дело и с фоном символов. Однако в зависимости от некоторых условий (обсуждение возможностей и регистров видеоадаптера явно выходит за рамки этой книги), установленный флаг интенсивности тона обеспечивает либо повышенную интенсивность фона, либо мигание символа. Но все это касается только программ, работающих в полноэкранном режиме (не путать максимизированную консоль с работой в полноэкранном режиме!). Рекомендую читателю проверить эту установку на своем компьютере. Программы в консольном режиме, к сожалению, не могут использовать мигание фона, поэтому те символы, для которых в полноэкранном режиме установлен флаг мигания, в консольном режиме не мигают, а отображаются с повышенной интенсивностью цвета фона символов.

Мы закончили краткое обсуждение функций, обеспечивающих вывод на консоль. Давайте теперь рассмотрим функцию, обеспечивающую

Ввод из окна консоли

Для ввода из окна консоли текста с клавиатуры используется функция ReadConsole(), описанная следующим образом:

```
WINBASEAPI BOOL WINAPI ReadConsoleA(
```

```
                                HANDLE hConsoleInput,
                                LPVOID lpBuffer,
                                DWORD nNumberOfCharsToRead,
                                LPDWORD lpNumberOfCharsRead,
                                LPVOID lpReserved);
WINBASEAPI BOOL WINAPI ReadConsoleW(
                                HANDLE hConsoleInput,
                                LPVOID lpBuffer,
                                DWORD nNumberOfCharsToRead,
                                LPDWORD lpNumberOfCharsRead,
                                LPVOID lpReserved);
#ifdef UNICODE
#define ReadConsole  ReadConsoleW
#else
#define ReadConsole  ReadConsoleA
#endif // !UNICODE
```

Аргументами этого файла являются:

- hConsoleInput - хэндл потока ввода;
- lpBuffer - указатель на символьный массив, в который будет записана строка символов;
- nNumberOfCharsToRead - максимальное число вводимых символов;
- lpNumberOfCharsRead - число фактически считанных символов;
- lpReserved - зарезервировано для дальнейшего использования и должно быть равно NULL.

На этом мы завершаем рассмотрение основных функций, обеспечивающих ввод-вывод в окно консоли. После демонстрационной программы рассмотрим еще несколько функций, обеспечивающих работу с клавиатурой и мышью.

Демонстрационная программа

Перед тем, как привести текст демонстрационной программы, мне бы хотелось обратить внимание читателя на одну тонкость. Программы, которые мы рассматривали до сих пор, являлись оконными программами. Я не давал никаких пояснений по поводу их компиляции. Но та программа, которую мы будем разбирать сейчас, не имеет собственного графического интерфейса, т. е. оконной программой не является, а следовательно, и компилировать ее нужно несколько иным способом. Предлагаю читателю обратиться к руководству по той системе, с которой он работает, и выяснить, каким образом можно скомпилировать консольную программу. Если читатель паче чаяния работает с Borland C++ 5.0 в IDE, то ему при создании нового проекта в TargetExpert необходимо изменить TargetMode с GUI на Console.

Сделано? Тогда листинг демонстрационной программы перед вами:

```
#include <windows.h>
#include <stdio.h>
main()
 {
  HANDLE hStdInputHandle, hStdOutputHandle;
  COORD Coord;
  char cMyString[255] = "This is our first consolt program! It's working !";
  DWORD dwResult;

  FreeConsole();
  AllocConsole();
  SetConsoleTitle("Console Demonstration program");
  hStdInputHandle = GetStdHandle(STD_INPUT_HANDLE);
  hStdOutputHandle = GetStdHandle(STD_OUTPUT_HANDLE);
  Coord.X = (80 - strlen(cMyString)) / 2;
  Coord.Y = 12;
  SetConsoleCursorPosition(hStdOutputHandle, Coord);
  SetConsoleTextAttribute(hStdOutputHandle, FOREGROUND_RED |
                                            BACKGROUND_RED |
                                            BACKGROUND_BLUE |
                                            BACKGROUND_GREEN |
                                            FOREGROUND_INTENSITY |
                                            BACKGROUND_INTENSITY);
  WriteConsole( hStdOutputHandle, cMyString, strlen(cMyString), &dwResult, NULL);
  SetConsoleTextAttribute(hStdOutputHandle, 0);
  getchar();
  SetConsoleCursorPosition(hStdOutputHandle, Coord);
  WriteConsole(hStdOutputHandle, cMyString, strlen(cMyString),
               &dwResult, NULL);
  Coord.X = 0;
  Coord.Y = 12;
  SetConsoleCursorPosition(hStdOutputHandle, Coord);
  SetConsoleTextAttribute(hStdOutputHandle, FOREGROUND_RED |
                                            FOREGROUND_BLUE |
                                            FOREGROUND_GREEN |
                                            FOREGROUND_INTENSITY );
  WriteConsole(hStdOutputHandle, "Type some letters and press Enter, please: ",
                strlen(cMyString), &dwResult, NULL);
  SetConsoleTextAttribute(hStdOutputHandle,
                          FOREGROUND_RED |
                          BACKGROUND_RED |
                          BACKGROUND_BLUE |
                          BACKGROUND_GREEN |
                          FOREGROUND_INTENSITY);
  ReadConsole(hStdInputHandle, cMyString, strlen(cMyString), &dwResult, NULL);
  return 0;
 }
```

Окно, отображаемое при запуске программы, показано на рис. 21.

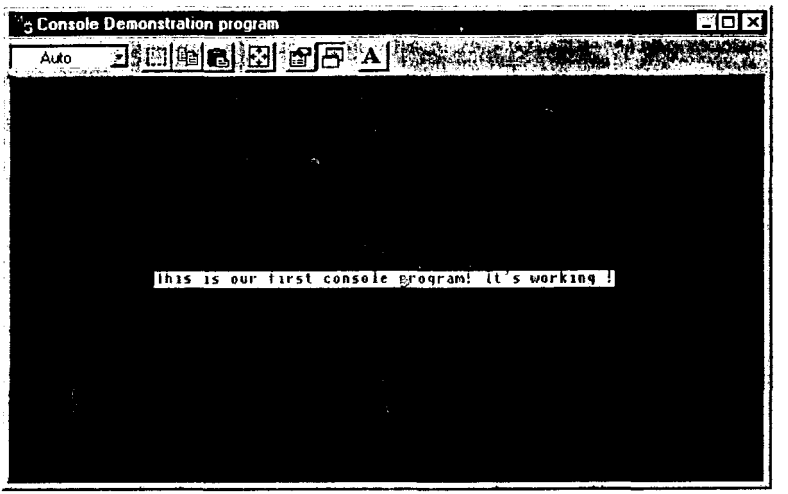


Рис. 21. Окно-консоль, в которое произведен вывод строки

Ничего особенного в этой программе не происходит. Программа запрашивает собственную консоль и устанавливает заголовок этой консоли, после чего получает хэндл потоков ввода и вывода. Курсор устанавливается с таким расчетом, чтобы выводимая фраза размещалась в центре экрана. Цвет выводимых символов устанавливается как ярко-красный на ярко-белом (на рисунке показана консоль программы именно после этого момента). После вывода фразы программа ждет нажатия Enter, после чего затирает фразу и предлагает пользователю что-нибудь набрать на экране, завершив набор нажатием клавиши Enter. Завершение ввода пользователя означает и завершение программы. Ничего сложного, но используются все функции управления консолью, которые мы уже изучили. На долю читателя я оставляю возможность самостоятельно исследовать, какие функции ввода - вывода из стандартных библиотек C и C++ могут быть использованы в консольных программах вместо функций API. Попробуйте, например, начать с исследования функции printf(). С другой стороны, приятной особенностью консольных программ является возможность вызова таких функций API, как, скажем, MessageBox(). Другими словами, при очень небольшой доработке большинство программ MS DOS могут быть перекомпилированы как консольные программы и могут использовать многие из тех возможностей, которые предоставляет

ОБРАБОТКА НАЖАТИЙ КЛАВИШ НА КЛАВИАТУРЕ И СОБЫТИЙ, ПРОИСШЕДШИХ С МЫШЬЮ

Мы уже знаем, что Windows - система, управляемая событиями. Консольные программы не являются исключением. Но есть одна деталь, которую необходимо отметить. События с клавиатурой и мышью записываются во входной буфер только в тех случаях, когда программа, во-первых, имеет клавиатурный фокус, и, во-вторых, указатель мыши находится в рабочей области консольного окна. Каждому событию, произошедшему с консолью, соответствует одна запись во входном буфере. Каждая запись о событии представляет собой заполненную структуру типа _INPUT_RECORD, описание которой можно найти в файле wincon.h:

```
typedef struct _INPUT_RECORD {
  WORD EventType;
  union {
        KEY_EVENT_RECORD KeyEvent;
        MOUSE_EVENT_RECORD MouseEvent;
        WINDOW_BUFFER_SIZE_RECORD WindowBufferSizeEvent;
        MENU_EVENT_RECORD MenuEvent;
        FOCUS_EVENT_RECORD FocusEvent;
  } Event;
} INPUT_RECORD, *PINPUT_RECORD;
```

Даже не особо вникая в смысл полей, видно, что консоль обрабатывает пять типов событий. Их перечень, взятый из файла wincon.h, приведен в табл. 58.

Таблица 58. События, обрабатываемые консолью

Флаг	Значение	Эффект
KEY_EVENT	0x0001	Событие с клавиатурой
MOUSE_EVENT	0x0002	Событие с мышью
WINDOW_BUFFER_SIZE_EVENT	0x0004	Событие по изменению размеров экрана
MENU_EVENT	0x0008	Событие с меню
FOCUS_EVENT	0x0010	Изменение фокуса

Обычно события меню, фокуса устройства и изменения размеров экрана обрабатываются системой, на долю программиста остаются события с клавиатурой и мышью. Всю информацию о событии можно получить с помощью функции ReadConsoleInput(). По привычке приводим ее описание из файла wincon.h:

```
WINBASEAPI BOOL WINAPI ReadConsoleInputA(
                        HANDLE hConsoleInput,
                        PINPUT_RECORD lpBuffer,
                        DWORD nLength,
                        LPDWORD lpNumberOfEventsRead);
WINBASEAPI BOOL WINAPI ReadConsoleInputW(
                        HANDLE hConsoleInput,
                        PINPUT_RECORD lpBuffer,
                        DWORD nLength,
                        LPDWORD lpNumberOfEventsRead);
#ifdef UNICODE
#define ReadConsoleInput  ReadConsoleInputW
#else
#define ReadConsoleInput  ReadConsoleInputA
#endif // !UNICODE
```

Здесь:

hConsoleInput - хэндл входного потока консоли;

lpBuffer - указатель на структуру типа INPUT_RECORD, в которую будут записаны данные о событии (или массив структур, если считываются данные более чем об одном событии);

nLength - число считываемых записей о событии;

lpNumberOfEventsRead - указатель на двойное слово, в которое записывается число реально считанных данных.

До нормальной работы осталось немного - узнать, какая информация записывается в структуру типа INPUT_RECORD и как мы можем ее использовать. Давайте остановимся на каждом типе событий отдельно.

События с клавиатурой

События клавиатуры генерируются каждый раз при нажатии клавиши. При этом поле EventType структуры типа _INPUT_RECORD содержит значение KEY_EVENT, а в объединение Event записывается поле KeyEvent типа KEY_EVENT_RECORD. Этот тип определен в wincon.h:

```
typedef struct _KEY_EVENT_RECORD {
  BOOL bKeyDown;
  WORD wRepeatCount;
  WORD wVirtualKeyCode;
  WORD wVirtualScanCode;
  union {
        WCHAR UnicodeChar;
        CHAR  AsciiChar;
      } uChar;
  DWORD dwControlKeyState;
} KEY_EVENT_RECORD, *PKEY_EVENT_RECORD;
```

Для того чтобы нормально обрабатывать события с клавиатурой, нам необходимо подробно разобрать назначение полей этой структуры. Программисты, знакомые с обработкой клавиатурных прерываний в DOS, увидят здесь множество знакомых характеристик. К сожалению, рамки этой книги не позволяют мне описать основы работы с клавиатурой. Если читатель чувствует, что у него в этой области есть пробел, рекомендую изучить этот вопрос по другим изданиям.

Если событие с клавиатурой состояло в нажатии клавиши, то поле bKeyDown принимает значение TRUE. Значение FALSE говорит о том, что произошло отжатие клавиши.

Если клавиша нажата и код клавиши начал генерироваться повторно, поле wRepeatCount является счетчиком повторов, что, кстати, следует и из его названия.

Виртуальный код нажатой клавиши записывается в поле wVirtualKeyCode, а виртуальный скан-код - в поле wVirtualScanCode.

Объединение uChar содержит ASCII или Unicode код нажатой клавиши в зависимости от того, какая версия функции ReadConsoleInput(), ASCII или Unicode, используется.

И наконец, поле dwControlKeyState указывает на состояние управляющих клавиш. Их возможные значения приведены в табл. 59.

Таблица 59. Флаги состояния управляющих клавиш

Флаг	Значение	Эффект
RIGHT_ALT_PRESSED	0x0001	Нажат правый Alt
LEFT_ALT_PRESSED	0x0002	Нажат левый Alt
RIGHT_CTRL_PRESSED	0x0004	Нажат правый Ctrl
LEFT_CTRL_PRESSED	0x0008	Нажат левый Ctrl
SHIFT_PRESSED	0x0010	Нажат Shift
NUMLOCK_ON	0x0020	NumLock горит
SCROLLLOCK_ON	0x0040	ScrollLock горит
CAPSLOCK_ON	0x0080	CapsLock горит
ENHANCED_KEY	0x0100	Клавиша с двойным скан-кодом

Демонстрационная программа

Я думаю, что даже конспективного изложения достаточно для того, чтобы можно было начать работу с клавиатурой в консольной сессии. Разве не так, уважаемый читатель? Тем не менее, я привожу демонстра-

ционную программу для того, чтобы показать, как можно обрабатывать клавиатурные события.

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

main()
 {
  HANDLE hStdInputHandle, hStdOutputHandle;
  char cMyMessage[80] = "Do something with mouse to exit";
  COORD Coord = {0, 24};
  DWORD dwResult;
  BOOL bMyFlag = TRUE;
  _INPUT_RECORD InputRecord;
  char cMyString[16];
  char* cMyKeys[9] = {" RAlt", " LAlt", " RCtrl", " LCtrl", " Shift", " NumLock",
                      " ScrollLock", " CapsLock", " EnhKey"};
  DWORD dwMyFlag;

  FreeConsole();
  AllocConsole();
  SetConsoleTitle("Keyboard in console session demo program");
  hStdInputHandle = GetStdHandle(STD_INPUT_HANDLE);
  hStdOutputHandle = GetStdHandle(STD_OUTPUT_HANDLE);
  SetConsoleCursorPosition(hStdOutputHandle, Coord);
  SetConsoleTextAttribute(hStdOutputHandle, FOREGROUND_RED |
                                            FOREGROUND_GREEN |
                                            FOREGROUND_BLUE);
 WriteConsole(hStdOutputHandle, cMyMessage, strlen(cMyMessage), &dwResult,
              NULL);
  while(bMyFlag)
   {
    ReadConsoleInput(hStdInputHandle, &InputRecord, 1, &dwResult);
    if(dwResult >= 1)
      {
       if(InputRecord.EventType == KEY_EVENT)
         {
          SetConsoleCursorPosition(hStdOutputHandle, Coord);
          SetConsoleTextAttribute(hStdOutputHandle,0);
          WriteConsole(hStdOutputHandle, cMyMessage, strlen(cMyMessage),
                       &dwResult, NULL);
          for( int i = 0; i < 80; i++)
            cMyMessage[i] = 0;
          Coord.X = 0;
          Coord.Y = 1;
          SetConsoleCursorPosition(hStdOutputHandle, Coord);
          if(InputRecord.Event.KeyEvent.bKeyDown)
            strcat(cMyMessage, "Pressed ");
           else
```

```
            strcat(cMyMessage, "Released ");
          strcat(strcat(cMyMessage,
                        itoa(InputRecord.Event.KeyEvent.wVirtualKeyCode,
                             cMyString, 16)), " ");
          strcat(cMyMessage,
                 itoa(InputRecord.Event.KeyEvent.wVirtualScanCode,
                      cMyString, 16));
          if(InputRecord.Event.KeyEvent.dwControlKeyState != 0)
          for(int i = 0; i <= 8; i++)
           {
            dwMyFlag = 1;
            if(InputRecord.Event.KeyEvent.dwControlKeyState &
              (dwMyFlag << i) )
             strcat(cMyMessage, cMyKeys[i]);
           }
          SetConsoleTextAttribute(hStdOutputHandle, FOREGROUND_RED |
                                  FOREGROUND_GREEN |
                                  FOREGROUND_BLUE);
          WriteConsole(hStdOutpu tHandle, cMyMessage, strlen(cMyMessage),
                       &dwResult, NULL);
        }
       else
        if(InputRecord.EventType == MOUSE_EVENT)
          bMyFlag = FALSE;
    }
 return 0;
}
```

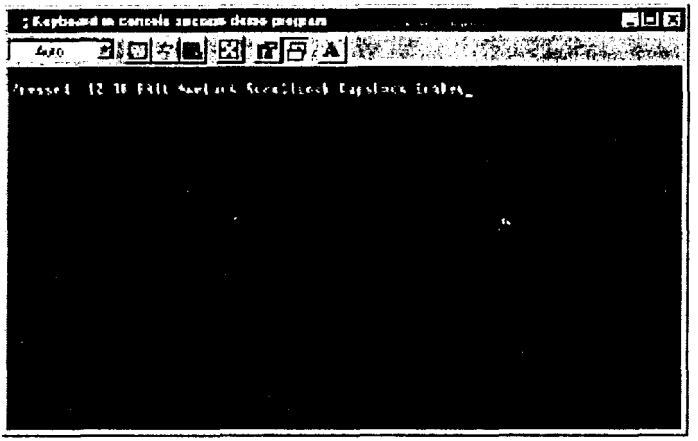


Рис. 22. Окно-консоль, отображающее положение курсора мыши и состояние клавиатуры

На рис. 22 показан вид окна, созданного этой программой.

Эта программа запрашивает для себя отдельную консоль, после чего в нижней части экрана выдает сообщение о том, для выхода необходимо сделать что-либо с мышкой. Но суть ее не в этом. При нажатии любой клавиши в верхней части экрана появляется строка, в которой указывается, какой тип действия (нажатие или отжатие клавиши) был произведен с клавиатурой, а также перечисляются некоторые характеристики нажатой клавиши, как-то ее виртуальный и скан-коды и состояние управляющих клавиш. Я думаю, что каких-либо трудностей при разборе программы не встретится.

События с мышью

События с мышью происходят в тех случаях, когда мышь двигается (при этом курсор должен находиться поверх окна консоли), либо на ней нажимается одна или более кнопок. При возникновении события с мышью поле EventType структуры типа _INPUT_RECORD содержит значение MOUSE_EVENT, В объединении Event этой структуры в этом случае будет содержаться поле MouseEvent типа MOUSE_EVENT_RECORD. Для того чтобы понять, какую информацию мы можем извлечь из события с мышью, рассмотрим описание типа MOUSE_EVENT_RECORD. Его мы извлечем из заголовочного файла wincon.h:

```
typedef struct _MOUSE_EVENT_RECORD {
  COORD dwMousePosition;
  DWORD dwButtonState;
  DWORD dwControlKeyState;
  DWORD dwEventFlags;
} MOUSE_EVENT_RECORD, *PMOUSE_EVENT_RECORD;
```

В этой структуре некоторые поля нам уже знакомы. Первое поле dwMousePosition типа COORD - координаты курсора мыши во время наступления события. Если обычно координаты курсора указываются в пикселах, то в данном случае они указываются в символах, причем начало отсчета - левый верхний угол рабочей области окна консоли. Не забудьте, экран-то текстовый!

Поле dwButtonState описывает состояние кнопок мыши на момент возникновения события. Кодируется это поле достаточно замысловато. Считается, что максимум у мыши может быть четыре кнопки (лично я таких мышей не видел и не слышал о них. Может, Microsoft боится повторить ситуацию с 640 кбайтами в DOS?). При этом младший бит определяет состояние самой левой клавиши (1 - клавиша нажата), сле-

дующий по старшинству бит определяет состояние самой правой клавиши. Очередной бит связан со второй слева кнопкой, следующий - с третьей слева и, наконец, последний - с четвертой слева кнопкой. Для каждого из этих битов в файле wincon.h определены макросы, которые приведены в табл. 60.

Таблица 60. Флаги, определяющие нажатую клавишу мыши

Макрос	Значение
FROM_LEFT_1ST_BUTTON_PRESSED	0x0001
RIGHTMOST_BUTTON_PRESSED	0x0002
FROM_LEFT_2ND_BUTTON_PRESSED	0x0004
FROM_LEFT_3RD_BUTTON_PRESSED	0x0008
FROM_LEFT_4TH_BUTTON_PRESSED	0x0010

Таблица 61. События от мыши

Флаг	Значение	Эффект
MOUSE_MOVED	0x0001	Перемещение мыши
DOUBLE_CLICK	0x0002	Второй щелчок кнопки (при двойном щелчке)

До чего же приятно работать с макросами, название которых определяет их назначение!

С полем dwControlKeyState мы познакомились при изучении работы с клавиатурой. Никаких изменений это поле по сравнению с аналогичным в структуре KEY_EVENT_RECORD не претерпело.

Значение последнего поля, dwEventFlags, определяет действие, которое привело к возникновению события. Если его значение равно нулю, то это означает, что была нажата или отпущена одна из кнопок мыши. Еще два возможных значения этого поля приведены в табл. 61.

Не напоминает ли это все нотификационные события при разработке оконных программ?

Демонстрационная программа

Думаю, что все дальнейшие объяснения излишни. Вспомним о принципе «Seeing is believing». Давайте разберем небольшую демонстрационную программу. При написании этой программы я, чтобы не утомлять читателя, сделал одно допущение: у мыши всего две кнопки. Надеюсь, это допущение не повлияет на восприятие программы читателем:

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

```
main()
 {

  HANDLE hStdInputHandle, hStdOutputHandle;
  COORD Coord = {0,24};
  char cMyMessage[80] = "Press any key to exit";
  DWORD dwResult;
  BOOL bMyFlag = TRUE;
  _INPUT_RECORD InputRecord;
  char cMyString[16];
  char* cMyButtons[4] = {" LeftButton", " RightButton",
                         " Mouse moved", " Double Click"};
  DWORD dwMyFlag;

  FreeConsole();
  AllocConsole();
  SetConsoleTitle("Mouse in console session demo program");
  hStdInputHandle = GetStdHandle(STD_INPUT_HANDLE);
  hStdOutputHandle = GetStdHandle(STD_OUTPUT_HANDLE);
  SetConsoleCursorPosition(hStdOutputHandle, Coord);
  SetConsoleTextAttribute(hStdOutputHandle, FOREGROUND_RED |
                                            FOREGROUND_GREEN |
                                            FOREGROUND_BLUE);
  WriteConsole(hStdOutputHandle, cMyMessage, strlen(cMyMessage),
               &dwResult, NULL);
  while(bMyFlag)
   {
    ReadConsoleInput(hStdInputHandle, &InputRecord, 1, &dwResult);
    if(dwResult >= 1)
      {
       if(InputRecord.EventType == MOUSE_EVENT)
         {
          SetConsoleCursorPosition(hStdOutputHandle, Coord);
          SetConsoleTextAttribute(hStdOutputHandle,0);
          WriteConsole(hStdOutputHandle, cMyMessage, strlen(cMyMessage),
                       &dwResult, NULL);
          for( int i = 0; i < 80; i++)
            cMyMessage[i] = 0;
          Coord.X = 0;
          Coord.Y = 1;
          SetConsoleCursorPosition(hStdOutputHandle, Coord);
          strcat(cMyMessage, "Position - ");
          strcat(cMyMessage,
                 itoa(InputRecord.Event.MouseEvent.dwMousePosition.X,
                      cMyString, 10));
          strcat(cMyMessage, ", ");
          strcat(cMyMessage,
                 itoa(InputRecord.Event.MouseEvent.dwMousePosition.Y,
                      cMyString, 10));
```

```
            strcat(cMyMessage, " ");
            for(int i = 0; i <= 1; i++)
             {
               dwMyFlag = 1;
               if(InputRecord.Event.MouseEvent.dwButtonState & (dwMyFlag << i))
                 strcat(cMyMessage, cMyButtons[i]);
               if(InputRecord.Event.MouseEvent.dwEventFlags & (dwMyFlag << i))
                 strcat(cMyMessage, cMyButtons[i+2]);
             }
            SetConsoleTextAttribute(hStdOutputHandle, FOREGROUND_RED |
                                                      FOREGROUND_GREEN |
                                                      FOREGROUND_BLUE);
            WriteConsole(hStdOutputHandle, cMyMessage, strlen(cMyMessage),
                         &dwResult, NULL);
          }
         else
          if(InputRecord.EventType == KEY_EVENT)
            bMyFlag = FALSE;
       }

   }
  return 0;
}
```

И, как всегда, вид окна, созданного программой (рис. 23).

Рис. 23. Окно-консоль, отображающее положение и состояние мыши

Как и в предыдущей программе, здесь нет ничего особенного. При запуске программа сообщает пользователю, что для выхода необходимо нажать любую клавишу. При возникновении же любого события с мышью, в первой строке экрана появляется информация о событии. Все данные берутся из структуры типа _INPUT_RECORD. Попробуйте подвигать мышь внутри окна и понажимать кнопки мыши, понаблюдайте за результатами.

КРЮЧКИ (ХУКИ)

А сейчас мне хотелось бы поговорить с уважаемым читателем на достаточно интересную тему, которой почему-то уделяется очень мало внимания. Однако эта тема очень важна для понимания механизма работы Windows. Используя технику хуков, любознательный читатель сможет заглянуть непосредственно во внутренности Windows, и, соответственно, более глубоко понять принципы работы системы.

Итак, что же такое хуки? Хуки – это своеобразный механизм в Windows, который позволяет приложению производить обработку некоторых сообщений до того, как они будут переданы на обработку, скажем, DefWindowProc(). Другими словами, хук – это небольшая программка, которая берет на себя обработку сообщений ДО ТОГО, КАК СООБЩЕНИЕ БУДЕТ ОБРАБОТАНО СИСТЕМОЙ. Из этого утверждения вытекает одно очень важное следствие. Раз подключение хуков приводит к увеличению объёма работы системы, то, очевидно, работа системы в случае использования хуков ЗАМЕДЛЯЕТСЯ, следовательно, хуки должны устанавливаться только в самых крайних случаях и удаляться сразу же после того, как надобность в них отпала.

Схематично мы можем показать работу хуков следующим образом:

Сообщение---------- ----------→Система

Таким образом, происходит обычная обработка сообщения.

Сообщение→Хук1→Хук2→Хук3→------→Хукn→Система

А таким образом происходит обработка сообщения в случае применения хуков.

Любознательный читатель, посмотрев на эту схему, может задать вопрос: так что же, одно сообщение может обрабатываться несколькими ху-

ками? Да, может! Более того, именно такой механизм и предусмотрен в Windows и последовательности хуков так и называются – цепочки хуков. Если в недрах Windows возникает сообщение, ассоциированное с каким-то хуком, то Windows сначала передает его на обработку первому хуку, потом второму и так далее. После того, как все хуки отработают, сообщение обрабатывается так, как оно обрабатывалось бы в обычном случае.

Следующий вопрос - что необходимо знать для того, чтобы написать нормальную процедуру хука?

Во-первых, программист должен знать, какие типы хуков используются в Windows. Типы хуков, которые используются в Windows, я привожу в таблице:

Тип хука	Код хука
WH_MIN	(-1)
WH_MSGFILTER	(-1)
WH_JOURNALRECORD	0
WH_JOURNALPLAYBACK	1
WH_KEYBOARD	2
WH_GETMESSAGE	3
WH_CALLWNDPROC	4
WH_CBT	5
WH_SYSMSGFILTER	6
WH_MOUSE	7
WH_HARDWARE	8
WH_DEBUG	9
WH_SHELL	10
WH_FOREGROUNDIDLE	11
WH_CALLWNDPROCRET	12
WH_MAX	12
WH_MINHOOK	WH_MIN
WH_MAXHOOK	WH_MAX

Во-вторых, программисту необходимо знать, каким образом можно установить хук? Для того чтобы решить эту задачу, приложение должно вызвать функцию SetWindowsHookEx(), которая в файле winuser.h описана так:

```
WINUSERAPI HHOOK WINAPI SetWindowsHookExA(int idHook,
                                          HOOKPROC lpfn,
                                          HINSTANCE hmod,
```

```
                                                DWORD dwThreadId);
WINUSERAPI HHOOK WINAPI SetWindowsHookExW(int idHook,
                                                HOOKPROC lpfn,
                                                HINSTANCE hmod,
                                                DWORD dwThreadId);
#ifdef UNICODE
#define SetWindowsHookEx  SetWindowsHookExW
#else
#define SetWindowsHookEx  SetWindowsHookExA
#endif // !UNICODE
```

Первый аргумент этой функции, idHook, должен определять тип хука. Второй аргумент, lpfn, должен указывать непосредственно на процедуру хука. Что из этого следует? То, что процедура хука должна иметь строго определённое количество и тип аргументов. Тип этой процедуры описан в файле winuser.h так:

```
typedef LRESULT (CALLBACK* HOOKPROC)
                        (int code, WPARAM wParam, LPARAM lParam);
```

Так как назначение аргументов функции в зависимости от типа хука разные, мы поговорим о них чуть позже. А пока... Я откровенно устал от описания бесконечных функций, структур, полей... Может быть, настало время для того, чтобы и автор, и читатель несколько отдохнули и выпили бы по чашечке чая? Я предпочитаю всем другим сортам «Эрл Грей» № 1. Вкус – великолепный, отлично согревает, «недостаток» только один – чай «утренний», т. е. содержащий достаточно много тонизирующих веществ, поэтому не советую пить его на ночь.

А теперь, как говорится, вернемся к нашим баранам, т. е. к аргументам функции SetWindowsHookEx(). Третий аргумент, hMod, должен либо хранить хэндл DLL, в которой находится процедура хука, либо быть равным NULL, если процедура хука находится внутри кода текущего процесса. И наконец, последний аргумент, dwThreadId, указывает номер потока, в котором находится процедура хука. Если этот аргумент равен нулю, то это означает, что процедура хука находится внутри текущего потока.

Если возвращаемое функцией SetWindowsHookEx() значение равно NULL, то программист должен найти ошибку в своей программе и опреде-

лить, почему хук не инсталлировался нормально. Если возвраемое значение не равно NULL, то оно является хэндлом установленного хука.

Но я уже сказал о том, что сразу после того, как надобность в хуке отпала, хук должен быть тут же деинсталлирован. Для этого необходимо обратиться к функции UnhookWindowsHookEx(), которая в файле winuser.h описана следующим образом:

```
WINUSERAPI BOOL WINAPI UnhookWindowsHookEx(HHOOK hhk);
```

Единственным аргументом этой функции является хэндл установленного ранее хука.

А теперь я хотел бы остановиться на каждом типе хука более подробно, а также рассмотреть возможности, предоставляемые тем или иным видом хука.

Хуки типов WH_CALLWNDPROC и WH_CALLWNDPROCRET.

Хуки типов WH_CALLWNDPROC и WH_CALLWNDPROCRET позволяют приложению осуществлять обработку тех сообщений, которые были посланны оконной процедуре при помощи функции SendMessage(). Windows вызывает процедуру хука типа WH_CALLWNDPROC ДО передачи сообщения оконной процедуре. После этого происходит передача сообщения на обработку оконной процедуре. И после того, как оконная процедура закончила свою работу, Windows вызывает процедуру хука типа WH_CALLWNDPROCRET. При этом процедуре хуку WH_CALLWNDPROCRET передаётся адрес структуры типа CWPRETSTRUCT, которая описана в файле winuser.h следующим образом:

```
typedef struct tagCWPRETSTRUCT {
  LRESULT lResult;
  LPARAM  lParam;
  WPARAM  wParam;
  UINT    message;
  HWND    hwnd;
} CWPRETSTRUCT, *PCWPRETSTRUCT, NEAR *NPCWPRETSTRUCT,
  FAR *LPCWPRETSTRUCT;
```

Первое поле этой структуры, lResult, содержит значение, которое было возвращено оконной процедурой после обработки сообщения. Второе поле содержит lParam сообщения. Третье поле содержит wParam сообщения. Поле message определяет тип сообщения, и, наконец, поле hwnd определяет окно, которому было передано на обработку сообщение.

Но, наверное, читателю уже наскучили бесконечные описания структур и полей. Не пора ли перейти к демонстрационной программе? Пусть наша первая демонстрационная программа всего-навсего перехватывает сообщения и выдаёт информацию о параметрах сообщения в файл с именем asdfghjk.lkj.

Ниже я привожу текст демонстрационной программы.

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

HINSTANCE hInst;
HHOOK hHook;
HANDLE hFile;

LRESULT CALLBACK HooksWndProc (HWND, UINT, UINT, LONG);
LRESULT CALLBACK CallWndProc(int, WPARAM, LPARAM);

int APIENTRY WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                      LPSTR lpszCmdParam, int nCmdShow )
 {
  HWND hWnd ;
  WNDCLASS WndClass ;
  MSG Msg;

  hInst = hInstance;
/* Registering our window class */
/* Fill WNDCLASS structure */
  WndClass.style = CS_HREDRAW | CS_VREDRAW;
```

```
WndClass.lpfnWndProc = (WNDPROC) HooksWndProc;
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 = "WH_CALLWNDPROCExample";
if ( !RegisterClass(&WndClass) )
  {
   MessageBox(NULL,"Cannot register class","Error",MB_OK);
   return 0;
  }
hWnd = CreateWindow("WH_CALLWNDPROCExample", "Hooks example",
            WS_OVERLAPPEDWINDOW,
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            NULL, NULL,
            hInstance,NULL);
if(!hWnd)
  {
   MessageBox(NULL,"Cannot create window","Error",MB_OK);
   return 0;
  }
ShowWindow(hWnd,nCmdShow);
UpdateWindow(hWnd);
```

```
/* Beginning of messages cycle */
  while(GetMessage(&Msg, NULL, 0, 0))
   {
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
   }
  return Msg.wParam;
 }

LRESULT CALLBACK HooksWndProc (HWND hWnd, UINT Message, UINT
                                        wParam, LONG lParam)
 {
  switch(Message)
   {
    case WM_CREATE:
     hFile = CreateFile("asdfghjk.lkj", GENERIC_WRITE, 0, NULL,
                    CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0),
     hHook = SetWindowsHookEx(WH_CALLWNDPROC,
                                (HOOKPROC) CallWndProc,
                                NULL, GetCurrentThreadId());
     return 0;
    case WM_DESTROY:
     UnhookWindowsHookEx(hHook);
     CloseHandle(hFile);
     PostQuitMessage(0);
     return 0;
    default:
     return DefWindowProc(hWnd, Message, wParam, lParam);
   }
```

```
}

LRESULT CALLBACK CallWndProc(int nCode, WPARAM wParam,
                             LPARAM lParam)
{
  char cBuffer[0x80];
  DWORD dwNumberOfBytesWritten;

  Beep(440, 0.2);
  if(nCode < 0)
    return CallNextHookEx(hHook, nCode, wParam, lParam);
   else
    if(nCode == HC_ACTION)
      {
// Записываем в файл информацию о сообщении.
        sprintf(cBuffer, "nCode - %08x ", nCode);
        WriteFile(hFile, cBuffer, 17, &dwNumberOfBytesWritten, NULL);
        sprintf(cBuffer, "wParam - %08x \n", wParam);
        WriteFile(hFile, cBuffer, 19, &dwNumberOfBytesWritten, NULL);
        sprintf(cBuffer, "PCWPSTRUCT->lParam - %08x \n",
              ((PCWPSTRUCT) lParam)->lParam);
        WriteFile(hFile, cBuffer, 31, &dwNumberOfBytesWritten, NULL);
        sprintf(cBuffer, "PCWPSTRUCT->wParam - %08x \n",
              ((PCWPSTRUCT) lParam)->wParam);
        WriteFile(hFile, cBuffer, 31, &dwNumberOfBytesWritten, NULL);
        sprintf(cBuffer, "PCWPSTRUCT->message - %08x \n",
              ((PCWPSTRUCT) lParam)->message);
        WriteFile(hFile, cBuffer, 32, &dwNumberOfBytesWritten, NULL);
        sprintf(cBuffer, "PCWPSTRUCT->hwnd - %08x \n",
```

```
            ((PCWPSTRUCT) lParam)->hwnd);
        WriteFile(hFile, cBuffer, 29, &dwNumberOfBytesWritten, NULL);
        return CallNextHookEx(hHook, nCode, wParam, lParam);
    }
}
```

Итак, давайте попробуем рассмотреть эту программу более подробно. Я не стану останавливаться на всех деталях, скажу только, что при обработке сообщения WM_CREATE мы создаем файл с уникальным, надеюсь, именем. Тут же мы устанавливаем хук типа WH_CALLWNDPROC и указываем процедуру хука – CallWndProc(). При закрытии окна (сообщение WM_DESTROY) мы отменяем этот хук.

Если первый аргумент функции CallWndProc() принимает значение HC_ACTION, процедура хука должна обрабатывать сообщение. В противном случае мы должны передать сообщение на обработку следующему хуку в цепочке при помощи вызова функции CallNextHookEx() и вернуть значение, возвращенное этой функцией. Если второй аргумент не равен нулю, то сообщение послано текущим процессом. Если сообщение послано иным процессом, второй аргумент функции равен нулю. Наибольший интерес представляет третий аргумент, который является указателем на структуру типа CWPSTRUCT, которая в файле winuser.h описана следующим образом:

```
typedef struct tagCWPSTRUCT {
  LPARAM  lParam;
  WPARAM  wParam;
  UINT    message;
  HWND    hwnd;
} CWPSTRUCT, *PCWPSTRUCT, NEAR *NPCWPSTRUCT,
  FAR *LPCWPSTRUCT;
```

А вот здесь-то и может порезвиться программист! Первое поле этой структуры – lParam перехватываемого сообщения, второе поле – это wParam перехватываемого сообщения, третье поле – номер сообщения, а четвёртое – хэндл окна, которому послано сообщение. Все эти параметры в программе фиксируются и записываются в тот самый файл с уникальным именем, который мы создали при обработке сообщения WM_CREATE. Кстати, если у читателя на компьютере установлена звуковая карта, то

каждый вход в процедуру хука этой программы читатель сможет не только увидеть при анализе содержимого упомянутого файла, но и услышать. ☺

К сожалению, CallWndProc() может анализировать параметры сообщения, но не может модифицировать их. И всегда, в любом случае функция CallWndProc() должна возвращать нулевое значение.

А теперь давайте в демонстрационной программе, не изменяя функцию WinMain(), несколько изменим функции HooksWndProc(). Изменения будут заключаться только в том, что вместо хука типа WH_CALLWNDPROC мы установим хук типа WH_CALLWNDPROCRET. Естественно, нам придётся переписать и функцию CallWndProc(). В новом варианте программы она будет иметь следующий вид:

```
LRESULT CALLBACK CallWndProc(int nCode, WPARAM wParam,
                                        LPARAM lParam)
 {
  char cBuffer[0x80];
  DWORD dwNumberOfBytesWritten;

  Beep(440, 0.2);
  if(nCode < 0)
    return CallNextHookEx(hHook, nCode, wParam, lParam);
   else
    if(nCode == HC_ACTION)
      {
// Записываем в файл информацию о сообщении.
        sprintf(cBuffer, "nCode - %08x ", nCode);
        WriteFile(hFile, cBuffer, 17, &dwNumberOfBytesWritten, NULL);
        sprintf(cBuffer, "wParam - %08x \n", wParam);
        WriteFile(hFile, cBuffer, 19, &dwNumberOfBytesWritten, NULL);
        sprintf(cBuffer, "PCWPRETSTRUCT->lResult - %08x \n",
              ((PCWPRETSTRUCT) lParam)->lResult);
        WriteFile(hFile, cBuffer, 35, &dwNumberOfBytesWritten, NULL);
        sprintf(cBuffer, "PCWPRETSTRUCT->lParam - %08x \n",
```

```
            ((PCWPRETSTRUCT) lParam)->lParam);
        WriteFile(hFile, cBuffer, 34, &dwNumberOfBytesWritten, NULL);
        sprintf(cBuffer, "PCWPRETSTRUCT->wParam - %08x \n",
            ((PCWPRETSTRUCT) lParam)->wParam);
        WriteFile(hFile, cBuffer, 34, &dwNumberOfBytesWritten, NULL);
        sprintf(cBuffer, "PCWPRETSTRUCT->message - %08x \n",
            ((PCWPRETSTRUCT) lParam)->message);
        WriteFile(hFile, cBuffer, 35, &dwNumberOfBytesWritten, NULL);
        sprintf(cBuffer, "PCWPRETSTRUCT->hwnd - %08x \n",
            ((PCWPRETSTRUCT) lParam)->hwnd);
        WriteFile(hFile, cBuffer, 32, &dwNumberOfBytesWritten, NULL);
        return CallNextHookEx(hHook, nCode, wParam, lParam);
      }
}
```

Что изменилось? Процедуре хука типа WH_CALLWNDPROCRET в качестве третьего аргумента передаётся указатель на структуру типа CWPRETSTRUCT. Поля со второго по пятое этой структуру полностью идентичны полям структуры типа CWPSTRUCT. А в первом поле структуры записывается код возврата, выработанный оконной процедурой обрабатывающей сообщение. Процедура хука в нашей демонстрационной программе записывает все эти данные в файл с уникальным именем. Пример данных, записанных в файл нашей процедурой хука, я привожу ниже.

```
nCode - 00000000 wParam - 00000000
PCWPRETSTRUCT->lResult - 00000000
PCWPRETSTRUCT->lParam - 00000000
PCWPRETSTRUCT->wParam - 00000001
PCWPRETSTRUCT->message - 00000018
PCWPRETSTRUCT->hwnd - 00000f90
nCode - 00000000 wParam - 00000000
PCWPRETSTRUCT->lResult - 00000000
PCWPRETSTRUCT->lParam - 0064fb84
```

PCWPRETSTRUCT->wParam - 00000000

PCWPRETSTRUCT->message - 00000046

PCWPRETSTRUCT->hwnd - 00000f90

nCode - 00000000 wParam - 00000001

PCWPRETSTRUCT->lResult - 00000000

PCWPRETSTRUCT->lParam - 00000000

PCWPRETSTRUCT->wParam - 00000f90

PCWPRETSTRUCT->message - 00000434

PCWPRETSTRUCT->hwnd - 000000e4

Естественно, что любой программист может представлять эти данные так, как ему хочется, всё ограничивается только фантазией программиста.

ХУКИ ТИПА WH_CBT

Хуки этого типа используются для того, чтобы перехватывать сообщения о создании окон, переходе окон в активное состояние, удалении, минимизации и максимизации их, изменении размеров и перемещении окон. Эти хуки получают управление перед выполнением системных команд, перед удалением событий от мышки или клавиатуры из системной очереди. Возвращаемое этим хуком значение зависит от того, разрешает ли Windows выполнение процедуры хука. Обычно эти хуки используются в системах обучения с применением компьютера, но, естественно, их использование этим не ограничивается.

В данном случае определённый интерес представляют только значения аргументов, которые передаются процедуре хука этого типа. В качестве первого аргумента, кода типа хука, могут использоваться следующие значения:

Название	Значение	Назначение
HCBT_MOVESIZE	0	Окно готовится к перемещению или изменению размеров
HCBT_MINMAX	1	Окно готовится к минимизации или максимизации
HCBT_QS	2	
HCBT_CREATEWND	3	Окно готовится к созданию
HCBT_DESTROYWND	4	Окно готовится к удалению

HCBT_ACTIVATE	5	Окно готовится к активизации
HCBT_CLICKSKIPPED	6	Система удалила сообщение от мышки из системной очереди
HCBT_KEYSKIPPED	7	Система удалила сообщение от клавиатуры из системной очереди
HCBT_SYSCOMMAND	8	К выдаче готовится системная команда
HCBT_SETFOCUS	9	Окно готовится получить клавиатурный фокус

Итак, исходя из приведенной выше таблицы, мы можем определить, на какие события может отреагировать наша программа. Если первый аргумент функции меньше нуля, то наша процедура не должна обрабатывать сообщение, а должна передать его на обработку следующему хуку в цепочке при помощи функции CallNextHokEx() и вернуть значение, возвращенное этой функцией.

Интерпретация второго и третьего аргументов зависит от того, что передано процедуре в качестве первого аргумента.

Если в качестве первого аргумента функции хука передано значение HCBT_ACTIVATE, то второй аргумент является хэндлом окна, которое будет активизировано. Третий аргумент является указателем на структуру типа CBTACTIVATESTRUCT, которая в файле winuser.h описана следующим образом:

```
typedef struct tagCBTACTIVATESTRUCT
{
  BOOL   fMouse;
  HWND   hWndActive;
} CBTACTIVATESTRUCT, *LPCBTACTIVATESTRUCT;
```

Первое поле этой структуры определяет, является ли активизация окна результатом щелчка кнопки мышки. Если это поле равно TRUE, то окно было активизировано при помощи щелчка мышкой на нём. В противном случае активизация окна была вызвана какими-то другими причинами.

Второе поле, что следует из его названия, является хэндлом окна, которое готовится к активизации.

В том случае, если в качестве первого аргумента функции хука передано значение HCBT_CLICKSKIPPED, то второй аргумент содержит номер удаленного из системной очереди сообщения от мышки, а третий аргумент является указателем на структуру типа MOUSEHOOKSTRUCT, которая следующим образом описана в файле winuser.h:

```
typedef struct tagMOUSEHOOKSTRUCT {
    POINT   pt;
    HWND    hwnd;
    UINT    wHitTestCode;
    DWORD   dwExtraInfo;
} MOUSEHOOKSTRUCT, FAR *LPMOUSEHOOKSTRUCT,
*PMOUSEHOOKSTRUCT;
```

Первое поле этой структуры содержит указатель на структуру типа POINT, которую мы уже рассматривали раньше. В этой структуре определяются координаты той точки экрана, на которой находился курсор в момент выдачи сообщения. Хэндл окна, получившего сообщение от мышки, находится во втором поле этой структуры. Третье поле определяет, где находился курсор мышки в момент выдачи сообщения. Это поле может принимать следующие значения:

Название	Значение	Назначение
HTERROR	(-2)	То же самое, что и HTNOWHERE, но в этом случае функция DefWindwProc() выдаёт звук для того, чтобы сообщить об ошибке
HTTRANSPARENT	(-1)	На окне, закрытом другим окном
HTNOWHERE	0	На нерабочей части экрана или на разделяющей два окна линии
HTCLIENT	1	В клиентской области окна
HTCAPTION	2	На заголовке окна
HTSYSMENU	3	На системном меню или на кнопке закрытия дочернего окна
HTGROWBOX	4	На кнопке изменения размеров
HTSIZE	HTGROWBOX	
HTMENU	5	На меню
HTHSCROLL	6	На горизонтальной полосе прокрутки
HTVSCROLL	7	На вертикальной полосе прокрутки
HTMINBUTTON	8	На кнопке минимизации
HTMAXBUTTON	9	На кнопке максимизации
HTLEFT	10	На левой границе окна

HTRIGHT	11	На правой границе окна
HTTOP	12	На верхней горизонтальной границе окна
HTTOPLEFT	13	На левом верхнем углу границы окна
HTTOPRIGHT	14	На правом верхнем углу границы окна
HTBOTTOM	15	На нижней горизонтальной границе окна
HTBOTTOMLEFT	16	На левом нижнем углу границы окна
HTBOTTOMRIGHT	17	На правом нижнем углу границы окна
HTBORDER	18	На границе окна, у которого нет границы, позволяющей изменять его размеры
HTREDUCE	HTMINBUTTON	На кнопке минимизации
HTZOOM	HTMAXBUTTON	На кнопке максимизации
HTSIZEFIRS	HTLEFT	
HTSIZELAST	HTBOTTOMRIGHT	
HTOBJECT	19	
HTCLOSE	20	На кнопке закрытия окна
HTHELP	21	На кнопке помощи

И наконец, четвертое поле структуры содержит дополнительную информацию о сообщении, если таковая присутствует.

Если в качестве первого аргумента процедуре хука передаётся HCBT_CREATEWND, то второй аргумент является хэндлом создаваемого окна, а третий элемент указывает на структуры типа CBT_CREATEWND, описанную в файле winuser.h следующим образом:

```
typedef struct tagCBT_CREATEWNDA
{
  struct tagCREATESTRUCTA *lpcs;
  HWND        hwndInsertAfter;
} CBT_CREATEWNDA, *LPCBT_CREATEWNDA;
typedef struct tagCBT_CREATEWNDW
{
  struct tagCREATESTRUCTW *lpcs;
```

```
    HWND        hwndInsertAfter;
} CBT_CREATEWNDW, *LPCBT_CREATEWNDW;
```

Первое поле этой структуры является указателем на структуру типа CREATESTRUCT, о которой мы уже говорили, а второе является хэндлом окна, после которого (в Z-порядке) будет размещено создаваемое окно.

Очередной код – HCBT_DESTROYWND. Второй аргумент является хэндлом закрываемого окна, а третий аргумент должен всегда быть равен нулю.

При коде HCBT_KEYSKIPPED в качестве второго аргумента передается код виртуальной клавиши, а в качестве третьего аргумента – данные о нажатой клавише. Формат этих данных приведен в таблице:

Номер бита	Назначение
0 – 15	Счётчик нажатия клавиши
16-23	Скан-код клавиши
24	Имеет ли нажатая клавиша расширенный код
25-28	Зарезервированы для использования в будущем
29	1 – клавиши нажата, 0 – клавиша отжата
30	Предыдущее состояние клавиши. Всегда установлен в нуль для события отжатия клавиши
31	

Если первый аргумент равен HCBT_MINMAX, то второй аргумент является хэндлом окна, которое будет минимизировано или максимизировано, а третий аргумент – набором характеристик о характере отображения окна (SW_-характеристик). Эти характеристики мы рассмотрели при разборе функции ShowWindow().

При первом аргументе, равном HCBT_MOVESIZE, второй аргумент также содержит хэндл окна, размеры или положение которого на экране изменилось. Третий аргумент содержит указатель на структуру типа RECT, которая определяет повое положение, а следовательно, и размеры окна.

При коде HCBT_QS назначения второго и третьего аргументов, передаваемых функции хука не определены и всегда устанавливаются в ноль.

Если в качестве первого аргумента функции хука передано значениеHCBT_SETFOCUS, то второй аргумент является хэндлом окна, получающего фокус, а третий – хэндлом окна, которое теряет фокус.

И наконец, если первый аргумент функции хука имеет значение, равное HCBT_SYSCOMMAND, то второй аргумент определяет системную команду (SC_-команду). Если системная команда была отдана при помощи

клавиатуры, то третий аргумент не используется. Если же системная команда была отдана при помощи мышки, то младшее слово третьего аргумента содержит x-координату курсора, а старшее слово – y-координату курсора на момент возникновения команды.

А теперь, как всегда, настала очередь демонстрационной программы.

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

HINSTANCE hInst;
HHOOK hHook;
HANDLE hFile;

LRESULT CALLBACK HooksWndProc (HWND, UINT, UINT, LONG);
LRESULT CALLBACK CBTProc(int, WPARAM, LPARAM);

int APIENTRY WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPSTR lpszCmdParam, int nCmdShow )
 {
  HWND hWnd ;
  WNDCLASS WndClass ;
  MSG Msg;

  hInst = hInstance;
/* Registering our window class */
/* Fill WNDCLASS structure */
  WndClass.style = CS_HREDRAW | CS_VREDRAW;
  WndClass.lpfnWndProc = (WNDPROC) HooksWndProc;
  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 = "WH_CBTExample";
  if ( !RegisterClass(&WndClass) )
    {
     MessageBox(NULL,"Cannot register class","Error",MB_OK);
```

```
        return 0;
      }
    hWnd = CreateWindow("WH_CBTExample", "Hooks example",
                WS_OVERLAPPEDWINDOW,
                CW_USEDEFAULT,
                CW_USEDEFAULT,
                CW_USEDEFAULT,
                CW_USEDEFAULT,
                NULL, NULL,
                hInstance,NULL);
    if(!hWnd)
      {
       MessageBox(NULL,"Cannot create window","Error",MB_OK);
       return 0;
      }
    ShowWindow(hWnd,nCmdShow);
    UpdateWindow(hWnd);
  /* Beginning of messages cycle */
    while(GetMessage(&Msg, NULL, 0, 0))
     {
      TranslateMessage(&Msg);
      DispatchMessage(&Msg);
     }
    return Msg.wParam;
  }

  LRESULT CALLBACK HooksWndProc (HWND hWnd, UINT Message,
                                 UINT wParam, LONG lParam)
  {
    switch(Message)
     {
      case WM_CREATE:
       hFile = CreateFile("asdfghjk.lkj", GENERIC_WRITE, 0, NULL,
                          CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, 0);
       hHook = SetWindowsHookEx(WH_CBT, (HOOKPROC) CBTProc,
                                NULL, GetCurrentThreadId());
       return 0;
      case WM_DESTROY:
       UnhookWindowsHookEx(hHook);
       CloseHandle(hFile);
```

```
        PostQuitMessage(0);
        return 0;
      default:
        return DefWindowProc(hWnd, Message, wParam, lParam);
    }
  }

LRESULT CALLBACK CBTProc(int nCode, WPARAM wParam,
                                    LPARAM lParam)
  {
   char cBuffer[0x80];
   DWORD dwNumberOfBytesWritten;

   if(nCode < 0)
     return CallNextHookEx(hHook, nCode, wParam, lParam);
    else
     switch(nCode)
      {
       case HCBT_ACTIVATE:
        sprintf(cBuffer, "nCode = HCBT_ACTIVATE hWnd = %08x\n",
wParam);
        WriteFile(hFile, cBuffer, 38, &dwNumberOfBytesWritten, NULL);
        sprintf(cBuffer,
              "LPCBTACTIVATESTRUCT->fMouse -  %08x\n",
               ((LPCBTACTIVATESTRUCT) lParam)->fMouse);
        WriteFile(hFile, cBuffer, 39, &dwNumberOfBytesWritten, NULL);
        sprintf(cBuffer,
              "LPCBTACTIVATESTRUCT->hWndActive - %08x\n",
               ((LPCBTACTIVATESTRUCT) lParam)->hWndActive);
        WriteFile(hFile, cBuffer, 43, &dwNumberOfBytesWritten, NULL);
        break;
       case HCBT_CREATEWND:
        sprintf(cBuffer,  "nCode  =  HCBT_CREATEWND  hWnd  =
%08x\n",
              wParam);
        WriteFile(hFile, cBuffer, 39, &dwNumberOfBytesWritten, NULL);
        sprintf(cBuffer, "LPCBT_CREATEWND->lpcs - %08x\n",
              ((LPCBT_CREATEWND) lParam)->lpcs);
        WriteFile(hFile, cBuffer, 33, &dwNumberOfBytesWritten, NULL);
        sprintf(cBuffer,     "LPCBT_CREATEWND->hwndInsertAfter     -
%08x\n",
```

```
                ((LPCBT_CREATEWND) lParam)->hwndInsertAfter);
            WriteFile(hFile, cBuffer, 44, &dwNumberOfBytesWritten, NULL);
            break;

          case HCBT_DESTROYWND:
            sprintf(cBuffer, "nCode = HCBT_DESTROYWND hWnd =
%08x\n",
                    wParam);
            WriteFile(hFile, cBuffer, 40, &dwNumberOfBytesWritten, NULL);
            break;
          default:
            return CallNextHookEx(hHook, nCode, wParam, lParam);
        }
      return CallNextHookEx(hHook, nCode, wParam, lParam);
    }
```

ЗАКЛЮЧЕНИЕ

На этом книга завершена. Не знаю, для кого она оказалась более трудной – для читателя или для меня. Не знаю, какой получилась – хорошей или плохой, принесла она пользу или читатель пожалел о потерянном времени. Писал я ее с душой. Конечно, можно было бы рассказать о Windows намного больше, но, как мне кажется, цель достигнута – читатель получил первоначальные знания и знает, где искать дополнительную информацию.

Пусть читатель не судит меня строго.

Теперь я должен расстаться со своим читателем и мне немного грустно. Я не знаю, что я должен сказать: «До свидания» или «Прощайте». Надеюсь, что до свидания, мой читатель.

Автор будет рад любым отзывам о его книге и постарается ответить всем, приславшим сообщения.

В настоящее время связаться с автором можно по электронной почте (pavel@lantel.wsnet.ru) и по ICQ (Аське) (UIN 12883243, ник Pavel).

Список макросов, используемых для создания программ, способных работать как в кодировке ANSI, так и в кодировке Unicode

Макрос из tchar.h	Функция	Макрос из tchar.h	Функция
_tWinMain	WinMain	_puttchar	putchar
_targv	_argv	_tputenv	putenv
_tenviron	_environ	_putts	puts
_tfullpath	_fullpath	_tremove	remove
_tmakepath	_makepath	_trename	rename
_tpopen	_popen	_trmdir	rmdir
_tsplitpath	_splitpath	_tscanf	scanf
_tstrdate	_strdate	_tsetlocale	setlocale
_tstrtime	_strtime	_tsopen	sopen
_taccess	_access	_tspawnl	spawnl
_tasctime	asctime	_tspawnle	spawnle
_ttoi	atoi	_tspawnlp	spawnlp
_ttol	atol	_tspawnlpe	spawnlpe
_tchdir	chdir	_tspawnv	spawnv
_tchmod	chmod	_tspawnve	spawnve
_tcreat	creat	_tspawnvp	spawnvp
_tctime	ctime	_tspawnvpe	spawnvpe
_texecl	execl	_stprintf	sprintf
_texecle	execle	_stscanf	sscanf
_texeclp	execlp	_tstat	stat
_texeclpe	execlpe	_tcscat	strcat
_texecv	execv	_tcschr	strchr
_texecve	execve	_tcscmp	strcmp
_texecvp	execvp	_tcscoll	strcoll
_texecvpe	execvpe	_tcscpy	strcpy
_tfdopen	fdopen	_tcscspn	strcspn
_fgettc	fgets	_tcsdec	strdec
_fgettchar	fgetchar	_tcsdup	strdup
_fgettc	fgetc	_tcsftime	strftime
_tfindfirst	findfirst	_tcsicmp	stricmp
_tfindnext	findnext	_tcsinc	strinc
_tfopen	fopen	_tcslen	strlen
_ftprintf	fprintf	_tcslwr	strlwr
_fputtc	fputc	_tcsncmp	strncmp
_fputtchat	fputchar	_tcsnbcnt	strncnt
_fputts	fputs	_tcsncoll	strncoll
_tfreopen	freopen	_tcsncpy	strncpy
_ftscanf	fscanf	_tcsnextc	strnextc
_tfsopen	fsopen	_tcsnicmp	strnicmp
_gettc	getc	_tcsninc	strninc

Макрос из tchar.h	Функция	Макрос из tchar.h	Функция
_gettchar	getchar	_tcspbrk	strpbrk
_tgetcwd	getcwd	_tcsrchr	strrchr
_tgetenv	getenv	_tcsrev	strrev
_getts	gets	_tcsset	strset
_istalnum	isalnum	_tcsspn	strspn
_istalpha	isalpha	_tcsspnp	strspnp
_istascii	isascii	_tcsstr	strstr
_istcntrl	iscntrl	_tcstod	strtod
_istdigit	isdigit	_tcstok	strtok
_istgraph	isgraph	_tcstol	strtol
_istlower	islower	_tcstoul	strtoul
_istprint	isprint	_tcsupr	strupr
_istpunct	ispunct	_tcsxfrm	strxfrm
_istspace	isspace	_tsystem	system
_istupper	isupper	_ttempnam	tempnam
_istxdigit	isxdigit	_ttmpnam	tmpnam
_itot	itoa	_totlower	tolower
_ltot	ltoa	_totupper	toupper
_tmain	main	_ultot	ultoa
_tmkdir	mkdir	_ungettc	ungetc
_tmktemp	mktemp	_tunlink	unlink
_topen	open	_tutime	utime
_tperror	perror	_vftprintf	vfprintf
_tprintf	printf	_vtprintf	vprintf
_puttc	putc	_vstprintf	vsprintf

СОДЕРЖАНИЕ