А.П. Полищук, С.А. Семериков

Событийно-ориентированное программирование

(Конспект вводного курса лекций и методические пособия по программированию на C++ для Windows)

Кривой Рог: КГПУ, 2001

Часть первая. Конспект лекций.

1. Основные концепции.

Любая компьютерная программа выполняется следующим образом: после размещения в памяти глобальных переменных и конструирования глобальных объектов классов начинается выполнение команд, содержащихся в главной подпрограмме. Как правило, вначале осуществляется ряд инициализирующих действий – разбор параметров командной строки, формирование интерфейса пользователя (вывод меню услуг и пр.). После выполнения таких работ возможны два принципиально различных метода работы программы – либо она выполняется далее по жесткому пути последовательно оператор за оператором (команда за командой), вызывая в заданной последовательности необходимые подпрограммы, вплоть до завершения, либо после выполнения инициализирующих действий переходит в режим ожидания событий, требующих выполнения определенных действий, и после наступления события вызывает на выполнение подпрограмму – обработчик этого события, продолжая одновременно анализировать поступление других событий. и т. д. в бесконечном цикле, пока не поступит событие, требующее покинуть цикл и завершить программу. При этом момент наступления того или иного события и их последовательность не регламентированы, но события могут быть ранжированы по приоритетам их обслуживания и более приоритетное событие может прервать обслуживание менее приоритетного на время своего обслуживания. Менее приоритетное по отношению к обслуживаемому в текущий момент событие должно стать в очередь на обслуживание и его обработчик выполнится, когда подойдет его очередь. Методы составления такого типа программ мы и называем **событийно – ориентированным программированием**.

На физическом уровне событие в компьютере - это всегда электрический сигнал на входе процессора, прерывающий его работу; такие сигналы могут поступать от подключенных к компьютеру устройств и каждому из них присваивается уникальный номер, по которому процессор формирует адрес команды, с которой может начинаться обработка соответствующего запроса. Чтобы освободить программиста от необходимости непосредственно анализировать сигналы устройств, создаются программные комплексы операционных систем, размещающих по согласованным с действиями процессора адресам свои подпрограммы реакции на поступающие от устройств сигналы. Каждая прикладная программа может реагировать на одно и то же событие (например, нажатие комбинации клавиш или кнопок мыши) по разному, кроме того, она должна иметь возможность сама имитировать события для выполнения соответствующих действий (для вызова нужного обработчика события), поэтому одна из основных задач операционной системы состоит в **преобразовании поступившего сигнала в информационное сообщение** (это как правило структура с необходимым набором полей), **определение адресата – обработчика события и вызов этого обработчика с передачей ему соответствующего событию сообщения в качестве одного из аргументов.**

Чтобы операционная система могла выполнить вызов необходимого обработчика, структура прикладной программы должна соответствовать требованиям той операционной системы, под управлением которой она будет выполняться.

2. Требования OS Microsoft Windows к структуре C++ - прикладной программы (Windows – приложения).

2.1.Интерфейс с Microsoft Windows API (Application Programming Interface) обеспечивается через заголовочные файлы

#include <windows.h>
#include <windowsx.h>

Эти файлы непосредственно или через другие включаемые файлы, например windef.h, содержат множество определений типов данных, макросов, прототипов функций, констант и т. д.

Перед включением файла windows.h для выполнения более строгой проверки типов и страховки от ошибок рекомендуется определить символ STRICT:

#define STRICT

Для разработки приложений Windows используется большое количество типов данных и констант, определенных в заголовочных файлах. Эти типы данных как бы замещают собой стандартные типы данных языка Си. Такая замена позволяет отделить программный интерфейс Windows от самой операционной системы Windows и от конкретных реализаций компиляторов языков Си, C++ с другой.

Например, в 16-разрядной среде тип LPSTR отображается на дальний указатель, состоящий из селектора и смещения. В 32-разрядной среде при использовании сплошной (FLAT) модели памяти содержимое сегментных регистров устанавливается один раз при запуске приложения и в дальнейшем не изменяется (самим приложением). Для адресации в этом случае используется только 32-разрядная компонента смещения и смысл понятия «дальний указатель» нивелируется.

Если ваша программа использует тип LPSTR, при ее переносе в среду Windows NT вам не придется изменять исходные тексты, достаточно выполнить новую трансляцию. Для этой операционной системы ключевое слово FAR определено как пустое место:

#define FAR

Поэтому все дальние указатели, определенные с использованием этого ключевого слова, автоматически превратятся в ближние (что и требовалось). Если бы вы определяли дальний указатель на строку символов как char _far *, вам бы пришлось удалять или переопределять ключевое слово _far.

Для создания мобильных приложений следует пользоваться не стандартными типами данных, реализованными в конкретной версии системы, а теми типами данных, которые определены через файлы windows.h, windowsx.h -

---

BOOL, BYTE, WORD, DWORD, UINT
PSTR, NPSTR, LPSTR, LPCSTR, PBYTE, LPBYTE, PINT, LPINT, PWORD, LPWORD
PLONG, LPLONG, PDWORD, LPDWORD, LPVOID

Файл windows.h содержит определения для многочисленных структур данных, таких, как POINT, RECT, TEXTMETRICS и т. п. Для всех структур данных определены указатели, например:

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

Еще один важный тип данных, определенный в файле windows.h, - это различные идентификаторы (handle). Для использования того или иного объекта или ресурса Windows вы должны получить его идентификатор. Для получения идентификатора вызывается одна из функций программного интерфейса Windows. Например, для получения идентификатора контекста отображения можно воспользоваться функцией GetDC:

```
HDC hdc;
hdc = GetDC(hwnd);
```

Идентификатор должен использоваться только для ссылки на ресурс, например:

```
DrawText(hdc, (LPSTR)szBuf, nBufSize, &rc,
  DT_CENTER | DT_VCENTER | DT_NOCLIP | DT_SINGLELINE);
```

Некоторые ресурсы являются ограниченными, поэтому после того, как вы их использовали, эти ресурсы следует отдать Windows с помощью специально предназначенных для этого функций. Например, для того чтобы отдать идентификатор контекста отображения, полученного через GetDC ,следует вызвать функцию ReleaseDC:

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

Приведем список некоторых типов идентификаторов ресурсов:

Тип идентификатора	Описание
HGLOBAL	Идентификатор блока глобальной памяти
HACCEL	Акселератор
HBITMAP	Изображение в виде битового образа (bitmap)
HBRUSH	Кисть
HCURSOR	Курсор
HDC	Контекст устройства
HDRVR	Драйвер устройства
HFONT	Шрифт
HGDIOBJ	Объект графического интерфейса GDI
HICON	Пиктограмма
HLOCAL	Идентификатор блока локальной памяти
HMENU	Меню
HMETAFILE	Метафайл
HPALETTE	Палитра
HPEN	Перо
HRGN	Область
HRSRC	Ресурс
HSTR	Строка символов

HTASK	Задача
HWND	Окно

Файл windows.h содержит определение большого количества имен символических констант, таких, как коды сообщений, режимы работы и возможные варианты параметров для различных функций программного интерфейса Windows. Символические имена большинства констант, определенные в файле windows.h и других файлах, включаемых в исходные тексты приложений Windows, начинаются с префиксов, таких, как WM_ или WS_. Имена некоторых констант не имеют таких префиксов, например OPAQUE, TRANSPARENT и т. п.

Список некоторых префиксов имен констант.

Префикс символического имени константы	Описание
BI	Компрессия изображений bitmap
BN	Коды сообщений от кнопок
BS	Стили логических кистей
CB	Стили органа управления Combo Box
CBN	Коды сообщений от органа управления Combo Box
CBS	Стили при создании органа управления Combo Box
CE	Коды ошибок при передаче данных
CF	Форматы универсального буфера обмена данными Clipboard
COLOR	Системные цвета отдельных элементов пользовательского интерфейса Windows
CS	Стили класса окна
DRIVE	Тип диска
DS	Стили при создании диалоговых панелей
DT	Режимы форматирования при выводе текста функцией DrawText
HT	Коды, возвращаемые функцией DefWindowProc при обработке сообщения WM_NCHITTEST
ID	Идентификаторы команд диалоговой панели
IDC	Идентификаторы встроенных курсоров
IDI	Идентификаторы встроенных пиктограмм
MB	Флаги для функции MessageBox
META	Коды для метафайлов
MM	Режимы отображения
CC	Способность устройства рисовать различные кривые
OF	Константы для работы с файлами
PS	Стили пера
RT	Типы ресурсов

SB	Команды и константы для полосы просмотра (Scroll Bar)
SBS	Стили полосы просмотра
SC	Значения системных команд, передаваемых вместе с сообщением WM_SYSCOMMAND
SIZE	Константы для описания способа изменения размера окна
SM	Константы для определения системных метрик
SW	Константы для функции ShowWindow
TA	Константы для выбора способа выравнивания текста при его выводе с помощью функций TextOut и ExtTextOut.
VK	Коды виртуальных клавиш
WM	Сообщения Windows
WS	Стили окна
WS_EX	Расширенные стили окна

Изобилие типов в приложениях Windows создает определенные трудности для программиста. Ему приходится постоянно думать о соответствии имен переменных типам переменных. Для облегчения работы программистов рекомендуется для всех имен параметров функций и других переменных использовать префиксы. Эти префиксы должны быть заданы маленькими буквами. Список некоторых префиксов, рекомендуемых для различных типов данных:

Префикс	Тип данных
b	BYTE
lpb	BYTE FAR*
lpch	char FAR*
Dlgprc	DLGPROC
dw	DWORD
lpdw	DWORD FAR*
haccl	HACCEL
hbm	HBITMAP
hbr	HBRUSH
hcur	HCURSOR
hdc	HDC
hdrvr	HDRVR
hdwp	HDWP
hf	HFILE
hfont	HFONT
Hgdiobj	HGDIOBJ
hglb	HGLOBAL

Hhook	HHOOK
hicon	HICON
hinst	HINSTANCE
hloc	HLOCAL
hmenu	HMENU
hmf	HMETAFILE
hmod	HMODULE
hkprc	HOOKPROC
hpal	HPALETTE
hpen	HPEN
hrgn	HRGN
hrsrc	HRSRC
hstr	HSTR
htask	HTASK
hwnd	HWND
n	Int
l	LONG
lParam	LPARAM
lpb	LPBYTE
lpsz	LPCSTR
lpn	LPINT
lpl	LPLONG
lpsz	LPSTR
lpv	LPVOID
lpw	LPWORD
lResult	LRESULT
npsz	NPSTR
npb	PBYTE
lppt	POINT FAR*
lprc	RECT FAR*
tmprc	TIMERPROC
u	UINT
wndenmprc	WNDENUMPROC
wndprc	WNDPROC
u или w	WORD
wParam	WPARAM

2.2.Главная подпрограмма (C или C++) должна иметь следующий формат:

```
int APIENTRY                    //    (синонимы - WINAPI, CALLBACK)
WinMain(HINSTANCE hInstance,  HINSTANCE hPrevInstance,
            LPSTR     lpszCmdLine,    int       nCmdShow)
{// Тело функции
}
```

В 16-разрядных версиях Windows 3.x вместо APIENTRY использовался тип PASCAL, определенный в файле windows.h следующим образом:
#define PASCAL _stdcall. Можно и в Win32 использовать и PASCAL, хотя и не рекомендуется.
Функция, описанная с ключевым словом _stdcall, использует соглашение языка Паскаль при передаче параметров – справа налево, такая функция должна сама освобождать стек перед возвратом.
Функция WinMain должна возвращать int (и ничего другого) и иметь ровно четыре параметра.
Первые два параметра имеют тип HINSTANCE (идентификатор приложения).
Ваша программа перед запуском получает свой уникальный идентификатор, который передается ей через параметр **hInstance**. Идентификатор приложения используется при вызове многих функций программного интерфейса Windows, поэтому желательно сохранить его в памяти на все время работы приложения в глобальной области данных, чтобы сделать доступным всем подпрограммам .
У вас существует возможность запустить одно приложение несколько раз. Каждая копия приложения в этом случае также будет иметь свой собственный идентификатор.
Приложение Win16 может легко определить идентификаторы всех своих одновременно работающих копий. Для этого предназначен второй параметр функции WinMain - параметр hPrevInstance. Если запущена только одна копия приложения, этот параметр равен нулю. В противном случае в Win16 параметр hPrevInstance равен идентификатору предыдущей копии данного приложения. Анализируя параметр hPrevInstance, приложение Win16 может выполнять различные действия в зависимости от того, была ли уже загружена на момент запуска другая копия приложения. Вы можете полностью блокировать возможность параллельной работы нескольких копий одного приложения Win16, если в начале функции WinMain расположите следующую строку:

```
if(hPrevInstance) return 0;
```

Приложения Win32 работают в разных адресных пространствах и им недоступны идентификаторы ранее запущенных копий, второй параметр hPrevInstance всегда NULL и использовать его невозможно – в Windows9x, Windows NT это по существу архаизм для совместимости с ранее составленными программами под Win16. Тем не менее вы можете обнаружить ранее запущенную копию вашей программы, вызвав предназначенную для этого функцию поиска с именем класса окна в качестве аргумента:

```
HWND hWnd = FindWindow(szClassName, NULL);
 if(hWnd)                                  //Если копия найдена
{ if(IsIconic(hWnd))                       //Если окно свернуто
```

```
  ShowWindow(hWnd, SW_RESTORE); //Восстановим его
  SetForegroundWindow(hWnd); }          //И выдвинем на передний план
```

Третий параметр функции WinMain имеет имя lpszCmdLine. Он имеет тип LPSTR, который определяется в include-файле windows.h следующим образом:

```
typedef char FAR* LPSTR;
```

Из этого определения видно, что параметр lpszCmdLine является указателем на символьную строку – это «хвост» командной строки Вашей программы – аргументы, которые Вы передаете через командную строку при запуске программы. После запуска приложение может проанализировать строку параметров. Перед передачей параметров приложению никакой обработки строки параметров не производится, приложение через стек получает строку параметров точно в таком виде, в котором она была указана при запуске. Строка параметров будет закрыта двоичным нулем.

Последний параметр функции WinMain имеет имя nCmdShow и тип int. Этот параметр содержит рекомендации приложению относительно того, как оно должно нарисовать свое главное окно (в развернутом или свернутом виде, видимом или нет и т.д.) и может использоваться при отображении этого окна.

Таким образом традиционная простейшая C++ - программа для Windows, не обрабатывающая никаких сообщений и не создающая своего главного окна, может выглядеть например так:

```
#define STRICT  //Строгая проверка типов
#include <windows.h>
#pragma argsused //Отключение предупреждений о неиспользуемых аргументах
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpszCmdLine, int nCmdShow)
{//Вызов функции вывода вспомогательного окна с сообщением
 MessageBox(NULL, "Hello, world!", "Main Window", MB_OK); return 0;
}
```

2.3.Главное окно приложения.

Ориентированная на персональные компьютеры **OS Microsoft Windows** имеет графический многооконный интерфейс пользователя и целесообразность ее использования без окон трудно себе представить. Обычная структура оконного интерфейса **Windows** предполагает наличие главного окна приложения (обычного или диалогового) и дочерних окон в виде панелей диалога, меню, панелей инструментов, отдельных кнопок и т.п. Большинство сообщений **Windows** посылаются оконным обработчикам, поэтому создание главного окна приложения является по сути обязательным аттрибутом Windows - приложения. Для создания окна в Windows предусмотрена структура WNDCLASS и ее расширенный вариант WNDCLASSEX, определенный так:

```
typedef struct _WNDCLASSEX {
   UINT    cbSize;                 //Размер структуры в байтах
   UINT    style;                  // Стиль окна
   WNDPROC lpfnWndProc;     //Указатель на оконную процедуру обраб сообщений
   int     cbClsExtra;             // Размер дополнительной области данных,
                                   // зарезервированной в описании класса окна
   int     cbWndExtra;             //Размер дополнительной области данных,
```

```
                                    // зарезервированной для каждого окна,
                                    // созданного на основе данного класса
   HANDLE  hInstance;               // Идентификатор приложения
   HICON   hIcon;                   // Идентификатор пиктограммы для окна
   HCURSOR hCursor;                 // Идентификатор курсора
   HBRUSH  hbrBackground;           // Цвет фона окна
   LPCTSTR lpszMenuName;            // Идентификатор меню
   LPCTSTR lpszClassName;           //Имя класса окна
   HICON   hIconSm;                 // Малогабаритный идентификатор пиктограммы
} WNDCLASSEX;
```

Для создания окна нам необходимо разместить в памяти переменную этого типа, присвоить значения всем ее полям, и зарегистрировать в ОС например так:

```
char const szClassName[] = "WindowAppClass";       // Имя класса окна
char const szWindowTitle[] = "Window Application"; // Заголовок окна
ATOM aWndClass; // для результата регистрации класса окна
WNDCLASS wc;    // структура для регистрации  класса окна
// Прототип функции окна для обработки сообщений
LRESULT CALLBACK  WndProc(HWND, UINT, WPARAM, LPARAM);
//Главная функция
int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpszCmdLine, int nCmdShow)
{
HWND hWnd; // идентификатор главного окна приложения
MSG msg;       // структура для работы с сообщениями
// Проверяем, не запускалось ли это приложение ранее
hWnd = FindWindow(WindowAppClass, NULL);
  if(hWnd)
  {
// Если окно приложения было свернуто в пиктограмму, восстанавливаем его
    if(IsIconic(hWnd))  ShowWindow(hWnd, SW_RESTORE);
// Выдвигаем окно приложения на передний план
   SetForegroundWindow(hWnd);
    return FALSE;
  }
//Перечисленные ниже операторы инициализации оконного объекта могут быть вы-
не//сены в подпрограмму начальной инициализации, вызываемую здесь
wc.cbSize=sizeof(WNDCLASSEX);
// Поле wc.hIconSm задает идентификатор маленькой  пиктограммы, которая будет
//отображаться в левой  части заголовка окна (в области системного меню).  Загру-
жаем //пиктограмму из ресурсов приложения при  помощи функции LoadImage, так
как //функция  LoadIcon может загрузить только обычную пиктограмму
 wc.hIconSm = LoadImage(hInst,  MAKEINTRESOURCE(IDI_APPICON_SM),
IMAGE_ICON, 16, 16, 0);
 // Завершаем заполнение структуры WNDCLASSEX
wc.style = 0;                                    //Стиль класса окна
```

```
wc.lpfnWndProc = (WNDPROC) WndProc;    //Адрес оконной процедуры
wc.cbClsExtra = 0;                     //Дополнительные данные класса окна
wc.cbWndExtra = 0;                     //Дополнительные данные окна
wc.hInstance = hInstance;              //Идентификатор программы
wc.hIcon = LoadIcon(NULL, IDI_APPLICATION); //Иконка для окон этого класса
wc.hCursor = LoadCursor(NULL, IDC_ARROW);   //Курсор мыши
wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1); //Цвет фона (кисти)
wc.lpszMenuName = (LPSTR)NULL;         //Имя меню – если есть
wc.lpszClassName = (LPSTR)szClassName; //Имя класса окна
aWndClass = RegisterClass(&wc);        // Регистрация класса окна
if(aWndClass==0) return FALSE;         //Если регистрация не прошла
 // После успешной регистрации  создаем  главное окно приложения
 hWnd = CreateWindow( szClassName, // имя класса окна
                      szWindowTitle, // заголовок окна
                       WS_OVERLAPPEDWINDOW, // стиль окна
                      CW_USEDEFAULT, // задаем размеры и расположение
                      CW_USEDEFAULT, // окна, принятые по умолчанию
                      CW_USEDEFAULT,
                      CW_USEDEFAULT,
                      0, // идентификатор родительского окна
                      0, // идентификатор меню
                      hInstance, // идентификатор приложения
                      NULL);   // указатель на дополнительные  параметры
// Если создать окно не удалось, завершаем приложение
if(!hwnd) return FALSE;
// Рисуем окно. Для этого после функции ShowWindow,  рисующей окно, вызываем
//функцию UpdateWindows,  посылающую сообщение WM_PAINT в функцию окна
//для его перерисовки:
ShowWindow(hwnd, nCmdShow);
UpdateWindow(hwnd);
// Запускаем цикл обработки сообщений
while(GetMessage(&msg, 0, 0, 0)) //Взять сообщение из очереди
{ DispatchMessage(&msg); }        //Направить сообщение адресату
//Завершаем работу приложения
return msg.wParam;}
```

2.4.Окна Windows как объекты

Все окна, формируемые приложением или операционной системой Windows для приложения, можно рассматривать как объекты, над которыми можно выполнять различные операции. В операционной системе Windows объектами, с которыми можно что-либо делать, являются окна . Для каждого окна приложение должно создать свои данные и свой набор методов, то есть функций, реагирующих на те или иные действия, которые выполняет над окнами оператор или операционная система. Например, вы можете щелкнуть кнопкой мыши в то время, когда курсор мыши находится над окном. Это событие, на которое окно может реагировать, а может и

не реагировать. Вы можете щелкнуть мышью по любому окну, принадлежащему приложению, и каждое окно должно реагировать на это по-своему.
В Windows существует механизм, позволяющий задать для каждого окна данные и набор методов обработки событий, имеющих отношение к любому окну, созданному приложением. Этот механизм основан на использовании так называемой функции окна (window function) и сообщений.

2.5.Функция окна

Функция окна - это подпрограмма , которая определяется для одного окна или группы окон. Каждый раз, когда происходит какое-либо событие, имеющее отношение к окну (например, щелчок мышью в окне), операционная система Windows вызывает соответствующую функцию окна и передает ей параметры, описывающие событие. Функция окна анализирует эти параметры и выполняет соответствующие действия. При возникновении события операционная система Windows формирует сообщение, описывающее событие, и затем направляет его в нужную функцию окна. В качестве параметров, передаваемых функции окна, используются отдельные компоненты сообщения, соответствующего событию (или, иными словами, созданному в результате появления события). Поэтому основная задача функции окна - обработка сообщений, распределяемых окну операционной системой Windows.
Можно считать, что единственная для каждого окна функция окна реализует все методы окна как объекта. В объектно-ориентированных языках программирования используется такое понятие, как наследование. Объекты классов могут наследовать методы других классов. В операционной системе Windows также предусмотрен механизм наследования методов. Он реализуется с использованием так называемых классов окна.
Для каждого класса окна определяется функция окна. При создании окна необходимо указать, к какому классу оно будет принадлежать и, соответственно, какую функцию окна будет использовать для обработки сообщений. Приложения могут создавать собственные классы окна, определяя свои функции окна (и следовательно, свои методы), либо использовать стандартные, определенные в Windows классы окна.
Пусть, например, нам надо создать окно- кнопку. Мы можем создать собственный класс кнопочного окна и для него определить собственную функцию окна. Эта функция будет обрабатывать сообщения и при необходимости изображать в окне нажатую или отжатую кнопку и выполнять другие действия. Однако в Windows уже определен класс окна, соответствующий кнопкам. Если вы воспользуетесь этим классом, вам не придется создавать свою функцию окна, так как будет использоваться функция, уже имеющаяся в Windows и выполняющая все необходимые действия.
Любое создаваемое вами окно может наследовать свойства уже созданных ранее окон, добавляя свои или переопределяя уже имеющиеся в базовом классе методы. В этом и заключается механизм наследования Windows.
Использование механизма наследования значительно упрощает процесс создания приложений, так как для большинства стандартных органов управления, таких, как кнопки, меню, полосы просмотра, строки редактирования, панели диалога и т. п., в

операционной системе Windows уже определены классы окон и все необходимые методы.

2.6.Стили окон.

Обычно ни одно приложение не ограничивается созданием главного окна приложения. Как правило, внутри главного окна создаются другие окна.
Напомним, что для создания окна вам необходимо зарегистрировать класс окна.
Есть классы окон, зарегистрированные при инициализации Windows. Ваше приложение может создавать окна либо на базе собственных классов (созданных и зарегистрированных приложением), либо на базе готовых классов, созданных и зарегистрированных самой операционной системой Windows.
На базе одного класса окна приложение может создать несколько окон. Все эти окна могут быть сделаны в одном или нескольких стилях. Стиль окна определяет внешний вид окна и его поведение. Для класса окна также определяется понятие стиля - стиль класса определяет внешний вид и поведение всех окон, созданных на базе данного класса.

Стиль класса окна

Стиль класса окна определяется при регистрации класса окна.
Стиль класса окна задается в виде отдельных битов, для которых в файле windows.h определены символические константы с префиксом CS_:

Стиль	Описание
CS_BYTEALIGNCLIENT	Внутренняя область окна (client area) должна быть выравнена по границе байта видеопамяти. Иногда используется для ускорения процесса вывода изображения
CS_BYTEALIGNWINDOW	Все окно (не только внутренняя область окна) должно быть выравнено по границе байта видеопамяти
CS_CLASSDC	Необходимо создать единый контекст отображения, который будет использоваться всеми окнами, создаваемыми на базе данного класса
CS_DBLCLKS	Функция окна будет получать сообщения при двойном щелчке клавишей мыши (double click)
CS_GLOBALCLASS	Данный класс является глобальным и доступным другим приложениям. Другие приложения могут создавать окна на базе этого класса
CS_HREDRAW	Внутренняя область окна должна быть перерисована при изменении ширины окна
CS_NOCLOSE	В системном меню окна необходимо запретить выбор функции закрытия окна (строка Close будет отображаться серым цветом, и ее нельзя выбрать)
CS_OWNDC	Для каждого окна, определяемого на базе данного класса, будет создаваться отдельный контекст отображения

CS_PARENTDC	Окно будет пользоваться родительским контекстом отображения, а не своим собственным. Родительский контекст - это контекст окна, создавшего другое окно (см. дальше)
CS_SAVEBITS	Для данного окна Windows должна сохранять изображение в виде битового образа (bitmap). Если такое окно будет перекрыто другим окном, то после уничтожения перекрывшего окна изображение первого окна будет восстановлено Windows на основании сохраненного ранее образа
CS_VREDRAW	Внутренняя область окна должна быть перерисована при изменении высоты окна

Чаще всего используются стили CS_HREDRAW и CS_VREDRAW:

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

Если для класса заданы стили CS_HREDRAW и CS_VREDRAW, при изменении размеров окна функция окна может получить сообщение WM_PAINT. В этом случае функция окна должна перерисовать часть окна или все окно. Разумеется, если вы просто уменьшили размер окна, перерисовывать ничего не надо, и функция окна в этом случае не получит сообщения WM_PAINT.

Стиль CS_DBLCLKS используется при необходимости отслеживать двойные щелчки мышью. При этом в функцию окна посылаются сообщения WM_LBUTTONDBLCLK и WM_RBUTTONDBLCLK. Если этот стиль не будет задан, как бы быстро вы ни щелкали мышью, функция окна получит только идущие парами сообщения о том, что вы нажимаете и отпускаете левую или правую клавишу мыши.

Стиль окна

Определенный в классе окна стиль класса окна используется при создании на базе этого класса всех окон. Для дальнейшего уточнения внешнего вида и поведения окна используется другая характеристика – стиль конкретного окна. Стиль окна указывается при создании окна функцией CreateWindow.

Для определения стиля окна используются символические константы с префиксом WS_, определенные в файле windows.h. С помощью этих констант можно определить примерно два десятка стилей окна, однако чаще всего используются несколько основных стилей.

Мы рассмотрим три основных стиля окон - перекрывающиеся окна (overlapped window), временные окна (pop-up window) и дочерние окна (child window).

Перекрывающиеся (overlapped) окна

Перекрывающиеся окна обычно используются в качестве главного окна приложения. Такие окна имеют заголовок (title bar), рамку и, разумеется, внутреннюю часть окна (client region). Дополнительно перекрывающиеся окна могут иметь (а могут и не иметь) системное меню, кнопки для максимального увеличения размера окна и для сворачивания окна в пиктограмму, вертикальную и горизонтальную полосу просмотра (scroll bar) и меню.

Перекрывающиеся окна называются также окнами верхнего уровня (top-level window).
Константа WS_OVERLAPPED определяет базовый стиль окна - перекрывающееся окно. Стиль WS_OVERLAPPEDWINDOW в добавление к базовому указывает, что окно должно иметь заголовок (константа WS_CAPTION), системное меню (WS_SYSMENU), толстую рамку для изменения размера окна (WS_THICKFRAME), кнопку минимизации размера окна (WS_MINIMIZEBOX) и кнопку для максимального увеличения размера окна (WS_MAXIMIZEBOX). Если окно имеет заголовок, вы можете его перемещать по экрану.
Приложение Windows может создавать несколько окон, связанных между собой "узами родства" и "отношениями собственности". В частности, при создании перекрывающегося окна при помощи функции CreateWindow в качестве восьмого параметра функции вы можете указать так называемый идентификатор окна-владельца. Окно-владелец уже должно существовать на момент создания второго окна, имеющего владельца.
Таким образом, если вы создаете несколько перекрывающихся окон, одни окна могут принадлежать другим.
Если окно-хозяин сворачивается в пиктограмму, все окна, которыми оно владеет, становятся невидимыми. Если вы сначала свернули в пиктограмму окно, которым владеет другое окно, а затем и окно-хозяин, пиктограмма первого (подчиненного) окна исчезает.Если вы уничтожили окно, автоматически уничтожаются и все принадлежащие ему окна. Обычное перекрывающееся окно, не имеющее окна-владельца, может располагаться в любом месте экрана и принимать любые размеры. Подчиненные окна располагаются всегда над поверхностью окна-владельца, загораживая его.
Координаты создаваемых функцией CreateWindow перекрывающихся окон указываются по отношению ко всему экрану. Таким образом, если вы создаете перекрывающееся окно с координатами (0, 0), оно будет расположено в верхнем левом углу экрана.

Временные (pop-up) окна

Другим базовым стилем является стиль временных окон, которые обычно используются для вывода информационных сообщений и остаются на экране непродолжительное время.Временные окна имеют стиль WS_POPUP.
Временные окна, в отличие от перекрывающихся, могут не иметь заголовок (title bar). Если для временного окна определен заголовок, оно может иметь и системное меню. Часто для создания временных окон, имеющих рамку, используется стиль WS_POPUPWINDOW .Если надо добавить к временному окну системное меню и заголовок, стиль WS_POPUPWINDOW следует использовать в комбинации со стилем WS_CAPTION, добавляющим заголовок. Временные окна могут иметь окно владельца и могут сами владеть другими окнами. Все замечания, сделанные относительно владения перекрывающимися окнами, справедливы и для временных окон. Начало системы координат, используемой при создании временных окон, находится в левом верхнем углу экрана. Поэтому при создании временных окон используются экранные координаты (так же, как и при создании перекрывающихся окон).

При изменении размеров временного окна (так же, как и дочернего) функция окна получает сообщение WM_PAINT, в параметрах которого указаны новые размеры окна.
В общем случае вы можете рассматривать временные окна как специальный вид перекрывающихся окон.

Дочерние окна

Дочерние окна чаще всего используются приложениями Windows. Эти окна нужны для создания органов управления, например таких, как кнопки или переключатели. Windows имеет множество классов, на базе которых созданы стандартные органы управления - кнопки, полосы просмотра и т. п. Все эти органы управления представляют собой дочерние окна. Базовый стиль дочерних окон определяется при помощи константы WS_CHILD.
По аналогии с другими базовыми стилями в файле windows.h определена константа WS_CHILDWINDOW, которая полностью эквивалентна константе WS_CHLD.
В отличие от перекрывающихся и временных окон дочерние окна, как правило, не имеют рамки, заголовка, кнопок минимизации и максимального увеличения размера окна, а также полос просмотра. Дочерние окна сами рисуют все, что в них должно быть изображено.
Особенности дочерних окон.
Дочерние окна должны иметь окно-родителя .
Дочерние окна всегда располагаются на поверхности окна-родителя. При создании дочернего окна начало системы координат расположено в левом верхнем углу внутренней поверхности окна-родителя (но не в верхнем углу экрана, как это было для перекрывающихся и временных окон). Так как дочерние окна перекрывают окно-родителя, если вы сделаете щелчок мышью над поверхностью дочернего окна, сообщение от мыши попадет в функцию дочернего, но не родительского окна.
При создании дочернего окна в качестве девятого параметра (вместо идентификатора меню, которого не может быть у дочернего окна) функции CreateWindow необходимо указать созданный вами идентификатор дочернего окна. Таким образом, если для какого-либо окна приложения вы создаете несколько дочерних окон, необходимо для каждого окна указать свой идентификатор (типа int). Этот идентификатор будет использован дочерним окном при отправлении сообщений родительскому окну, поэтому вы должны при создании разных дочерних окон использовать разные идентификаторы, хотя это и не обязательно.
Дочернее окно как бы "прилипает" к поверхности родительского окна и перемещается вместе с ним. Оно никогда не может выйти за пределы родительского окна. Все дочерние окна скрываются при сворачивании окна-родителя в пиктограмму и появляются вновь при восстановлении родительского окна.
При изменении размеров родительского окна дочерние окна, на которых отразилось такое изменение (которые вышли за границу окна и появились вновь), получают сообщение WM_PAINT. При изменении размеров родительского окна дочерние окна не получают сообщение WM_SIZE. Это сообщение попадает только в родительское окно.

Список стилей окна

Приведем полный список возможных стилей окна, определенных в виде символических констант в файле windows.h.

Имя константы	Описание стиля
WS_BORDER	Окно с рамкой
WS_CAPTION	Окно с заголовком. Этот стиль несовместим со стилем WS_DLGFRAME. При использовании стиля WS_CAPTION подразумевается использование стиля WS_BORDER
WS_CHILD	Дочернее окно. Несовместим со стилем WS_POPUP
WS_CHILDWINDOW	То же самое, что и WS_CHILD
WS_CLIPCHILDREN	Этот стиль используется при создании родительского окна. При его использовании родительское окно не перерисовывает свои внутренние области, занятые дочерними окнами
WS_CLIPSIBLINGS	При указании этого стиля дочерние окна не перерисовывают свои области, перекрытые "братьями", то есть другими дочерними окнами, имеющими тех же родителей
WS_DISABLED	Вновь созданное окно сразу становится заблокированным (не получает сообщения от мыши и клавиатуры)
WS_DLGFRAME	Окно с двойной рамкой без заголовка. Несовместим со стилем WS_CAPTION
WS_GROUP	Определяет первый орган управления в группе органов управления. Используется только в диалоговых панелях
WS_HSCROLL	В окне создается горизонтальная полоса просмотра
WS_ICONIC	То же самое, что и WS_MINIMIZE
WS_MAXIMIZE	Создается окно максимально возможного размера
WS_MAXIMIZEBOX	Окно содержит кнопку для увеличения его размера до максимально возможного. Этот стиль необходимо использовать вместе со стилями WS_OVERLAPPED или WS_CAPTION, в противном случае указанная кнопка не появится
WS_MINIMIZE	Создается окно, уменьшенное до предела (свернутое в пиктограмму). Этот стиль необходимо использовать вместе со стилем WS_OVERLAPPED
WS_MINIMIZEBOX	Окно содержит кнопку для сворачивания окна в пиктограмму (минимизации размеров окна). Этот стиль необходимо использовать вместе со стилем WS_OVERLAPPED или WS_CAPTION, в противном случае указанная кнопка не появится

WS_OVERLAPPED	Создается перекрывающееся окно, имеющее заголовок и рамку
WS_OVERLAPPEDWINDOW	Создается перекрывающееся окно, имеющее заголовок, рамку для изменения размера окна, системное меню, кнопки для изменения размеров окна. Этот стиль является комбинацией стилей WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX и WS_MAXIMIZEBOX
WS_POPUP	Создается временное (pop-up) окно
WS_POPUPWINDOW	Комбинация стилей WS_POPUP, WS_BORDER и WS_SYSMENU. Для того чтобы сделать системное меню доступным, необходимо дополнительно использовать стиль WS_CAPTION
WS_SYSMENU	Окно должно иметь системное меню
WS_TABSTOP	Этот стиль указывает орган управления, на который можно переключиться при помощи клавиши <Tab>. Данный стиль может быть использован только дочерними окнами в диалоговых панелях
WS_THICKFRAME	Окно должно иметь толстую рамку для изменения размера окна
WS_VISIBLE	Создается окно, которое сразу становится видимым. По умолчанию окна создаются невидимыми, и для их отображения требуется вызывать функцию ShowWindow
WS_VSCROLL	В окне создается вертикальная полоса просмотра
WS_TILED	Устаревший стиль, аналогичен WS_OVERLAPPED
WS_SIZEBOX	Устаревший стиль, аналогичен WS_THICKFRAME
WS_TILEDWINDOW	Устаревший стиль, аналогичен WS_OVERLAPPEDWINDOW
MDIS_ALLCHILDSTYLES	Этот стиль используется при создании дочерних MDI-окон и определяет окна, которые могут иметь любые комбинации стилей. По умолчанию дочерние MDI-окна имеют стили WS_MINIMIZE, WS_MAXIMIZE, WS_VSCROLL, WS_HSCROLL

3.События и сообщения о событиях.

3.1.Очередь сообщений.

Большинство сообщений создают драйверы устройств ввода/вывода, таких, как клавиатура, мышь, таймер при возникновении событий (физических фактов). Драйверы создают сообщения при поступлении аппаратных прерываний. Например, когда вы нажимаете и затем отпускаете клавишу, драйвер обрабатывает прерывания от кла-

виатуры и создает несколько сообщений. Аналогично сообщения создаются при перемещении мыши или в том случае, когда вы нажимаете кнопки мыши. Таким образом драйверы устройств ввода/вывода преобразуют аппаратные прерывания в сообщения (информационные структуры).
Созданные драйверами сообщения попадают в системную очередь сообщений Windows. Системная очередь сообщений одна. Далее из нее сообщения распределяются в очереди сообщений приложений. Для каждого приложения создается своя очередь сообщений. Очередь сообщения приложений может пополняться не только из системной очереди. Любое приложение может послать сообщение любому другому приложению, в том числе и само себе.
Основная работа, которую должно выполнять приложение, заключается в обслуживании собственной очереди сообщений. Обычно приложение в цикле опрашивает свою очередь сообщений. Обнаружив сообщение, приложение с помощью специальной функции DispatchMessage из программного интерфейса Windows распределяет его нужной функции окна, которая и выполняет обработку сообщения.
Функция WinMain может не выполнять никакой работы, имеющей отношение к поведению приложения. Внутри этой функции находятся инициализирующий фрагмент и цикл обработки сообщений (Message Loop). **Вся основная работа выполняется в функции окна**. Приложение Windows после инициализации переходит в состояние циклического опроса собственной очереди сообщений. Как только происходит какое-либо событие, имеющее отношение к приложению, в очереди приложения появляется сообщение и приложение начинает его обрабатывать. После обработки приложение вновь возвращается к опросу собственной очереди сообщений. Иногда функция окна может получать сообщения непосредственно, минуя очередь приложения.
Так как Windows - мультизадачная операционная система, ее разработчики должны были предусмотреть механизм совместного использования несколькими параллельно работающими приложениями таких ресурсов, как мышь и клавиатура. Так как все сообщения, создаваемые драйверами мыши и клавиатуры, попадают в системную очередь сообщений, должен существовать способ распределения этих сообщений между различными приложениями.
В Windows существует понятие фокуса ввода (input focus), помогающее в распределении сообщений. Фокус ввода - это атрибут, который в любой момент времени может относиться только к одному окну. Если окно имеет фокус ввода, все сообщения от клавиатуры распределяются сначала в очередь сообщений приложения, создавшего окно, а затем - функции окна, владеющего фокусом ввода. Нажимая определенные клавиши, вы можете перемещать фокус ввода от одного окна к другому. Например, если вы работаете с диалоговой панелью, содержащей несколько окон, предназначенных для ввода текста, с помощью клавиши <Tab> вы можете переключать фокус с одного такого окна на другое и вводить текст в различные окна диалоговой панели. Существуют также комбинации клавиш AltTab, с помощью которых вы можете переключиться на другое приложение. При этом фокус ввода будет отдан другому окну, принадлежащему другому приложению.
Сообщения от драйвера мыши всегда передаются функции того окна, над которым находится курсор мыши. При необходимости приложение может выполнить операцию захвата (capturing) мыши. В этом случае все сообщения от мыши будут посту-

пать в очередь приложения, захватившего мышь, вне зависимости от положения курсора мыши.
Простейший цикл обработки сообщений состоит из вызовов двух функций - GetMessage и DispatchMessage.
Функция GetMessage предназначена для выборки сообщения из очереди приложения. Сообщение выбирается из очереди и записывается в область данных, принадлежащую приложению. Функция DispatchMessage предназначена для распределения выбранного из очереди сообщения нужной функции окна. Так как приложение обычно создает много окон и эти окна используют различные функции окна, необходимо распределить сообщение именно тому окну, для которого оно предназначено. Windows оказывает приложению существенную помощь в решении этой задачи - приложению достаточно вызвать функцию DispatchMessage.
Вот как выглядит простейший вариант цикла обработки сообщений:

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

Завершение цикла обработки сообщений происходит при выборке из очереди специального сообщения, в ответ на которое функция GetMessage возвращает нулевое значение.

3.2.Что же представляет собой сообщение?

Структура для сообщений определена так:

```
typedef struct tagMSG
{
 HWND   hWnd;          //Идентификатор окна которому предназначено сообщение
 UINT   message;       //Код или номер сообщения
 WPARAM wParam;        //Специфическая для кажного типа информация
 LPARAM lParam;
 DWORD  time;          //Время отправки сообщения
 POINT  pt;            //Позиция курсора в момент отправки
} MSG;
```

Типы сообщений, с которыми оперирует Windows, вы можете просмотреть через систему помощи Windows API Help (Win32 OnLine). Там же для каждого сообщения будет указано, какая информация содержится в wParam, lParam. Мы приведем эти сведения только для некоторых системных сообщений.

Сообщение	wParam	lParam
WM_CHAR	Код символа	Количество нажатий, скэн –код и другие сопутствующие данные

WM_COMMAND	HIWORD(wParam)- код уведомления от органа управления родительскому окну LOWORD(wParam)-handle позициименю, органа управления	LParam- идентификатор органа управления, пославшего команду
WM_CREATE		LParam-указатель на структуру CREATESTRUCT с данными о создаваемом окне
WM_KEYDOWN	Виртуальный код нажатой клавиши	Сопутствующие данные-количество повторов, скэн-код и пр.
WM_LBUTTONDOWN	Код нажатой кнопки мыши – левая, правая, с Ctrl, Shift	LOWORD-x-координата курсора HIWORD –y – координата
WM_MOVE		LOWORD-горизонтальная позиция окна на экране HIWORD-вертикальная позиция
WM_PAINT	Handle контекста устройства для рисования	
WM_TIMER	Идентификатор таймера	Адрес процедуры обработки таймерных сообщений

Объект, связанный с системным сообщением, кодируется префиксом :
ABM_ - Application desktop toolbar, BM - Button control , CB - Combo box control, CDM - Common dialog box, DBT - Device , DL - Drag list box, DM - Default push button control, EM - Edit control , HDM - Header control , LB - List box control, LVM - List view control, PBM - Progress bar, PSM - Property sheet, SB - Status bar window, SBM - Scroll bar control, STM - Static control , TB - Toolbar, TBM – Trackbar, TCM - Tab control, TTM - Tooltip control, TVM - Tree-view control, UDM - Up-down control, WM - General window.

Среди оконных особое место занимает тип сообщения WM_COMMAND, определяющий сообщение как команду от органов управления, расположенных на поверхности окна (дочерних окон), например от меню при выборе его позиции.

3.3.Структура функции окна для обработки сообщений.

Функция окна получает сообщения при создании окна, в процессе работы приложения, а также при разрушении окна. Сообщение с кодом WM_CREATE передается функции окна в момент создания окна. Функция окна при обработке этого сообщения выполняет инициализирующие действия (аналогично конструктору класса в языке C++). Обработчики сообщений, определенные в функции окна, являются методами для работы с окном как с объектом.
При разрушении структуры данных окна (при уничтожении окна) функция окна получает сообщение с кодом WM_DESTROY. Обработчик этого сообщения действует как деструктор. Если ваша функция окна во время обработки сообщения WM_CREATE создала какие-либо структуры данных, эти структуры должны быть разрушены (а заказанная для них память возвращена операционной системе) во время обработки сообщения WM_DESTROY.

В качестве простейшего промера ниже приводится
оконная функция для обработки сообщений, реагирующая на нажатия кнопок мыши и вызываемая операционной системой:

```
LRESULT CALLBACK  WndProc(HWND hWnd, UINT msg, WPARAM wParam,
LPARAM lParam)
{
 switch (msg)
{
case WM_LBUTTONDOWN: { MessageBox(NULL, "Нажата левая клавиша мыши",
"Сообщение", MB_OK | MB_ICONINFORMATION); return 0; }
case WM_RBUTTONDOWN: { MessageBeep(-1); // звуковой сигнал Message-
Box(NULL, "Нажата правая клавиша мыши", "Сообщение", MB_OK |
MB_ICONINFORMATION); return 0; }
case WM_DESTROY: { PostQuitMessage(0); return 0; }
}
return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

Первый параметр является идентификатором окна, для которого предназначено сообщение. Напомним, что адрес функции окна указывается при регистрации класса окна:
Следующие три параметра функции окна соответствуют полям msg, wParam и lParam структуры MSG. В поле msg записывается код сообщения, поля wParam и lParam описывают дополнительную информацию, передаваемую в функцию окна вместе с сообщением. Формат этой информации зависит от кода сообщения.
В нашем приложении функция окна представляет собой переключатель, выполняющий различные действия для сообщений с разными кодами. Сообщения WM_LBUTTONDOWN, WM_RBUTTONDOWN и WM_DESTROY обрабатываются функцией окна, остальные передаются функции DefWindowProc.

При завершении работы приложения функции окна передается сообщение WM_DESTROY, в ответ на которое функция окна помещает в очередь приложения сообщение WM_QUIT, вызывая функцию PostQuitMessage. При выборке сообщения WM_QUIT завершается цикл обработки сообщений и работа приложения.
Все остальные сообщения передаются без изменения функции DefWindowProc для дальнейшей обработки.

3.4.Макрокоманды для разборки сообщений (Message crackers).

Информация в сообщениях Windows часто представлена в «упакованном» виде и приложению в функции окна приходится извлекать из сообщений отдельные параметры. Например, сообщение WM_COMMAND содержит код сообщения в старшем слове wParam, а младшее слово wParam содержит идентификатор органа управления (например пункта меню или кнопки), породившего коианду. lParam содержит идентификатор окна, пославшего сообщения. Для выделения значений из старшего и младшего слов предназначены специальные макросы HIWORD, LOWORD:

```
case WM_COMMAND:
{
nCmd=HIWORD(wParam); wId=LOWORD(wParam); hWnd=(HWND)(UINT)lParam;
….
}
```

В файле windowsx.h определены и альтернативные макрокоманды анализа сообщений

```
#define GET_WM_COMMAND_ID(wp, lp)              LOWORD(wp)
#define GET_WM_COMMAND_HWND(wp, lp)            (HWND)(lp)
#define GET_WM_COMMAND_CMD(wp, lp)             HIWORD(wp)
```

Группа макрокоманд **Message crackers** позволяет улучшить читаемость тела оператора switch в оконной функции. Покажем это на примере.Пусть нам надо обрабатывать сообщения WM_CREATE и WM_LBUTTONDOWN. Составим функции – обработчики этих сообщений:

```
BOOL WndProc_OnCreate(HWND hWnd, LPCREATESTRUCT lpCreateStruct)
{
…
//Этот обработчик должен вернуть TRUE, чтобы окно создалось
return TRUE;
}
#pragma warning(disable: 4098);
void WndProc_OnLButtonDown(HWND hWnd,BOOL fDoubleClick, int x,int y, UINT
keyFlags)
{
…
}
```

В функции окна используем макрокоманду **HANDLE_MSG**, передав ей первым параметром идентификатор окна, вторым – код сообщения, третьим – имя обработчика сообщения:

```
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
HANDLE_MSG(hWnd,WM_LBUTTONDOWN, WndProc_OnLButtonDown);
HANDLE_MSG(hWnd,WM_CREATE, WndProc_ OnCreate);
default: return(DefWindowProc(hWnd,msg,wParam,lParam));
}
```

В таком синтаксисе оконная функция выглядит компактнее и “читабельнее”, чем при непосредственном использовании операторов вида case WM_LBUTTONDOWN :.

Внимание: имена последних 2-х параметров WndProc при использовании HANDLE_MSG **wParam,lParam обязательны и не могут біть другими**!

В файле windowsx.h вы найдете много удобных и полезных макрокоманд – заглядывайте в него при составлении программ.

4.Работа с устройствами на уровне сообщений.

4.1.Обработка клавиатурных сообщений.

При нажатии или отпускании клавиши, клавиатура генерирует аппаратное прерывание. Обработчик клавиатурного прерывания находится внутри драйвера клавиатуры. Драйвер клавиатуры является составной частью операционной системы Windows. Его задача - генерация на основе аппаратных прерываний клавиатурных сообщений и запись этих сообщений во внутреннюю системную очередь сообщений Windows. Когда вы нажимаете на клавишу и затем отпускаете ее, в системную очередь записывается несколько сообщений.

В среде Windows работает одновременно несколько приложений. Каждое приложение может создать несколько окон. Задачей Windows является правильное распределение сообщений в очереди приложений, с одной стороны, и в функции окон, принадлежащих приложениям, с другой.

Для распределения сообщений от клавиатуры используется концепция так называемого фокуса ввода (input focus). Фокус ввода - это атрибут, который может присваиваться окну, созданному приложением или Windows. Если окно имеет фокус ввода, соответствующая функция окна получает все клавиатурные сообщения из системной очереди.

Приложение может передавать фокус ввода от одного окна другому. Когда вы переключаетесь на другое приложение, фокус ввода передается активному окну приложения, на которое было выполнено переключение, или дочернему окну, созданному активным окном.

Функция окна может проследить за получением и потерей фокуса ввода. Когда окно получает фокус ввода, функции окна передается сообщение WM_SETFOCUS. Когда окно теряет фокус ввода, функции окна передается сообщение WM_KILLFOCUS.

Программный интерфейс Windows содержит две функции, позволяющие узнать или изменить окно, владеющее фокусом ввода. Эти функции имеют соответственно имена GetFocus и SetFocus.

Сообщения, генерируемые драйвером клавиатуры, являются сообщениями низкого уровня. Они несут в себе такую информацию, как, например, скан-код нажатой клавиши. Приложения не пользуются такими сообщениями. Операционная система Windows выполняет преобразование клавиатурных сообщений низкого уровня в сообщения, которые содержат коды виртуальных клавиш (virtual-key code).
Код виртуальной клавиши соответствуют не расположению клавиши на клавиатуре, а действию, которое выполняет эта клавиша. Приложение обрабатывает коды виртуальных клавиш, а соответствие виртуальных и физических клавиш обеспечивается драйвером клавиатуры.
Есть и еще более высокий уровень сообщений. Сообщения этого уровня получаются в результате преобразований сообщений с кодами виртуальных клавиш. Примером могут служить сообщения от меню. Вы можете выбрать строку меню с помощью клавиатуры, и при этом приложение не будет анализировать последовательность нажимаемых при выборе из меню клавиш. Приложение получит от Windows сообщение о том, что из такого-то меню выбрана такая-то строка. Приложение может даже не знать, каким образом сделан выбор - с помощью клавиатуры или мыши. Оно получает информацию лишь о том, что сделан выбор. Таким образом, Windows облегчает работу программиста при создании меню, избавляя его от необходимости следить за действиями оператора при выборе строки.
Сразу отметим, что приложения Windows редко используют методику посимвольного ввода и отображения, широко распространенную при создании программ MS-DOS. Практически посимвольный ввод и отображение используются только в сложных текстовых редакторах, использующих шрифты с переменной шириной символов и шрифтовое выделение слов. Если вам нужен простой однострочный или многострочный редактор текста, вы можете воспользоваться стандартным средством Windows - окном, для которого в Windows зарегистрирован класс окна с именем EDIT. Поэтому если ваше приложение должно обеспечить ввод отдельных слов, чисел, заполнение простейших форм или создание несложных (без шрифтового выделения) текстов, лучше всего воспользоваться готовым классом окна типа редактор.
Другое дело - работа с функциональными клавишами и клавишами перемещения курсора. Чаще всего приложения Windows обрабатывают сообщения, поступающие от этих клавиш, для выполнения каких-либо действий, связанных, например, с просмотром текста в окне, переключением режимов работы или выполнением каких-либо других функций.
Так как операционная система Windows является графически ориентированной, приложения активно используют мышь. Однако обычно мышь дублируется клавиатурой, то есть те действия, которые вы можете выполнить с помощью мыши, можно выполнить и с помощью клавиатуры. Это не относится к графическим редакторам, которые часто становятся полностью неработоспособными, если компьютер не оснащен мышью.
Некоторые приложения могут даже полностью игнорировать присутствие клавиатуры.

Клавиатурные сообщения

От клавиатуры может поступать четыре сообщения - WM_KEYDOWN, WM_KEYUP, WM_SYSKEYDOWN, WM_SYSKEYUP. Когда вы нажимаете клави-

шу, генерируется сообщение WM_KEYDOWN или WM_SYSKEYDOWN, в зависимости от того, какая была нажата клавиша и была ли эта клавиша нажата в комбинации с клавишей <Alt>. При отпускании клавиши генерируется сообщение WM_KEYUP или WM_SYSKEYUP.
Клавиатурные сообщения с префиксом WM_SYS называются системными клавиатурными сообщениями. Приложение редко обрабатывает системные сообщения от клавиатуры, передавая их в неизменном виде функции DefWindowProc. Обработка системных клавиатурных сообщений обычно сводится к выбору элементов меню, переключения на другое окно или другое приложение. Системные клавиатурные сообщения предназначены для Windows.
Сообщения WM_KEYDOWN и WM_KEYUP, напротив, предназначены для приложения. Если приложение желает отреагировать на ту или иную клавишу или комбинацию клавиш, оно должно обработать соответствующее сообщение.
Когда окно сворачивается в пиктограмму, его тоже можно сделать активным. При этом надпись, расположенная около пиктограммы, выделяется изменением цвета фона. Ни одно окно в свернутом приложении не получает фокус ввода, поэтому все сообщения от клавиатуры транслируются в системные сообщения. Вы, однако, можете перехватить любое системное сообщение и определить для него собственный обработчик.

Параметры клавиатурных сообщений

Сообщения WM_KEYDOWN, WM_KEYUP, WM_SYSKEYDOWN, WM_SYSKEYUP передают информацию о нажатой клавише через параметры lParam и wParam.
Параметр lParam для этих сообщений содержит информацию низкого уровня, такую, как скан-код, счетчик повторов, флаг предыдущего состояния клавиши и т. п. Эта информация редко используется приложениями Windows.
Параметр wParam содержит код виртуальной клавиши, соответствующей нажатой физической клавише. Именно этот параметр используется приложениями для идентификации нажатой клавиши.
Приведем описание отдельных бит парамера lParam.

Бит	Описание
0-15	Счетчик повторов. Если нажать клавишу и держать ее в нажатом состоянии, несколько сообщений WM_KEYDOWN и WM_SYSKEYDOWN будут слиты в одно. Количество объединенных таким образом сообщений
16-23	OEM скан-код клавиши. Изготовители аппаратуры (OEM - Original Equipment Manufacturer) могут заложить в своей клавиатуре различное соответствие скан-кодов и обозначений клавиш. Скан-код генерируется клавиатурным контроллером. Это тот самый код, который получают в регистре AH программы MS-DOS, вызывая прерывание INT16h
24	Флаг расширенной клавиатуры. Этот бит установлен в 1, если сообщение соответствует клавише, имеющейся только на расширенной 101- или 102-клавишной клавиатуре. Это может быть одна из следующих клавиш: <Home>, <End>, <PgUp>, <PgDn>, <Insert>, <Delete>, клави-

	ши дополнительной клавиатуры.
25-26	Не используются
26-28	Зарезервированы для использования Windows
29	Код контекста. Этот бит равен 1, если сообщение соответствует комбинации клавиши <Alt> с любой другой, и 0 в противном случае
30	Предыдущее состояние клавиши. Если перед приходом сообщения клавиша, соответствующая сообщению, была в нажатом состоянии, этот бит равен 1. В противном случае бит равен 0
31	Флаг изменения состояния клавиши (transition state). Если клавиша была нажата, бит равен 0, если отпущена - 1

Если нажать клавишу и оставить ее в нажатом состоянии, функция окна может получить подряд несколько сообщений WM_KEYDOWN, прежде чем придет сообщение WM_KEYUP. В этом случае бит предыдущего состояния клавиши (бит 30) можно использовать для обнаружения сообщений, возникших в результате включения автоповтора клавиатуры. Приложение может игнорировать такие сообщения, исключая эффект накопления, когда приложение не может обрабатывать сообщения с такой скоростью, с какой они поступают. Например, если вы будете долго держать клавишу пролистывания страниц в текстовом редакторе, то после того, как вы ее отпустите, текстовый редактор будет еще долго листать ваш документ.
Для сообщений WM_KEYDOWN и WM_KEYUP значение кода контекста (бит 29) и флага изменения состояния (бит 31) всегда равно 0.
Для сообщений WM_SYSKEYUP и WM_SYSKEYDOWN бит 31 равен 1. Но есть два исключения. Во-первых, если активное окно свернуто в пиктограмму, все сообщения от клавиатуры преобразовываются в системные и, если клавиша <Alt> не нажата, код контекста равен 0. Во-вторых, на некоторых клавиатурах для ввода символов национального языка могут использоваться комбинации с участием клавиш <Alt>, <Control>, <Shift> и т. п. В этом случае код контекста может быть равен 1, хотя сообщение не является системным.
Обработчик клавиатурного сообщения должен возвратить значение 0 для всех перехваченных сообщений.
Параметр wParam содержит код виртуальной клавиши, соответствующей нажатой физической клавише. Код виртуальной клавиши не зависит от аппаратной реализации клавиатуры. Многие коды виртуальных клавиш имеют символьное обозначение, определенное в файле windows.h. Приведем полный список кодов виртуальных клавиш. Те из них, которые определены в файле windows.h, имеют префикс VK_ (Virtual Key).

Символическое имя	Код виртуальной клавиши	Клавиша, которой соответствует данный код	Клавиша на клавиатуре IBM PC
Не определено	0x0		
VK_LBUTTON	0x1	Левая клавиша мыши	
VK_RBUTTON	0x2	Правая клавиша мыши	

VK_CANCEL	0x3	<Control + Break>	<Control + Break>
VK_MBUTTON	0x4	Средняя клавиша мыши	
Не определено	0x5 - 0x6	Не определено	
VK_BACK	0x8	Клавиша забоя	Клавиша забоя <Backspace>
VK_TAB	0x9	Клавиша табуля-тора	<Tab>
Не определено	0xa - 0xb	Не определено	
VK_CLEAR	0xc	CLEAR	Соответствует клавише <5> дополнительной клавиатуры при выключенном режиме <Num Lock>
VK_RETURN	0xd	RETURN	<Enter>
Не определено	0xe - 0xf	Не определено	
VK_SHIFT	0x10	SHIFT	<Shift>
VK_CONTROL	0x11	CONTROL	<Control>
VK_MENU	0x12	MENU	<Alt>
VK_PAUSE	0x13	PAUSE	<Pause>
VK_CAPITAL	0x14	CAPITAL	<Caps Lock>
Не определено	0x15 - 0x19	Зарезервировано для систем Kanji	
Не определено	0x1a	Не определено	
VK_ESCAPE	1b	ESCAPE	<Esc>
Не определено	0x1c - 0x1f	Не определено	
VK_SPACE	0x20	Клавиша пробела SPACEBAR	Клавиша пробела
VK_PRIOR	0x21	PAGE UP	<PgUp>
VK_NEXT	0x22	PAGE DOWN	<PgDn>
VK_END	0x23	END	<End>
VK_HOME	0x24	HOME	<Home>
VK_LEFT	0x25	Перемещение курсора влево LEFT ARROW	Клавиша перемещения курсора влево <Left>
VK_UP	0x26	Перемещение курсора вверх UP ARROW	Клавиша перемещения курсора вверх <Up>
VK_RIGHT	0x26	Перемещение курсора вправо RIGHT ARROW	Клавиша перемещения курсора вправо <Right>

VK_DOWN	0x28	Перемещение курсора вниз DOWN ARROW	Клавиша перемещения курсора вниз <Down>
VK_SELECT	0x29	SELECT	
VK_PRINT	0x2a	Зависит от изготовителя клавиатуры	
VK_EXECUTE	0x2b	EXECUTE	
VK_SNAPSHOT	0x2c	PRINTSCREEN	<PrtSc>
VK_INSERT	0x2d	INSERT	<Insert>
VK_DELETE	0x2e	DELETE	<Delete>
VK_HELP	0x2f	HELP	
Не определено	0x30	0	<0>
Не определено	0x31	1	<1>
Не определено	0x32	2	<2>
Не определено	0x33	3	<3>
Не определено	0x34	4	<4>
Не определено	0x35	5	<5>
Не определено	0x36	6	<6>
Не определено	0x36	6	<6>
Не определено	0x38	8	<8>
Не определено	0x39	9	<9>
Не определено	0x3a - 0x40	Не определено	
Не определено	0x41	A	<A>
Не определено	0x42	B	<B>
Не определено	0x43	C	<C>
Не определено	0x44	D	<D>
Не определено	0x45	E	<E>
Не определено	0x46	F	<F>
Не определено	0x46	G	<G>
Не определено	0x48	H	<H>
Не определено	0x49	I	<I>
Не определено	0x4a	J	<J>
Не определено	0x4b	K	<K>
Не определено	0x4c	L	<L>
Не определено	0x4d	M	<M>
Не определено	0x4e	N	<N>
Не определено	0x4f	O	<O>
Не определено	0x50	P	<P>
Не определено	0x51	Q	<Q>

Не определено	0x52	R	<R>
Не определено	0x53	S	<S>
Не определено	0x54	T	<T>
Не определено	0x55	U	<U>
Не определено	0x56	V	<V>
Не определено	0x56	W	<W>
Не определено	0x58	X	<X>
Не определено	0x59	Y	<Y>
Не определено	0x5a	Z	<Z>
Не определено	0x5b - 0x5f	Не определено	
VK_NUMPAD0	0x60	0 на цифровой клавиатуре	<0> на цифровой клавиатуре
VK_NUMPAD1	0x61	1 на цифровой клавиатуре	<1> на цифровой клавиатуре
VK_NUMPAD2	0x62	2 на цифровой клавиатуре	<2> на цифровой клавиатуре
VK_NUMPAD3	0x63	3 на цифровой клавиатуре	<3> на цифровой клавиатуре
VK_NUMPAD4	0x64	4 на цифровой клавиатуре	<4> на цифровой клавиатуре
VK_NUMPAD5	0x65	5 на цифровой клавиатуре	<5> на цифровой клавиатуре
VK_NUMPAD6	0x66	6 на цифровой клавиатуре	<6> на цифровой клавиатуре
VK_NUMPAD6	0x66	6 на цифровой клавиатуре	<6> на цифровой клавиатуре
VK_NUMPAD8	0x68	8 на цифровой клавиатуре	<8> на цифровой клавиатуре
VK_NUMPAD9	0x69	9 на цифровой клавиатуре	<9> на цифровой клавиатуре
VK_MULTIPLAY	0x6a	Клавиша умноже-ния	<*> на цифровой клавиатуре
VK_ADD	0x6b	Клавиша сложения	<+> на цифровой клавиатуре
VK_SEPARATOR	0x6c	Клавиша разделе-ния	
VK_SUBTRACT	0x6d	Клавиша вычита-ния	<-> на цифровой клавиатуре
VK_DECIMAL	0x6e	Клавиша десятич-ной точки	<.> на цифровой клавиатуре
VK_DIVIDE	0x6f	Клавиша деления	</> на цифровой клавиатуре

VK_F1	0x60	F1	<F1>
VK_F2	0x61	F2	<F2>
VK_F3	0x62	F3	<F3>
VK_F4	0x63	F4	<F4>
VK_F5	0x64	F5	<F5>
VK_F6	0x65	F6	<F6>
VK_F6	0x66	F6	<F6>
VK_F8	0x66	F8	<F8>
VK_F9	0x68	F9	<F9>
VK_F10	0x69	F10	<F10>
VK_F11	0x6a	F11	<F11>
VK_F12	0x6b	F12	<F12>
VK_F13	0x6c	F13	
VK_F14	0x6d	F14	
VK_F15	0x6e	F15	
VK_F16	0x6f	F16	
Не определено	0x80 - 0x86	Зависит от изгото-вителя клавиатуры	
Не определено	0x88 - 0x8f	Не определено	
VK_NUMLOCK	0x90	NUM LOCK	<Num Lock>
VK_SCROLL	0x91	SCROLL LOCK	<Scroll Lock>
Не определено	0x92 - 0xb9	Не определено	
Не определено	0xba	Клавиша знака пунктуации	;
Не определено	0xbb	Плюс	+ =
Не определено	0xbc	Запятая	, <
Не определено	0xbd	Минус	- _
Не определено	0xbe	Точка	. >
Не определено	0xbf	Клавиша знака пунктуации	/ ?
Не определено	0xc0	Клавиша знака пунктуации	` ~
Не определено	0xc1 - 0xda	Не определено	
Не определено	0xdb	Клавиша знака пунктуации	[{
Не определено	0xdc	Клавиша знака пунктуации	\ |
Не определено	0xdd	Клавиша знака пунктуации	] }

Не определено	0xde	Клавиша знака пунктуации	' "
Не определено	0xdf	Клавиша знака пунктуации	
Не определено	0xe0 - 0xe1	Зависит от изгото-вителя клавиатуры	
Не определено	0xe2	Знак неравенства	
Не определено	0xe3 - 0xe4	Зависит от изгото-вителя клавиатуры	
Не определено	0xe5	Не определено	
Не определено	0xe6	Зависит от изгото-вителя клавиатуры	
Не определено	0xe6 - 0xe8	Не определено	
Не определено	0xe9 - 0xf5	Зависит от изгото-вителя клавиатуры	
Не определено	0xf6 - 0xff	Не определено	

Рассматривая приведенную выше таблицу, нетрудно заметить, что в ней есть коды виртуальных клавиш, которые невозможно получить в компьютере с IBM-совместимой клавиатурой. Кроме того, используя только эти коды, невозможно различить строчные и прописные буквы.

Для определения состояния клавиш <Shift>, <Caps Lock>, <Control>, <Num Lock> сразу после получения сообщения функция окна должна вызвать функцию с именем GetKeyState, которая входит в программный интерфейс Windows. Прототип этой функции: int WINAPI GetKeyState(int vkey);

Параметр функции vkey должен указывать код виртуальной клавиши, для которой необходимо вернуть состояние. Старший бит возвращаемого значения, установленный в 1, говорит о том, что указанная клавиша была нажата. Если этот бит равен 0, клавиша не была нажата.

Младший бит возвращаемого значения указывает состояние переключения. Если он равен 1, клавиша (такая, как <Caps Lock> или <Num Lock>) находится во включенном состоянии, то есть она была нажата нечетное число раз после включения компьютера. Учтите, что при помощи программы установки параметров компьютера SETUP, расположенной в BIOS, вы можете задать произвольное состояние переключающих клавиш после запуска системы.

Функция GetKeyState возвращает состояние клавиши на момент извлечения сообщения из очереди приложения функцией GetMessage.

Для того чтобы узнать состояние клавиш в любой произвольный момент времени, можно воспользоваться функцией GetAsyncKeyState:

int WINAPI GetAsyncKeyState (int vkey);

Параметр функции vkey должен указывать код виртуальной клавиши, для которой необходимо вернуть состояние. Старший бит возвращаемого значения, установленный в 1, говорит о том, что указанная клавиша была нажата в момент вызова функции. Если этот бит равен 0, клавиша не была нажата. Младший бит возвращаемого

значения установлен в 1, если указанная клавиша была нажата с момента последнего вызова функции GetAsyncKeyState.
Если для функции в качестве параметра задать значения VK_LBUTTON или VK_RBUTTON, можно узнать состояние клавиш, расположенных на корпусе мыши.
Есть возможность определить и изменить состояние для всех клавиш одновременно.
Для определения состояния клавиш воспользуйтесь функцией GetKeyboardState:
void WINAPI GetKeyboardState (BYTE FAR* lpbKeyState);
Единственный параметр lpbKeyState этой функции - указатель на массив из 256 байт. После вызова функции этот массив будет заполнен информацией о состоянии всех виртуальных клавиш в момент генерации клавиатурного сообщения. В этом смысле функция аналогична функции GetKeyState.
Для любого байта массива установленный в 1 старший бит означает, что соответствующая клавиша была нажата. Если этот бит равен 0, клавиша не была нажата.
Младший бит, установленный в 1, означает, что клавиша была переключена. Если младший бит равен 0, клавиша не была переключена.
После вызова функции GetKeyboardState вы можете изменить содержимое массива и вызвать функцию SetKeyboardState, изменяющую состояние клавиатуры:

```
void WINAPI SetKeyboardState(BYTE FAR* lpbKeyState);
```

Функция GetKeyboardType позволит вам определить тип клавиатуры и количество имеющихся на ней функциональных клавиш:

```
int WINAPI GetKeyboardType(int fnKeybInfo);
```

В зависимости от значения параметра fnKeybInfo функция может возвращать различную информацию о клавиатуре.
Если задать значение параметра fnKeybInfo, равное 0, функция вернет код типа клавиатуры.Если задать значение параметра, равное 1, функция вернет код подтипа клавиатуры. И наконец, если задать значение параметра, равное 2, функция вернет количество функциональных клавиш, имеющихся на клавиатуре.
Интересна также функция GetKeyNameText, возвращающая для заданного кода виртуальной клавиши название соответствующей клавиши в виде текстовой строки.
Названия виртуальных клавиш определены в драйвере клавиатуры.
Прототип функции GetKeyNameText:
int WINAPI GetKeyNameText(LONG lParam, LPSTR lpszBuffer, int cbMaxKey);
Первый параметр функции lParam должен определять клавишу в формате компоненты lParam клавиатурного сообщения. Вы можете использовать в качестве этого параметра значение lParam, полученное функцией окна вместе с любым клавиатурным сообщением, таким, как WM_KEYDOWN или WM_SYSKEYDOWN.
Второй параметр - lpszBuffer является указателем на буфер, в который будет записано название клавиши.Третий параметр - cbMaxKey должен быть равен длине буфера, уменьшенной на 1.

Символьные клавиатурные сообщения

Рассмотренные клавиатурные сообщения хорошо подходят для работы с функциональными клавишами и клавишами перемещения курсора, но они непригодны для работы с обычными символьными клавишами. Если вам надо сделать, например, собственный текстовый редактор, вы должны уметь различать прописные и строч-

ные буквы, учитывая нижний и верхний регистр, а также учитывать особенности работы с национальными алфавитами.
Теоретически распознавание регистра можно выполнить с помощью описанной функции GetKeyState, однако есть более удобный способ, который к тому же обеспечивает учет национальных алфавитов. Этот способ основан на применении функции TranslateMessage, которая включается в цикл обработки сообщений:

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

Функция TranslateMessage преобразует клавиатурные сообщения WM_KEYDOWN, WM_KEYUP, WM_SYSKEYDOWN и WM_SYSKEYUP в символьные сообщения WM_CHAR, WM_DEADCHAR, WM_SYSCHAR, WM_SYSDEADCHAR. Образованные символьные сообщения помещаются в очередь сообщений приложения, причем оригинальные клавиатурные сообщения из этой очереди не удаляются.
Параметр lParam для символьного сообщения равен параметру lParam клавиатурного сообщения, из которого первое было образовано, то есть в процессе преобразования значение lParam не изменяется.
Параметр wParam содержит код символа, соответствующего нажатой клавише в так называемом стандарте ANSI, принятом в Windows для представления символов. Этот код определяется функцией TranslateMessage с учетом состояния клавиш <Control>, <Shift>, <Alt>, <Caps Lock> и используемого национального языка.
Из всех четырех символьных сообщений приложения чаще всего используют сообщение WM_CHAR, которое передается функции окна в результате трансляции сообщения WM_KEYDOWN. Сообщение WM_SYSCHAR образуется из сообщения WM_SYSKEYDOWN и обычно игнорируется приложением (передается функции DefWindowProc).
Сообщения WM_DEADCHAR и WM_SYSDEADCHAR образуются при использовании клавиатур, имеющих дополнительную клавишу для снабжения символов диакритическими знаками (например, символ "Ў" снабжен диакритическим знаком). Такие дополнительные клавиши называются "мертвыми" клавишами, так как они не образуют символов, а лишь изменяют действие следующей нажимаемой клавиши. Эти клавиши определяются на основе информации об используемом национальном алфавите.
Если после "мертвой" клавиши была нажата правильная, обычная клавиша (не все символы могут иметь диакритические знаки), в очередь приложения помещаются два сообщения - WM_DEADCHAR и WM_CHAR. Последнее в параметре wParam передает ANSI-код введенного символа, имеющего диакритический знак.
Если после "мертвой" клавиши была нажата клавиша, соответствующая символу, который не может иметь диакритического знака, то в очередь приложения после сообщения WM_DEADCHAR будут записаны два сообщения WM_CHAR. Первое сообщение будет соответствовать коду "мертвой" клавиши, интерпретированному как код символа, второе - коду клавиши, нажатой после "мертвой".
Поэтому приложению достаточно обрабатывать только сообщение WM_CHAR, игнорируя сообщение WM_DEADCHAR (за исключением тех случаев, когда после нажатия "мертвой" клавиши на экране необходимо отобразить диакритический

знак). Параметр wParam сообщения WM_CHAR будет содержать правильный ANSI-код символа, учитывающий использование "мертвых" клавиш.

Функции для перекодировки символьных строк.

Для перекодировки строки символов, закрытой двоичным нулем, из набора ANSI в набор OEM предназначена функция AnsiToOem:

void WINAPI AnsiToOem(const char _huge* hpszWindowsStr, char _huge* hpszOemStr);

Первый параметр (hpszWindowsStr) представляет собой указатель типа _huge на преобразуемую строку, второй (hpszOemStr) - на буфер для записи результата преобразования.

Похожая по назначению функция AnsiToOemBuff выполняет преобразование буфера заданного размера:

void WINAPI AnsiToOemBuff(LPCSTR lpszWindowsStr, LPSTR lpszOemStr, UINT cbWindowsStr);

Первый параметр этой функции (lpszWindowsStr) является дальним указателем на буфер, содержащий преобразуемые данные, второй (lpszOemStr) - на буфер для записи результата. Третий параметр (cbWindowsStr) определяет размер входного буфера, причем нулевой размер соответствует буферу длиной 64 Кбайт (65536 байт).

Обратное преобразование выполняется функциями OemToAnsi и OemToAnsiBuff:

void WINAPI OemToAnsi(const char _huge* hpszOemStr, char _huge* lpszWindowsStr);

void WINAPI OemToAnsiBuff(LPCSTR lpszOemStr, LPSTR lpszWindowsStr, UINT cbOemStr);

Назначение параметров этих функций аналогично назначению параметров функций AnsiToOem и AnsiToOemBuff.

Для преобразований символов в строчные или прописные приложение Windows должно пользоваться функциями AnsiLower, AnsiLowerBuff, AnsiUpper, AnsiUpperBuff.

Функция AnsiLower преобразует закрытую двоичным нулем текстовую строку в строчные буквы: LPSTR WINAPI AnsiLower(LPSTR);

Единственный параметр функции - дальний указатель на преобразуемую строку.

Функция AnsiUpper преобразует закрытую двоичным нулем текстовую строку в прописные буквы:LPSTR WINAPI AnsiLower(LPSTR lpsz);

Параметр функции lpsz - дальний указатель на преобразуемую строку.

Вы можете использовать эти функции для преобразования одного символа, если запишете этот символ в младший байт слова, старший байт этого слова сбросите в ноль и затем укажете адрес слова в качестве параметра.

Функция AnsiLowerBuff позволяет преобразовать в строчные буквы заданное количество символов:UINT WINAPI AnsiLowerBuff(LPSTR lpszString, UINT cbString);

Первый параметр функции (lpszString) является указателем на буфер, содержащий преобразуемые символы, второй (cbString) определяет количество преобразуемых символов (размер буфера). Нулевой размер соответствует буферу длиной 64 Кбайт (65536 байт).

Функция возвращает количество преобразованных символов.

Функция AnsiUpperBuff позволяет преобразовать в прописные буквы заданное количество символов:UINT WINAPI AnsiUpperBuff(LPSTR lpszString, UINT cbString);
Первый параметр функции lpszString(lpszString) является указателем на буфер, содержащий преобразуемые символы, второй (cbString) определяет количество преобразуемых символов (размер буфера). Нулевой размер соответствует буферу длиной 64 Кбайт (65536 байт).
Эта функция, как и предыдущая, возвращает количество преобразованных символов.
Еще одна проблема связана с необходимостью позиционирования вдоль текстовой строки. Если используется однобайтовое представление символов, позиционирование сводится к увеличению или уменьшению значения указателя на один байт. Однако в некоторых национальных языках (например, в японском) набор символов OEM для представления каждого символа использует два байта. Для правильного позиционирования (с учетом различных наборов символов) необходимо использовать специальные функции AnsiNext и AnsiPrev, которые входят в состав программного интерфейса Windows.
Функция возвращает новое значение для указателя, передвинутое вперед по строке на одни символ: LPSTR WINAPI AnsiNext(LPCSTR lpchCurrentChar);
Параметр функции указывает на текущий символ. Возвращаемое значение является указателем на следующий символ в строке или на закрывающий строку двоичный ноль.
Функция AnsiPrev выполняет передвижение указателя в направлении к началу строки: LPSTR WINAPI AnsiPrev(LPCSTR lpchStart, LPCSTR lpchCurrentChar);
Первый параметр функции указывает на начало строки (на первый символ строки). Второй параметр - указатель на текущий символ. Функция возвращает значение указателя, соответствующее предыдущему символу или первому символу в строке, если при продвижении достигнуто начало строки.
В составе программного интерфейса Windows имеются функции для преобразования символа ANSI в код виртуальной клавиши (VkKeyScan) или в соответствующий OEM скан-код и состояние (OemKeyScan).
Функция VkKeyScan используется для преобразования кода символа ANSI в код и состояние виртуальной клавиши: UINT WINAPI VkKeyScan(UINT uChar);
Параметр функции определяет символ ANSI, который будет преобразован в код виртуальной клавиши.
Младший байт возвращаемого значения содержит код виртуальной клавиши, старший - состояние клавиш сдвига (<Shift>, <Alt>, <Control>):

Значение	Описание
1	При выводе символа была нажата клавиша сдвига
2	Символ является управляющим
3 - 5	Данная комбинация клавиш сдвига не используется для представления символов
6	Символ образован при помощи комбинации клавиш <Control+Alt>
6	Символ образован при помощи комбинации клавиш <Shift+Control+Alt>

Эта функция обычно используется приложениями, которые передают символы другим приложениям с помощью сообщений WM_KEYDOWN и WM_KEYUP (то есть симулируют ввод с клавиатуры).
Функция OemKeyScan преобразует символ OEM в скан-код и состояние для набора OEM: DWORD WINAPI OemKeyScan(UINT uOemChar);
Параметр функции определяет символ OEM, который будет преобразован в скан-код.
Младшее слово возвращаемого значения содержит OEM скан-код для указанного символа.
Старшее слово указывает состояние клавиш сдвига для заданного символа. Если в этом слове установлен бит 1, нажата клавиша <Shift>, если бит 2 - клавиша <Control>.
Если преобразуемое значение не принадлежит к набору OEM, возвращается значение -1 (и в старшем, и в младшем слове).

4.2. Работа с сообщениями мыши.

За исключением простейших случаев, приложение Windows должно позволять пользователю выполнять большинство операций без клавиатуры, за исключением набора текста.
Определить присутствие мыши можно с помощью функции GetSystemMetrics, передав ей в качестве параметра значение SM_MOUSEPRESENT. Если мышь есть, эта функция возвращает ненулевое значение.

Сообщения, поступающие от мыши

Мышь может порождать много сообщений, всего их 22! Однако большинство из них можно передать функции DefWindowProc без обработки. Сообщения, поступающие от мыши, содержат информацию о текущем расположении курсора, о его расположении в момент, когда вы нажимаете на клавиши мыши, и другую аналогичную информацию.
Существует два режима, определяющих два способа распределения сообщений от мыши.
В первом режиме, который установлен по умолчанию, сообщения от мыши направляются функции окна, расположенного под курсором мыши. Если в главном окне приложения создано дочернее окно и курсор мыши располагается над дочерним окном, сообщения мыши попадут в функцию дочернего окна, но не в функцию главного окна приложения. Это же касается и временных (pop-up) окон.
Во втором режиме окно может захватить мышь для монопольного использования. В этом случае функция этого окна будет всегда получать все сообщения мыши, независимо от расположения курсора мыши. Для того чтобы захватить мышь, приложение должно вызвать функцию SetCapture: HWND WINAPI SetCapture(HWND hwnd);
Параметр hwnd функции указывает идентификатор окна, которое будет получать все сообщения от мыши вне зависимости от расположения курсора.
Функция SetCapture возвращает идентификатор окна, которое захватывало мышь до вызова функции или NULL, если такого окна не было.
Функция ReleaseCapture возвращает нормальный режим обработки сообщений мыши: void WINAPI ReleaseCapture(void);
Эта функция не имеет параметров и не возвращает никакого значения.

Функция GetCapture позволяет определить идентификатор окна, захватившего мышь: HWND WINAPI GetCapture(void);
Если ни одно окно не захватывало мышь, эта функция возвратит значение NULL.
В любом случае на получение сообщений от мыши никак не влияет факт приобретения или потери окном фокуса ввода.
Приведем полный список сообщений, поступающих от мыши.

Сообщение	Описание
WM_LBUTTONDBLCLK	Двойной щелчок левой клавишей мыши во внутренней (client) области окна
WM_LBUTTONDOWN	Нажата левая клавиша мыши во внутренней области окна
WM_LBUTTONUP	Отпущена левая клавиша мыши во внутренней области окна
WM_MBUTTONDBLCLK	Двойной щелчок средней клавишей мыши во внутренней области окна
WM_MBUTTONDOWN	Нажата средняя клавиша мыши во внутренней области окна
WM_MBUTTOMUP	Отпущена средняя клавиша мыши во внутренней области окна
WM_MOUSEMOVE	Перемещение курсора мыши во внутренней области окна
WM_RBUTTONDBLCLK	Двойной щелчок правой клавишей мыши во внутренней области окна
WM_RBUTTONDOWN	Нажата правая клавиша мыши во внутренней области окна
WM_RBUTTONUP	Отпущена правая клавиша мыши во внутренней области окна
WM_NCHITTEST	Перемещение мыши в любом месте экрана
WM_MOUSEACTIVE	Нажата клавиша мыши над неактивным окном
WM_NCLBUTTONDBLCLK	Двойной щелчок левой клавишей мыши во внешней (non-client) области окна
WM_NCLBUTTONDOWN	Нажата левая клавиша мыши во внешней области окна
WM_NCLBUTTONUP	Отпущена левая клавиша мыши во внешней области окна
WM_NCMBUTTONDBLCLK	Двойной щелчок средней клавишей мыши во внешней области окна
WM_NCMBUTTONDOWN	Нажата средняя клавиша мыши во внешней области окна
WM_NCMBUTTOMUP	Отпущена средняя клавиша мыши во внешней области окна

WM_NCMOUSEMOVE	Перемещение курсора мыши во внешней области окна
WM_NCRBUTTONDBLCLK	Двойной щелчок правой клавишей мыши во внешней области окна
WM_NCRBUTTONDOWN	Нажата правая клавиша мыши во внешней области окна
WM_NCRBUTTONUP	Отпущена правая клавиша мыши во внешней области окна

Из приведенных выше 22 сообщений 21 сообщение образуется из сообщения WM_NCHITTEST. Это сообщение генерируется драйвером мыши при любых перемещениях мыши. Разумеется, драйвер не отслеживает перемещение мыши для каждого пиксела экрана. Период возникновения сообщений WM_NCHITTEST зависит от скорости перемещения мыши, параметров драйвера, аппаратуры мыши и т. п. Сообщение WM_NCHITTEST не использует параметр wParam. В младшем слове параметра lParam передается горизонтальная позиция курсора мыши, а в старшем - вертикальная. Координаты вычисляются относительно верхнего левого угла экрана. Приложения редко обрабатывают сообщение WM_NCHITTEST, обычно оно передается функции DefWindowProc. Получив это сообщение, функция DefWindowProc определяет положение курсора мыши относительно расположенных на экране объектов и возвращает одно из приведенных ниже значений, описанных в файле windows.h).

Значение	Расположение курсора мыши
HTBORDER	На рамке окна, которое создано без толстой рамки, предназначенной для изменения размера окна
HTBOTTOM	На нижней горизонтальной линии рамки окна
HTBOTTOMLEFT	В левом нижнем углу рамки
HTBOTTOMRIGHT	В правом нижнем углу рамки
HTCAPTION	На заголовке окна (title-bar)
HTCLIENT	Во внутренней области окна (client area)
HTERROR	Над поверхностью экрана или на линии, разделяющей различные окна. Дополнительно функция DefWindowProc выдает звуковой сигнал
HTGROWBOX	В области изменения размера окна (size box)
HTHSCROLL	На горизонтальной полосе просмотра
HTLEFT	На левой вертикальной линии рамки окна
HTMAXBUTTON	На кнопке максимизиции
HTMENU	В области меню
HTMINBUTTON	На кнопке минимизации
HTNOWHERE	Над поверхностью экрана или на линии, разде-

	ляющей различные окна
HTREDUCE	В области минимизации
HTRIGHT	На правой вертикальной линии рамки окна
HTSIZE	В области изменения размера окна (size box). То же самое, что и HTGROWBOX
HTSYSMENU	В области системного меню
HTTOP	На верхней горизонтальной линии рамки окна
HTTOPLEFT	В верхнем левом углу рамки окна
HTTOPRIGHT	В правом верхнем углу рамки окна
HTTRANSPARENT	В окне, которое перекрыто другим окном
HTVSCROLL	На вертикальной полосе просмотра
HTZOOM	В области максимизиции

После обработки сообщения WM_HITTEST Windows анализирует расположение курсора и генерирует одно из сообщений, описанных выше.
Если курсор находится во внутренней области окна (client area), функция DefWindowProc возвращает значение HTCLIENT. В этом случае функция окна, над которой находится курсор мыши (или функция окна, захватившая мышь), будет получать сообщения о событиях во внутренней области окна. Это все описанные выше сообщения, кроме сообщений с префиксом WM_NC и сообщения WM_MOUSEACTIVATE (сочетание букв NC в символическом имени сообщения означает Non Client).
Внешняя область окна (Non Client) соответствует пространству между внешним контуром окна и его внутренней областью. В этом пространстве располагаются такие элементы, как рамка окна, кнопки максимизиции и минимизации, системное меню и меню окна и т. п.
Если при обработке сообщения WM_HITTEST выясняется, что курсор мыши расположен во внешней области окна, функция окна получает сообщения мыши с префиксом WM_NC.
Эти сообщения также редко используются приложениями и обычно передаются функции DefWindowProc. Обработка сообщений с префиксом WM_NC заключается в перемещении окна, изменении его размеров, активизации меню и т. д., в зависимости от самого сообщения. Ваше приложение может перехватить сообщения с префиксом WM_NC, но, если эти сообщения не будут переданы функции DefWindowProc, Windows не сможет выполнять соответствующие им действия.

Сообщения для внутренней области окна

Эти сообщения генерируются в том случае, если при обработке сообщения WM_HITTEST функция DefWindowProc вернула значение HTCLIENT.
Для всех сообщений из этой группы параметр lParam содержит координаты курсора мыши, а параметр wParam - значение, с помощью которого можно определить, какие клавиши на мыши и клавиатуре были нажаты в тот момент, когда произошло событие, связанное с сообщением.
Младшее слово параметра lParam содержит горизонтальные координаты курсора мыши, старшее - вертикальные.

Параметр wParam может состоять из отдельных битовых флагов, перечисленных ниже.

Значение	Описание
MK_CONTROL	На клавиатуре была нажата клавиша <Control>
MK_LBUTTON	Была нажата левая клавиша мыши
MK_MBUTTON	Была нажата средняя клавиша мыши
MK_RBUTTON	Была нажата правая клавиша мыши
MK_SHIFT	На клавиатуре была нажата клавиша <Shift>

Анализируя параметр wParam, приложение может определить, были ли в момент события нажаты какие-либо клавиши мыши или клавиши <Control> и <Shift>, расположенные на клавиатуре.

Следует учесть, что вы можете нажать клавишу мыши, когда курсор находится над одним окном, затем переместить курсор в другое окно и там отпустить клавишу мыши. В этом случае одно из сообщений о том, что была нажата клавиша мыши (WM_LBUTTONDOWN, WM_RBUTTONDOWN или WM_MBUTTONDOWN), попадет в функцию первого окна, а сообщение о том, что клавиша мыши была отпущена (WM_LBUTTONUP, WM_RBUTTONUP или WM_MBUTTONUP), - во второе. Когда мы работали с клавиатурными сообщениями, вслед за сообщением о том, что клавиша была нажата, всегда следовало сообщение о том, что клавиша была отпущена. При обработке сообщений мыши ваша функция окна может получить сообщение о том, что клавиша мыши была отпущена без предварительного сообщения о том, что она была нажата. Аналогично, вы можете никогда не дождаться сообщения об отпускании клавиши мыши после прихода сообщения о том, что клавиша мыши нажата: это сообщение может уйти в другое окно.

Следует сделать особое замечание относительно сообщений о двойном щелчке мыши. Это сообщения WM_LBUTTONDBLCLK, WM_MBUTTONDBLCLK и WM_RBUTTONDBLCLK.

Двойным щелчком (double click) называется пара одиночных щелчков, между которыми прошло достаточно мало времени. Изменить значение интервала, в течение которого должны поступить два одиночных щелчка, чтобы система распознала их как один двойной щелчок, проще всего при помощи стандартного приложения Windows с именем Control Panel.

Еще одно условие распознавания двойного щелчка менее очевидно и заключается в том, что за интервал между двумя одиночными щелчками курсор мыши не должен переместиться на слишком большое расстояние. С помощью функции GetSystemMetrics вы можете определить размеры прямоугольника, внутри которого должны быть сделаны два щелчка мышью, для того чтобы они могли распознаваться как один двойной щелчок. Для этого ей надо передать в качестве параметра значения SM_CXDOUBLECLK (ширина прямоугольника) и SM_CYDOUBLECLK (высота прямоугольника).

Кроме всего этого, для того чтобы окно могло получать сообщения о двойном щелчке мышью, при регистрации класса окна необходимо определить стиль класса окна CS_DBLCLKS.

Если выполнить двойной щелчок левой клавишей мыши в окне, для класса которого не определен стиль CS_DBLCLKS, функция окна последовательно получит следу-

ющие сообщения: WM_LBUTTONDOWN ,WM_LBUTTONUP, WM_LBUTTONDOWN ,WM_LBUTTONUP
Если же сделать то же самое в окне, способном принимать сообщения о двойном щелчке, функция окна в ответ на двойной щелчок получит следующую последовательность сообщений: WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDBLCLK, WM_LBUTTONUP
Как нетрудно заметить, перед сообщением WM_LBUTTONDBLCLK функция окна получит сообщение WM_LBUTTONDOWN. Дело в том, что после первого щелчка Windows еще не знает, будет ли следом обычный или двойной щелчок, - все зависит от интервала времени и перемещения курсора.
Обычно правильно спроектированное приложение по двойному щелчку выполняет ту же самую операцию, которую оно бы выполнило по второму щелчку из серии. Например, по одиночному щелчку мыши приложение может выбрать команду из меню. Для выбранной команды одиночный щелчок может запустить ее на выполнение. Тогда двойной щелчок должен сразу запускать команду на выполнение. В этом случае одиночный щелчок выберет команду, а двойной - запустит ее. Если вы соберетесь сделать двойной щелчок, а получится два одиночных, ничего особенного не произойдет - приложение будет вести себя так, как вы и ожидаете, а именно, запустится выбранная вами команда.
Сообщение WM_MOUSEMOVE извещает приложение о перемещении курсора мыши. С помощью этого сообщения приложение может, например, рисовать в окне линии вслед за перемещением курсора.
Последнее сообщение из группы сообщений для внутренней области окна имеет имя WM_MOUSEACTIVATE. Оно посылается функции неактивного окна, когда вы помещаете в это окно курсор мыши и делаете щелчок левой или правой клавишей. Если передать это сообщение функции DefWindowProc, в ответ на него Windows сделает указанное окно активным.
Сообщение WM_MOUSEACTIVATE передает параметры wParam и lParam.
Параметр wParam содержит идентификатор окна, которое будет активным. Если активным становится окно, имеющее дочерние окна, передается идентификатор самого старшего, родительского окна.
Младшее слово параметра lParam содержит результат обработки сообщения WM_NCHITTEST функцией DefWindowProc. Мы описали возможные значения, когда рассказывали о сообщении WM_NCHITTEST.
Старшее слово параметра lParam содержит код сообщения мыши, соответствующий способу, которым данное окно было выбрано. Это может быть код сообщений типа WM_LBUTTONDOWN, WM_RBUTTONDOWN и т. п.
Для сообщения WM_MOUSEACTIVATE определен код возврата:

Код возврата	Описание
MA_ACTIVATE	Сделать окно активным
MA_ACTIVATEANDEAT	Не делать окно активным
MA_NOACTIVATE	Сделать окно активным и удалить события, связанные с мышью
MA_NOACTIVATEANDEAT	Не делать окно активным и удалить события, связанные с мышью

Курсор мыши

Курсор мыши можно перемещать не только с помощью специальных функций, но и вручную. Вы также можете изменять форму курсора мыши. Можно определить форму курсора (или, иными словами, определить курсор) при регистрации класса окна или изменить ее в любое время в процессе работы приложения.
При регистрации класса окна мы задавали форму курсора следующим способом:
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
Второй параметр функции LoadCursor в нашем случае выбирает одну из предопределенных форм курсора, а именно стандартный курсор в виде стрелки. Возможные значения для выбора предопределенных форм курсора представлены ниже.

Символическое имя	Описание
IDC_ARROW	Стандартный курсор в виде стрелки
IDC_CROSS	Курсор в виде перекрещивающихся линий
IDC_IBEAM	Текстовый курсор в виде буквы "I"
IDC_ICON	Пустая пиктограмма
IDC_SIZE	Курсор в виде четырех стрелок, указывающих в разных направлениях
IDC_SIZENESW	Двойная стрелка, указывающая в северо-восточном и юго-западном направлении
IDC_SIZENS	Двойная стрелка, указывающая в севером и южном направлении
IDC_SIZENWSE	Двойная стрелка, указывающая в северо-западном и юго-восточном направлении
IDC_SIZEWE	Двойная стрелка, указывающая в восточном и западном направлении
IDC_UPARROW	Вертикальная стрелка
IDC_WAIT	Курсор в виде песочных часов

Вы можете попробовать изменить курсор в любом из уже рассмотренных нами ранее приложений, создающих окна.
Вы можете создать курсор произвольной формы с помощью такого средства, как редактора ресурсов. В этом случае вы должны нарисовать курсор в виде небольшой картинки, состоящей из отдельных точек. Эта картинка создается специальным графическим редактором и сохраняется в файле с расширением .cur. Затем файл подключается к ресурсам приложения, которые записываются в исполняемый exe-файл. Каждый ресурс в файле имеет свой идентификатор. Вы можете изменить форму курсора, если укажете идентификатор ресурса, соответствующего новому изображению курсора.
Для того чтобы можно было изменить форму курсора, прежде всего надо загрузить новый курсор при помощи функции LoadCursor, которая входит в программный интерфейс Windows:

```
HCURSOR WINAPI LoadCursor(HINSTANCE hinst, LPCSTR pszCursor);
```

Для загрузки нового курсора из ресурсов приложения в качестве первого параметра (hinst) необходимо указать идентификатор приложения, полученный через парамет-

ры функции WinMain. Второй параметр (lpszCursor) при этом должен указывать на идентификатор ресурса.
Если же в качестве первого параметра указать значение NULL, для загрузки курсора можно использовать перечисленные выше символические имена с префиксом IDC_. Именно так мы и поступаем при регистрации класса окна:
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
Функция LoadCursor возвращает идентификатор загруженного курсора или NULL при ошибке.
Для динамического изменения формы курсора (например, во время обработки сообщения) следует использовать функцию SetCursor:
HCURSOR WINAPI SetCursor(HCURSOR hcur);
Параметр hcur функции SetCursor должен указывать идентификатор нового курсора, подготовленный при помощи функции LoadCursor. Если указать параметр как NULL, изображение курсора исчезнет с экрана.
Для того чтобы выключить изображение курсора мыши или вновь включить его, используют функцию ShowCursor:
int WINAPI ShowCursor(BOOL fShow);
Функция управляет содержимым счетчика, который используется для определения момента включения или выключения изображения курсора мыши. Первоначально содержимое счетчика равно нулю. Этот счетчик увеличивается, когда необходимо включить курсор, и уменьшается при выключении курсора. Если счетчик больше или равен нулю, курсор мыши находится во включенном (видимом) состоянии.
Для включения курсора в качестве параметра fShow функции следует передать значение TRUE, для выключения - FALSE.
Возвращаемое функцией ShowCursor значение равно новому содержимому счетчика.
Наблюдая за работой стандартных приложений Windows, вы можете заметить, что часто на время выполнения длительных операций курсор принимает форму песочных часов. Как правило, все такие операции выполняются во время обработки какого-либо одного сообщения. Перед началом выполнения операции вы можете вызвать функцию LoadCursor с параметром IDC_WAIT, а затем вернуть прежнюю форму, вызвав эту же функцию еще раз. Дополнительно на время выполнения операции обработчик сообщения должен захватить мышь, вызвав функцию SetCapture. В этом случае вы не сможете с помощью мыши переключиться на другое приложение и прервать таким образом ход длительной операции. После выполнения операции следует освободить мышь, вызвав функцию ReleaseCapture.
Ваше приложение может установить курсор мыши в новое положение или определить текущие координаты курсора.
Для установки курсора мыши в новое положение следует вызвать функцию SetCursorPos: void WINAPI SetCursorPos(int x, int y);
Первый параметр функции определяет горизонтальную координату курсора, второй вертикальную. Начало системы координат расположено в верхнем левом углу экрана.
Для определения текущих экранных координат курсора мыши необходимо использовать функцию GetCursorPos: void WINAPI GetCursorPos(POINT FAR* lppt);

Эта функция записывает в поля x и y структуры типа POINT соответственно горизонтальную и вертикальную координату курсора мыши.
Существует редко используемая возможность ограничения перемещения курсора прямоугольной областью. Для ограничения свободы перемещения курсора следует использовать функцию ClipCursor: void WINAPI ClipCursor(const RECT FAR* lprc);
В качестве параметра lprc функции передается указатель на структуру типа RECT, в которой указываются координаты области ограничения. Как только необходимость в ограничении пропадет, следует освободить движение мыши, вызвав функцию ClipCursor с параметром NULL.
Программный интерфейс Windows версии 3.1 содержит функцию GetClipCursor, с помощью которой можно определить расположение и размер области, ограничивающей движение курсора: void WINAPI GetClipCursor(RECT FAR* lprc);
В качестве параметра lprc функции передается указатель на структуру типа RECT, в которую будут записаны координаты области ограничения.

Управление курсором мыши с помощью клавиатуры

Ваше приложение должно, по возможности, работать и без мыши. Для этого ему необходимо вначале включить курсор мыши (который по умолчанию находится в невидимом состоянии, если компьютер не оборудован мышью), а затем передвигать его самостоятельно. Для передвижения курсора мыши можно использовать, например, клавиши перемещения текстового курсора.

4.3.Обработка ссобщений таймера.

Во многих программах требуется следить за временем или выполнять какие-либо периодические действия.
Операционная система Windows позволяет для каждого приложения создать несколько виртуальных таймеров. Все эти таймеры работают по прерываниям одного физического таймера.
Так как работа Windows основана на передаче сообщений, работа виртуального таймера также основана на передаче сообщений. Приложение может заказать для любого своего окна несколько таймеров, которые будут периодически посылать в функцию окна сообщение с кодом WM_TIMER.
Есть и другой способ, также основанный на передаче сообщений. При использовании этого способа сообщения WM_TIMER посылаются не функции окна, а специальной функции, описанной с ключевым словом CALLBACK. Эта функция напоминает функцию окна и, так же как и функция окна, вызывается не из приложения, а из Windows. Функции, которые вызываются из Windows, называются функциями обратного вызова (callback function). Функция окна и функция, специально предназначенная для обработки сообщений таймера, являются примерами функций обратного вызова.
Точность виртуального таймера невысока. Сообщения таймера проходят через очередь приложения, к тому же другое приложение может блокировать на некоторое время работу вашего приложения. Поэтому сообщения от таймера приходят в общем случае нерегулярно. Кроме того, несмотря на возможность указания интервалов времени в миллисекундах, реальная дискретность таймера определяется перио-

дом прерываний, посылаемых таймером. Этот период (то есть длительность одного такта таймера) можно узнать с помощью функции GetTimerResolution:
DWORD WINAPI GetTimerResolution(void);
Нерегулярность прихода сообщений таймера не вызывает особых проблем, если речь не идет о работе в режиме реального времени. Системы реального времени, основанные на Windows, должны использовать для устройств ввода/вывода, критичных к скорости реакции системы, специальные драйверы. Строго говоря, операционная система Windows не предназначена для работы в режиме реального времени. Windows ориентирована на работу с человеком, когда небольшие задержки событий во времени не имеют никакого значения.

Создание и уничтожение таймера

Для создания виртуального таймера приложение должно использовать функцию SetTimer:
UINT WINAPI SetTimer(HWND hwnd, UINT idTimer, UINT uTimeout, TIMERPROC tmprc);
Первый параметр функции (hwnd) должен содержать идентификатор окна, функция которого будет получать сообщения от таймера, или NULL. В последнем случае с создаваемым таймером не связывается никакое окно и сообщения от таймера будут приходить в специально созданную для этого функцию.
Второй параметр (idTimer) определяет идентификатор таймера (он не должен быть равен нулю). Идентификатор используется только в том случае, если первый параметр функции SetTimer содержит идентификатор окна. Так как для одного окна можно создать несколько таймеров, для того чтобы различать сообщения, приходящие от разных таймеров, приложение при создании должно снабдить каждый таймер собственным идентификатором.
Если первый параметр указан как NULL, второй параметр функции игнорируется, так как для таймера задана специальная функция, получающая сообщения только от этого таймера.
Третий параметр (uTimeout) определяет период следования сообщений от таймера в миллисекундах. Учтите, что физический таймер тикает приблизительно 18,21 раза в секунду (точное значение составляет 1000/54,925). Поэтому, даже если вы укажете, что таймер должен тикать каждую миллисекунду, сообщения будут приходить с интервалом не менее 55 миллисекунд.
Последний параметр (tmprc) определяет адрес функции, которая будет получать сообщения WM_TIMER (мы будем называть эту функцию функцией таймера). Этот параметр необходимо обязательно указать, если первый параметр функции SetTimer равен NULL.
Тип TIMERPROC описан в файле windows.h следующим образом:
typedef void (CALLBACK* TIMERPROC)(HWND hwnd, UINT msg, UINT idTimer, DWORD dwTime);
Сравните это с описанием типа WNDPROC, который используется для знакомой вам функции окна:
typedef LRESULT (CALLBACK* WNDPROC)(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam);

Как видно из описания, функция таймера не возвращает никакого значения, имеет другие (по сравнению с функцией окна) параметры, но описана с тем же ключевым словом CALLBACK
Возвращаемое функцией SetTimer значение является идентификатором созданного таймера (если в качестве первого параметра функции было указано значение NULL). В любом случае функция SetTimer возвращает нулевое значение, если она не смогла создать таймер.
Если приложение больше не нуждается в услугах таймера, оно должно уничтожить таймер, вызвав функцию KillTimer:
BOOL WINAPI KillTimer (HWND hwnd, UINT idTimer);
Первый параметр функции (hwnd) определяет идентификатор окна, указанный при создании таймера функцией SetTimer.
Второй параметр (idTimer) - идентификатор уничтожаемого таймера. Это должен быть либо тот идентификатор, который вы указали при создании таймера (если таймер создавался для окна), либо значение, полученное при создании таймера от функции SetTimer (для таймера, имеющего собственную функцию обработки сообщений).
Функция KillTimer возвращает значение TRUE при успешном уничтожении таймера или FALSE, если она не смогла найти таймер с указанным идентификатором.

Сообщение WM_TIMER

Параметр wParam сообщения WM_TIMER содержит идентификатор таймера, который был указан или получен от функции SetTimer при создании таймера.
С помощью параметра lParam можно определить адрес функции, которая обрабатывает сообщения таймера.
После обработки этого сообщения приложение должно возвратить нулевое значение.
Заметим, что сообщение WM_TIMER является низкоприоритетным. Это означает, что функция DispatchMessage посылает это сообщение приложению только в том случае, если в очереди приложения нет других сообщений.

Подключение таймера к окну

Первый способ работы с таймером - подключение таймера к окну. В этом случае функция окна, к которому подключен таймер, будет получать сообщения от таймера с кодом WM_TIMER. Этот способ самый простой. Вначале вам надо вызывать функцию SetTimer, указав ей в качестве параметров идентификатор окна, идентификатор таймера и период, с которым от таймера должны приходить сообщения:
```
#define FIRST_TIMER 1
int TimerID = SetTimer(hwnd, FIRST_TIMER, 1000, NULL);
```
В данном примере создается таймер с идентификатором FIRST_TIMER, который будет посылать сообщения примерно раз в секунду.
Для уничтожения таймера, созданного этим способом, следует вызвать функцию KillTimer, указав параметры следующим образом: KillTimer(hwnd, FIRST_TIMER);
Для изменения интервала посылки сообщений вам следует вначале уничтожить таймер, а потом создать новый, работающий с другим периодом времени:
KillTimer(hwnd, FIRST_TIMER);

nTimerID = SetTimer(hwnd, FIRST_TIMER, 100, NULL);

Использование функции таймера

Второй способ работы с таймером заключается в использовании для таймера специальной функции, которая будет получать сообщения WM_TIMER. Эта функция является функцией обратного вызова:

void CALLBACK TimerProc(HWND hwnd, UINT msg, UINT idTimer, DWORD dwTime);

Как и для функции окна, для функции таймера можно выбрать любое имя. При создании таймера вам надо указать адрес функции таймера, а имя не имеет никакого значения.

Первый параметр функции таймера - идентификатор окна, с которым связан таймер. Если при создании таймера в качестве идентификатора было указано значение NULL, это же значение будет передано функции таймера.

Второй параметр представляет собой идентификатор сообщения WM_TIMER.

Третий параметр является идентификатором таймера, пославшего сообщение WM_TIMER.

И наконец, последний параметр - текущее время по системным часам компьютера. Это время выражается в количестве тиков таймера с момента запуска Windows. Вы можете узнать текущее системное время в любой момент, если воспользуетесь функцией GetCurrentTime или GetTickCount:

DWORD WINAPI GetCurrentTime(void);

DWORD WINAPI GetTickCount(void);

Эти функции совершенно аналогичны, однако название функции GetTickCount более точно отражает выполняемое ей действие.

Вы можете создать таймер, например, так:

nTimerID = SetTimer(hwnd, 0, 1000, (TIMERPROC)TimerProc);

Для удаления таймера в этом случае необходимо использовать идентификатор, возвращенный функцией SetTimer: KillTimer(hwnd, nTimerID);

5.Работа на уровне сообщений с органами управления программой.

Каждое приложение снабжается, как правило, набором органов управления ходом вычислительного процесса – меню, панелями диалога с пользователем, кнопками и т.д.

Эти органы тоже представляют собой окна и могут создаваться динамически во время выполнения программы или храниться в исполняемом модуле в двоичном формате в виде так называемых ресурсов.

5.1.Органы управления.

Органы управления обычно используют дочерние окна, созданные на базе предопределенных в Windows классов. При регистрации класса окна необходимо указать адрес функции окна, обрабатывающей все сообщения, предназначенные окну. Операционная система Windows регистрирует несколько предопределенных классов окна, обеспечивая для них свои функции окна. Приложение может создавать окна на базе предопределенных классов, при этом им не требуется определять для этих окон функции окна. Эти функции уже определены и находятся внутри Windows.

Например, приложение может создать дочернее окно на базе предопределенного класса "button". Окна этого класса - хорошо знакомые вам кнопки. Внешне поведе-

ние кнопки выглядит достаточно сложным. Когда вы устанавливаете на кнопку курсор мыши и нажимаете левую клавишу мыши, кнопка "уходит вглубь". Когда вы отпускаете клавишу мыши или выводите курсор мыши из области кнопки, кнопка возвращается в исходное положение. Если в диалоговой панели имеется несколько кнопок, вы можете при помощи клавиши <Tab> выбрать нужную, при этом на ней появится пунктирная рамка. Кнопку можно нажимать не только мышью, но и клавиатурой. Кнопка может находиться в активном и пассивном состоянии, причем в последнем случае вы не можете на нее нажать, а надпись, расположенная на кнопке, выводится пунктирным шрифтом. Надпись на кнопке может изменяться в процессе работы приложения.

Словом, поведение такого простого органа управления, как кнопка, при внимательном рассмотрении не выглядит простым. Если бы программист реализовал все функциональные возможности кнопки самостоятельно, это отняло бы у него много времени и сил.

Эта работа облегчается тем, что Windows создана как объектно-ориентированная система. Для таких объектов, как органы управления, в Windows созданы классы окна. Например, в Windows имеется класс окна "button", на базе которого можно создавать кнопки. Функция окна, определенная внутри Windows для класса "button", обеспечивает все необходимые функциональные возможности кнопки. Поэтому программисту достаточно создать собственное дочернее окно на базе класса "button". Это окно сразу становится кнопкой и начинает вести себя как кнопка.

Так как дочерние окна располагаются на поверхности окна-родителя, "прилипая" к ним, приложение может создать в главном или любом другом окне несколько органов управления, которые будет перемещаться сами вместе с окном-родителем. Для этого достаточно просто создать нужные дочерние окна на базе предопределенных классов, указав их размеры, расположение и некоторые другие атрибуты. После этого **приложение может взаимодействовать с органами управления, передавая им или получая от них различные сообщения**.

Для объединения органов управления используются диалоговые панели, представляющие собой временные окна, на поверхности которых располагаются органы управления. Функция этого окна выполняет работу по организации взаимодействия органов управления с приложением и обеспечивает клавиатурный интерфейс.

5.1.1.Кнопки.

Для создания кнопки ваше приложение должно создать дочернее окно на базе предопределенного класса "button". После этого родительское окно будет получать от кнопки сообщение с кодом WM_COMMAND. Этим сообщением кнопка информирует родительское окно о том, что с ней что-то сделали, например, нажали. Вы можете создать кнопку как ресурс с помощью редактора ресурсов – эта процедура очень проста и мы не будем на ней останавливаться.

Создание кнопки динамически.

Для создания кнопки по ходу выполнения программы вам надо вызвать универсальную функцию создания окон CreateWindow.

```
HWND CreateWindow(LPCSTR lpszClassName,  LPCSTR lpszWindowName,
DWORD dwStyle,  int x, int y, int nWidth, int nHeight,  HWND hwndParent, HMENU
hmenu, HINSTANCE hinst,  void FAR* lpvParam);
```

Параметр функции lpszClassName - указатель на строку, содержащую имя класса, на базе которого создается окно. Для создания кнопки необходимо указать имя класса "button". Параметр функции lpszWindowName - указатель на строку, содержащую заголовок окна (Title Bar). Эта строка будет написана на кнопке.
Параметр dwStyle - стиль создаваемого окна. Этот параметр задается как логическая комбинация отдельных битов. Для кнопки следует задать стиль как комбинацию констант WS_CHILD, WS_VISIBLE и константы, определяющей один из возможных стилей кнопки.
Парамеры x и y функции CreateWindow определяют горизонтальную (x) и вертикальную (y) координату кнопки относительно верхнего левого угла родительского окна.
Параметры nWidth и nHeight определяют, соответственно, ширину и высоту создаваемой кнопки.
Параметр hwndParent определяет идентификатор родительского окна, на поверхности которого создается кнопка.
Параметр hmenu - идентификатор меню или идентификатор порожденного (child) окна. Для каждого создаваемого вами дочернего окна вы должны определить собственный идентификатор. Родительское окно будет получать от дочерних окон сообщения. При помощи идентификатора дочернего окна функция родительского окна сможет определить дочернее окно, пославшее сообщение родительскому окну.
Параметр hinst - идентификатор приложения, которое создает окно. Необходимо использовать значение, передаваемое функции WinMain через параметр hInstance.
Последний параметр функции (lpvParam) представляет собой дальний указатель на область данных, определяемых приложением. Этот параметр передается в функцию окна вместе с сообщением WM_CREATE при создании окна. Для кнопки вы должны указать значение NULL.
Для создания кнопки с надписью "Help" в точке с координатами (10, 30) и размерами (40, 20) можно использовать, например, такой вызов функции CreateWindow:

```
hHelpButton = CreateWindow("button", "Help",  WS_CHILD | WS_VISIBLE |
BS_DEFPUSHBUTTON,  10, 30,  40, 20,  hWnd,  (HMENU)IDB_Help,  hInstance,
NULL);
```

Стиль кнопки влияет на ее внешний вид и поведение:

Стиль кнопки	Описание
BS_3STATE	Переключатель, который может находится в одном из трех состояний: включенном (квадратик перечеркнут), выключенном (квадратик не перечеркнут), неактивном (квадратик отображается серым цветом)
BS_AUTO3STATE	Аналогично стилю BS_3STATE, но внешний вид кнопки изменяется автоматически при ее переключении
BS_AUTOCHECKBOX	Переключатель, который может находиться в одном из двух состояний: включенном или выключенном. Внешний вид кнопки изменяется автоматически при ее переключении
BS_AUTORADIOBUTTON	Переключатель, который может находиться в одном из

	двух состояний: включенном (внутри окружности имеется жирная черная точка) или выключенном (окружность не закрашена). Внешний вид кнопки изменяется автоматически при ее переключении
BS_CHECKBOX	Переключатель, который может находиться в одном из двух состояний: включенном или выключенном.
BS_DEFPUSHBUTTON	Стандартная кнопка с толстой рамкой вокруг
BS_GROUPBOX	Прямоугольная область, внутри которой могут находиться другие кнопки. Обычно используется в диалоговых панелях. Этот орган управления не воспринимает сообщения от мыши или клавиатуры
BS_LEFTTEXT	Этот стиль указывается вместе с другими и означает, что текст, расположенный около кнопки, должен находиться слева, а не справа от кнопки
BS_OWNERDRAW	Внешний вид кнопки определяется родительским окном, которое само рисует кнопку во включенном, выключенном или неактивном состоянии
BS_PUSHBUTTON	Стандартная кнопка без рамки
BS_RADIOBUTTON	Переключатель, который может находиться в одном из двух состояний: включенном или выключенном.
BS_USERBUTTON	Устаревший стиль, аналогичный по назначению стилю BS_OWNERDRAW. Не рекомендуется к использованию. Этот стиль не описан в документации SDK для Windows версии 3.1, но определен в файле windows.h

Таким образом, указав функции CreateWindow класс окна "button", мы создаем кнопку. Но с помощью класса "button" можно реализовать несколько перечисленных видов кнопок. Для уточнения вида кнопки мы дополнительно в стиле окна определяем стиль кнопки, указывая константу с префиксом имени BS_. За исключением константы BS_LEFTTEXT, допустимо указывать только одну из перечисленных выше констант. Константа BS_LEFTTEXT используется совместно с другими стилями кнопок, как правило, при создании кнопок в виде переключателей с текстом, расположенным слева от квадратика или кружка переключателя.

Сообщение WM_COMMAND

Сообщение с кодом WM_COMMAND передается функции родительского окна от органа управления, созданного этим окном. При создании органа управления (например, кнопки на базе класса "button") вы вызываете функцию CreateWindow, которой указываете идентификатор родительского окна и идентификатор органа управления.

Если орган управления изменяет свое состояние (например, когда вы нажали на кнопку), функция родительского окна получает сообщение WM_COMMAND. Вместе с этим сообщением функция родительского окна получает в параметре wParam идентификатор органа управления. Младшее слово параметра lParam содержит идентификатор дочернего окна, т. е. идентификатор окна органа управления. Старшее слово содержит код извещения от органа управления (notification code), по ко-

торому можно судить о том, какое действие было выполнено над органом управления.
Когда вы нажимаете на кнопку, родительское окно получает сообщение WM_COMMAND с кодом извещения, равным BN_CLICKED. Получив такое сообщение, приложение определяет, что была нажата кнопка, идентификатор которой находится в параметре wParam.

Управление кнопкой из приложения

У вас есть две возможности управления из приложения Windows кнопкой (или другим органом управления). Во-первых, вы можете, вызывая специальные функции, динамически перемещать орган управления, делать его активным или неактивным, скрывать его или отображать в окне. Во-вторых, кнопке или другому органу управления (как и большинству объектов Windows) можно посылать сообщения, в ответ на которые этот орган будет выполнять различные действия.

Вызов функций управления окном

Для перемещения органа управления внутри окна можно воспользоваться функцией MoveWindow, описанной нами ранее. Функция MoveWindow определяет новое расположение и размеры окна:
BOOL WINAPI MoveWindow(HWND hwnd, int nLeft, int nTop, int nWidth, int nHeight, BOOL fRepaint);
Параметр hwnd указывает идентификатор перемещаемого окна. Для перемещения органа управления вам необходимо указать его идентификатор, полученный от функции CreateWindow.
Параметр nLeft указывает новую координату левой границы окна, параметр nTop - новую координату нижней границы окна. Эти параметры определяют новое положение органа управления в системе координат, связанной с родительским окном. Напомним, что при перемещении обычного перекрывающегося (overlapped) или временного (pop-up) окна используется экранная система координат.
Параметры nWidth и nHeight определяют, соответственно, ширину и высоту окна. Если при перемещении органа управления необходимо сохранить его размеры, укажите значения, использованные при создании этого органа управления.
Последний параметр fRepaint представляет собой флаг, определяющий, надо ли перерисовывать окно после его перемещения. Если значение этого параметра равно TRUE, функция окна после перемещения окна получит сообщение WM_PAINT. Если указать это значение как FALSE, никакая часть окна не будет перерисована. При перемещении органа управления в качестве этого параметра следует указать TRUE.
Иногда возникает необходимость заблокировать орган управления. Например, если в данный момент времени или в данной конфигурации программных или аппаратных средств некоторый режим работы приложения недоступен, имеет смысл заблокировать орган управления, включающий такой режим. Это позволит уберечь пользователя от ошибочного включения режима. Использование недоступного режима может закончиться для него в лучшем случае получением предупреждающего сообщения.

Для блокирования и разблокирования органа управления следует пользоваться функцией EnableWindow: BOOL WINAPI EnableWindow(HWND hWnd, BOOL fEnable);
Функция EnableWindow позволяет разрешать или запрещать поступление сообщений от клавиатуры или мыши в окно или орган управления, идентификатор которого задан параметром hWnd.
Параметр fEnable определяет, будет ли указанное окно заблокировано или наоборот, разблокировано. Для того чтобы заблокировать окно (или орган управления) необходимо для этого парамера указать значение FALSE. Если надо разблокировать окно, используйте значение TRUE.
В любой момент времени приложение может определить, является ли окно или орган управления заблокированным или нет. Для этого следует использовать функцию IsWindowEnabled: BOOL WINAPI IsWindowEnabled(HWND hWnd);
В качестве единственного параметра этой функции надо указать идентификатор проверяемого окна или органа управления. Для заблокированного окна функция возвращает значение FALSE, для разблокированного - TRUE.
Можно вообще убрать орган управления из окна, скрыв его при помощи функции ShowWindow: BOOL ShowWindow(HWND hwnd, int nCmdShow);
Функция отображает окно, идентификатор которого задан параметром hwnd, в нормальном, максимально увеличенном или уменьшенном до пиктограммы виде, в зависимости от значения параметра nCmdShow. Если использовать эту функцию для органа управления, вы можете его скрыть, указав в параметре nCmdShow значение SW_HIDE.
Для восстановления органа управления надо вызвать эту функцию с параметром SW_SHOWNORMAL.
Можно изменить текст, написанный на кнопке. Для этого следует использовать функцию SetWindowText: void WINAPI SetWindowText(HWND hWnd, LPCSTR lpszString);
Эта функция устанавливает новый текст для заголовка окна (или органа управления), идентификатор которого указан при помощи параметра hWnd. Параметр lpszString является дальним указателем на строку символов, закрытую двоичным нулем, которая будет использована в качестве нового заголовка.
И, наконец, вы можете уничтожить созданный вами орган управления. Для этого следует вызвать функцию DestroyWindow: BOOL WINAPI DestroyWindow(HWND hWnd);
Функция DestroyWindow уничтожает окно, идентификатор которого задан в качестве параметра hWnd, и освобождает все связанные с ним ресурсы.
Возвращаемое значение равно TRUE в случае успеха или FALSE при ошибке.

Передача сообщений органу управления

В операционной системе Windows широко используется практика передачи сообщений от одного объекта к другому.
Родительское окно может само послать сообщение кнопке или любому другому органу управления, если оно знает его идентификатор.
Существует два способа передачи сообщений.

Первый способ - запись сообщения в очередь приложения. Он основан на использовании функции PostMessage:
BOOL WINAPI PostMessage(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
Функция PostMessage помещает сообщение в очередь сообщений для окна, указанного параметром hWnd, и сразу возвращает управление. Возвращаемое значение равно TRUE в случае успешной записи сообщения в очередь или FALSE при ошибке. Записанное при помощи функции PostMessage сообщение будет выбрано и обработано в цикле обработки сообщений.
Параметр uMsg задает идентификатор передаваемого сообщения. Параметры wParam и lParam используются для передачи параметров сообщения.
Второй способ - непосредственная передача сообщения функции окна минуя очередь сообщений. Этот способ реализуется функцией SendMessage:
LRESULT WINAPI SendMessage(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
Параметры функции SendMessage используются аналогично параметрам функции PostMessage. Но в отличие от последней функция SendMessage вызывает функцию окна и возвращает управление только после возврата из функции окна.
Возвращаемое функцией SendMessage значение зависит от обработчика сообщения в функции окна.

Сообщения для кнопки

Для управления кнопкой вы можете использовать сообщение BM_SETSTATE, которое позволяет установить кнопку в нажатое или отжатое состояние.
Для установки кнопки в нажатое состояние следует передать ей сообщение BM_SETSTATE с параметром wParam, равным TRUE, и lParam, равным 0:
SendMessage(hButton, BM_SETSTATE, TRUE, 0L);
Для возврата кнопки в исходное состояние передайте ей то же самое сообщение, но с параметром wParam, равным FALSE:
SendMessage(hButton, BM_SETSTATE, FALSE, 0L);
Кнопки – переключатели.
Переключатели – кнопки со стилем BS_RADIOBUTTON и BS_AUTORADIOBUTTON используются аналогично кнопкам переключения диапазонов в радиоприемнике (отсюда и название стиля таких переключателей). Обычно в одной группе располагают несколько таких переключателей, причем включенным может быть только один (ни один радиоприемник не позволит вам принимать передачи сразу в двух диапазонах). Такие переключатели называются переключателями с зависимой фиксацией, так как включение одного переключателя в группе вызывает выключение остальных. Разумеется, ваше приложение должно само обеспечить такой режим работы переключателей.
Переключатели – кнопки со стилями BS_CHECKBOX, BS_AUTOCHECKBOX, BS_3STATE, BS_AUTO3STATE используются как отдельные независимые переключатели. Из них можно сделать переключатель с независимой фиксацией, в котором одновременно могут быть включены несколько переключателей.
Разумеется, все сказанное выше относительно использования переключателей имеет скорее характер рекомендаций, чем обязательное требование. Однако при разработ-

ке приложения вам необходимо позаботиться о том, чтобы интерфейс пользователя соответствовал стандарту, изложенному в руководстве по разработке пользовательского интерфейса (это руководство поставляется в составе SDK и называется The Windows Interface: An Application Design Guide). В этом случае органы управления вашего приложения будут делать то, что ожидает от них пользователь, освоивший работу с другими приложениями Windows.
Обычно переключатели и группы переключателей используются в более сложных органах управления - в диалоговых панелях. Однако ваше приложение может создавать отдельные переключатели в любом своем окне, как и рассмотренные нами ранее кнопки со стилями BS_PUSHBUTTON или BS_DEFPUSHBUTTON.
Вы можете работать с переключателями типа BS_AUTORADIOBUTTON или BS_AUTOCHECKBOX точно таким же образом, что и с кнопками типа BS_PUSHBUTTON или BS_DEFPUSHBUTTON. Когда вы устанавливаете курсор мыши на такой переключатель и нажимаете левую клавишу мыши, состояние переключателя меняется на противоположное. При этом неперечеркнутый квадратик становится перечеркнутым и наоборот, перечеркнутый квадратик становится неперечеркнутым. Состояние переключателя BS_AUTORADIOBUTTON отмечается жирной точкой, которая для включенного переключателя изображается внутри кружочка.
При изменении состояния переключателя родительское окно получает сообщение WM_COMMAND с кодом извещения BN_CLICKED.
Переключатель, имеющий стиль BS_3STATE или BS_AUTO3STATE, внешне похож на переключатель со стилем BS_CHECKBOX, но дополнительно имеет третье состояние. В этом третьем состоянии он изображается серым цветом и может использоваться, например, для индикации недоступного для установки параметра.
Слово "AUTO" в названии стиля переключателя используется для обозначения режима автоматической перерисовки переключателя при изменении его состояния. О чем здесь идет речь?
Когда вы нажимаете кнопку, имеющую стиль BS_PUSHBUTTON или BS_DEFPUSHBUTTON, она автоматически уходит "вглубь", т. е. автоматически перерисовывается в соответствии со своим текущим состоянием. Переключатели BS_CHECKBOX, BS_RADIOBUTTON, а также BS_3STATE не перерисовываются при их переключении. Вы должны их перерисовывать сами, посылая им сообщение BM_SETCHECK: SendMessage(hButton, BM_SETCHECK, 1, 0L);
Параметр wParam сообщения BM_SETCHECK определяет состояние переключателя, которое необходимо установить:

Значение	Описание
0	Установка переключателя в выключенное состояние (прямоугольник не перечеркнут, в кружке нет точки)
1	Установка переключателя во включенное состояние (прямоугольник перечеркнут, в кружке имеется точка)
2	Установка переключателя в неактивное состояние. Это значение используется только для переключателей, имеющих стиль BS_3STATE или BS_AUTO3STATE. При этом переключатель будет изображен серым цветом

Параметр lParam сообщения BM_SETCHECK должен быть равен 0.
В любой момент времени приложение может узнать состояние переключателя, посылая ему сообщение BM_GETCHECK:

```
WORD nState;nState = (WORD) SendMessage(hButton, BM_GETCHECK, 0, 0L);
```

Парамеры wParam и lParam сообщения BM_GETCHECK должны быть равны 0.
Возвращаемое значение, которое будет записано в переменную nState, может быть равно 0 (для выключенного переключателя), 1 (для включенного) или 2 (для переключателя, который находится в неактивном состоянии и отображается серым цветом).

Кнопки и клавиатура

Обычно для работы с кнопками используется мышь. Но, как мы уже говорили, с приложениями Windows вы можете работать и без мыши. В частности, в диалоговых панелях вы можете, нажимая клавишу <Tab>, передавать фокус ввода от одной кнопки к другой. Если кнопка имеет фокус ввода, ее функция окна будет получать сообщения от клавиатуры. Кнопка реагирует только на клавишу пробела - если вы нажмете пробел, когда кнопка имеет фокус ввода, кнопка (или переключатель, который есть ни что иное, как разновидность кнопки) изменит свое состояние.
Для того чтобы ваше приложение могло использовать клавишу <Tab> для передачи фокуса ввода от одного органа управления другому, оно должно создать для клавиши <Tab> свой обработчик сообщения WM_CHAR. Этот обработчик должен установить фокус ввода на следующий (из имеющихся) орган управления, вызвав функцию SetFocus.
К счастью, в Windows имеется объект, специально предназначенный для объединения нескольких органов управления - диалоговая панель. Функции поддержки диалоговых панелей определены внутри Windows. Они выполняют всю работу, необходимую для организации передачи фокуса ввода между различными органами управления, расположенными на диалоговой панели.

5.1.2 Статический орган

Статический орган - это окно, создаваемое на базе предопределенного класса "static". Статический орган нельзя использовать для управления работой приложения, так как он не воспринимает щелчки мыши и не способен обрабатывать сообщения от клавиатуры. Статический орган не посылает родительскому окну сообщение WM_COMMAND.
Когда курсор мыши перемещается над статическим органом управления, Windows посылает функции окна этого органа сообщение WM_NCHITTEST. В ответ на это сообщение статический орган возвращает Windows значение HTTRANSPARENT. В результате Windows посылает сообщение WM_NCHITTEST родительскому окну, лежащему под органом управления. В результате все сообщения от мыши попадают через "прозрачное" окно статического органа управления в родительское окно.
Обычно этот орган управления используется для оформления внешнего вида диалоговых панелей или окон приложения. Задавая различные стили, вы можете создать статический орган управления в виде закрашенного или незакрашенного прямоугольника, а также строки текста. Статические органы управления могут использоваться внутри диалоговых панелей для отображения пиктограмм.

Создание статического органа

Для динамического создания статического органа вы также должны использовать функцию CreateWindow. В качестве первого параметра этой функции следует указать класс окна "static":

```
HWND hStatic;hStatic = CreateWindow("static", NULL,  WS_CHILD | WS_VISIBLE |
SS_BLACKFRAME,  20, 40, 100, 50,  hWnd, (HMENU)-1, hInstance, NULL);
```

Второй параметр определяет текст, который будет расположен внутри органа . Вы можете указать этот параметр как NULL, если текст не используется.
В третьем параметре следует указать один из стилей статического органа . В нашем примере указан стиль SS_BLACKFRAME. Так как статический орган управления не посылает сообщения родительскому окну, в качестве девятого параметра (идентификатор органа управления) можно указать любое число, например, -1.

Стили статического органа

Стили статического органа определяют внешний вид и применение органа.

Прямоугольные рамки

Стили SS_BLACKFRAME, SS_GRAYFRAME и SS_WHITEFRAME предназначены для создания прямоугольных рамок. При помощи этих стилей создаются, соответственно, черные, серые и белые рамки. Внутренняя область рамки остается незакрашенной.
Цвета рамки соответствуют системным цветам, определенным в Windows. Эти цвета можно изменить при помощи стандартного приложения Windows с названием Control Panel. Черный цвет соответствует системному цвету COLOR_WINDOWFRAME, используемому для изображения рамок окон Windows. Белый цвет соответствует цвету COLOR_WINDOW. Это цвет внутренней области окон Windows. И, наконец, серый цвет соответствует цвету фона экрана COLOR_BACKGROUND.
При создании статических органов со стилями SS_BLACKFRAME, SS_GRAYFRAME и SS_WHITEFRAME текст заголовка окна не используется. Соответствующий параметр функции CreateWindow следует указать как NULL.

Закрашенные прямоугольники

Для создания закрашенных прямоугольников используются стили SS_BLACKRECT, SS_GRAYRECT и SS_WHITERECT. Эти стили позволяют создать статические органы управления в виде закрашенных прямоугольников, соответственно, черного, серого и белого цветов (используются системные цвета, как это описано выше).
Для этих стилей текст заголовка окна не используется. Соответствующий параметр функции CreateWindow следует указать как NULL.

Текст

Статические органы удобно использовать для вывода текста. Вы можете использовать пять базовых стилей SS_LEFT, SS_RIGHT, SS_CENTER, SS_LEFTNOWORDWRAP, SS_SIMPLE и один модификатор SS_NOPREFIX.
При использовании стиля SS_LEFT приложение задает размер органа управления, внутри которого будет выведен текст. Орган управления выводит текст (используя для этого функцию DrawText), выравнивая его влево и выполняя свертку слов.

Текст, который не поместился в окне, обрезается. Выполняется замена символов табуляции на пробелы.
Стили SS_RIGHT и SS_CENTER используются аналогично, но текст выравнивается, соответственно, по правой границе органа управления или центрируется.
При использовании стиля SS_LEFTNOWORDWRAP текст выводится без использования свертки слов и выравнивается по левой границе. Часть текста, которая не поместилась в окне, обрезается. Выполняется замена символов табуляции на пробелы.
Стиль SS_SIMPLE похож на стиль SS_LEFTNOWORDWRAP, но вывод текста выполняется быстрее (используется функция TextOut) и замена символов табуляции на пробелы не выполняется. Часть текста, которая не поместилась в окне, обрезается.
При повторном выводе текста содержимое окна не стирается, поэтому новая строка не должна быть короче той, которая была выведена в окно раньше. Этот стиль имеет смысл комбинировать со стилем SS_NOPREFIX. В этом случае для вывода текста используется более быстрая функция ExtTextOut.
Модификатор SS_NOPREFIX используется также в тех случаях, когда необходимо отменить специальную обработку символа "&". Обычно этот символ не выводится статическими органами управления на экран, а следующий за ним символ изображается подчеркнутым (для изображения символа "&" его надо повторить два раза подряд).

Пиктограммы

Стиль SS_ICON используется для изображения пиктограмм в диалоговых панелях.

5.1.3. Полоса просмотра

Полосы просмотра (Scrollbar) широко используются в приложениях Windows для просмотра текста или изображения, которое не помещается в окне. Полосы просмотра бывают горизонтальные или вертикальные. Обычно они располагаются, соответственно, в нижней и правой части окна.
Полоса просмотра представляет собой орган управления, созданный на базе предопределенного класса "scrollbar". Горизонтальная и вертикальная полоса просмотра посылают в функцию родительского окна сообщения WM_HSCROLL и WM_VSCROLL, соответственно. Параметр WParam этих сообщений несет в себе информацию о действии, которое вы выполнили над полосой просмотра.
Полоса просмотра состоит из нескольких объектов, имеющих различное назначение. Если вы устанавливаете курсор мыши на верхнюю кнопку (со стрелкой) полосы просмотра и нажимаете левую клавишу мыши, документ сдвигается в окне вниз на одну строку. Если вы выполняете аналогичную операцию с нижней кнопкой полосы просмотра, документ сдвигается на одну строку вверх. Положение ползунка при этом изменяется соответствующим образом.
Если установить курсор мыши в область полосы просмотра выше ползунка, но ниже верхней кнопки и нажимать на левую клавишу мыши, документ сдвигается вниз на одну страницу. Аналогично, если щелкнуть левой клавишей мыши в области ниже ползунка, но выше нижней кнопки, документ сдвинется на одну страницу вверх. Ползунок при этом устанавливается скачком в новое положение. Дискретность перемещения ползунка задается приложением при инициализации полосы просмотра.

Ползунок можно перемещать мышью вдоль полосы просмотра. При этом, в зависимости от того, как организована работа приложения с полосой просмотра, в процессе перемещения содержимое окна может либо сворачиваться, либо нет. В первом случае синхронно с перемещением ползунка происходит сдвиг документа в окне. Во втором случае после перемещения ползунка документ отображается в позиции, которая зависит от нового положения ползунка.
Понятие "страница" и "строка" больше подходят для текстовых документов. Для свертки других типов документов можно использовать другие термины, однако в любом случае полоса просмотра обеспечивает два типа позиционирования - грубое и точное. Грубое позиционирование выполняется при помощи ползунка или областей полосы просмотра, расположенных между ползунком и кнопками со стрелками, точное позиционирование выполняется кнопками, расположенными на концах полосы просмотра. Следует заметить, что понятия "грубое позиционирование" и "точное позиционирование" отражают факт наличия двух типов позиционирования. Вся логика, обеспечивающая свертку окна, выполняется вашим приложением при обработке сообщений, поступающих от полосы просмотра.
Горизонтальная полоса просмотра состоит из тех же объектов, что и вертикальная. Она обеспечивает свертку документа в горизонтальном направлении.

Создание полосы просмотра

Существует два способа создания полос просмотра в окне приложения.
Первый способ - с помощью функции CreateWindow, указав предопределенный класс окна "scrollbar". Этот способ аналогичен используемому при создании кнопок или статических органов управления.
Второй способ - при создании окна указать, что окно должно иметь горизонтальную, вертикальную или обе полосы просмотра.

Использование класса "scrollbar"

Для создания полосы просмотра с помощью функции CreateWindow вы должны в первом параметре функции указать класс окна "scrollbar":

```
#define IDC_SCROLLBAR 1
HWND hScroll = CreateWindow("scrollbar", NULL,  WS_CHILD | WS_VISIBLE |
BS_HORZ,  20, 40, 100, 50,  hWnd, IDC_SCROLLBAR, hInstance, NULL);
```

Заголовок окна не используется, поэтому второй параметр функции должен быть указан как NULL.
Третий параметр, определяющий стиль окна, наряду с константами WS_CHILD и WS_VISIBLE должен содержать определение стиля полосы просмотра. Существует девять стилей для полосы просмотра. Соответствующие символические константы определены в файле windows.h и имеют префикс имени SBS_ (например, SBS_HORZ).
Девятый параметр функции CreateWindow должен задавать идентификатор полосы просмотра.

Стили полосы просмотра

При создании полосы просмотра функцией CreateWindow вы можете указать в третьем параметре следующие стили.

Стиль	Описание
SBS_BOTTOMALIGN	Создается горизонтальная полоса просмотра, высота которой равна высоте системной полосы просмотра. Выполняется выравнивание нижнего края полосы просмотра по нижнему краю прямоугольника, координаты и размер которого определен при вызове функции CreateWindow. Этот стиль должен использоваться вместе со стилем SBS_HORZ
SBS_HORZ	Создается горизонтальная полоса просмотра. Размер и расположение полосы просмотра определяются при вызове функции CreateWindow
SBS_LEFTALIGN	Создается вертикальная полоса просмотра, ширина которой равна ширина системной полосы просмотра. Левый край полосы просмотра выравнивается по левому краю прямоугольника, координаты и размер которого определен при вызове функции CreateWindow. Этот стиль должен использоваться вместе со стилем SBS_VERT
SBS_RIGHTALIGN	Создается вертикальная полоса просмотра, ширина которой равна ширина системной полосы просмотра. Правый край полосы просмотра выравнивается по правому краю прямоугольника, координаты и размер которого определен при вызове функции CreateWindow. Этот стиль должен использоваться вместе со стилем SBS_VERT
SBS_SIZEBOX	Создается орган управления с небольшим прямоугольником серого цвета (Size Box). Если вы установите курсор мыши внутрь органа управления, нажмете левую клавишу мыши и будете перемещать мышь, родительское окно будет получать сообщения, аналогичные сообщениям от рамки, предназначенной для изменения размера окна.
SBS_SIZEBOXBOTTOMRIGHTALIGN	Аналогично предыдущему, но правый нижний угол прямоугольника выравнивается по правому нижнему углу прямоугольника, координаты и размер которого определен при

	вызове функции CreateWindow. Этот стиль должен использоваться вместе со стилем SBS_SIZEBOX. Для высоты и ширины органа управления используются системные значения
SBS_SIZEBOXTOPLEFTALIGN	Аналогично SBS_SIZEBOX, но верхний левый угол прямоугольника выравнивается по верхнему левому углу прямоугольника, координаты и размер которого определен при вызове функции CreateWindow. Этот стиль должен использоваться вместе со стилем SBS_SIZEBOX. Для высоты и ширины органа управления используются системные значения
SBS_TOPALIGN	Создается горизонтальная полоса просмотра, высота которой равна высоте системной полосы просмотра. Выполняется выравнивание верхнего края полосы просмотра по верхнему краю прямоугольника, координаты и размер которого определен при вызове функции CreateWindow. Этот стиль должен использоваться вместе со стилем SBS_HORZ
SBS_VERT	Создается вертикальная полоса просмотра. Размер и расположение полосы просмотра определяются при вызове функции CreateWindow

Определение полос просмотра при создании окна

Этот способ создания полос просмотра чрезвычайно прост, но с его помощью вы сможете создать только одну вертикальную и одну горизонтальную полосу просмотра, расположенные, соответственно, в правой и нижней части окна.
Для того чтобы у окна появились вертикальная и горизонтальная полосы просмотра, при регистрации класса окна в третьем парамере функции CreateWindow необходимо указать стили окна WS_VSCROLL или WS_HSCROLL (для того чтобы окно имело и вертикальную, и горизонтальную полосы просмотра, следует указать оба стиля):

```
hwnd = CreateWindow(szClassName, szWindowTitle,  WS_OVERLAPPEDWINDOW | WS_VSCROLL | WS_HSCROLL,    CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,    0, 0, hInstance, NULL);
```

Сообщения от полосы просмотра

Все горизонтальные полосы просмотра, определенные для окна (одним из описанных выше способов) посылают в окно сообщение WM_HSCROLL, а все вертикальные - WM_VSCROLL.
Если полоса просмотра была создана первым способом (как орган управления), эти сообщения будет получать функция родительского окна. Если полоса просмотра

была создана вторым способом (определена при создании окна), сообщения от нее будут поступать в функцию окна, имеющего полосы просмотра.
Параметр wParam сообщений полосы просмотра содержит так называемый код полосы просмотра. Этот код соответствует действию, совершенном пользователем над полосой просмотра. Возможны следующие значения (символические константы для них определены в файле windows.h).

Код полосы просмотра	Описание
SB_LEFT, SB_TOP (используются одинаковые значения констант для разных символических имен)	Сдвиг влево в начало документа (горизонтальная полоса просмотра), сдвиг вверх в начало документа (вертикальная полоса просмотра)
SB_LINELEFT, SB_LINEUP	Сдвиг влево на одну строку, сдвиг вверх на одну строку
SB_LINERIGHT, SB_LINEDOWN	Сдвиг вправо на одну строку, сдвиг вниз на одну строку
SB_PAGELEFT, SB_PAGEUP	Сдвиг на одну страницу влево, сдвиг на одну страницу вверх
SB_PAGERIGHT, SB_PAGEDOWN	Сдвиг на одну страницу вправо, сдвиг на одну страницу вниз
SB_RIGHT, SB_BOTTOM	Сдвиг вправо в конец документа, сдвиг вниз в конец документа
SB_THUMBPOSITION	Сдвиг в абсолютную позицию. Текущая позиция определяется младшим словом параметра lParam
SB_ENDSCROLL	Сообщение приходит в тот момент, когда вы отпускаете клавишу мыши после работы с полосой просмотра. Это сообщение обычно игнорируется (передается функции DefWindowProc)
SB_THUMBTRACK	Перемещение ползунка полосы просмотра. Текущая позиция определяется младшим словом параметра lParam

В ответ на сообщения полосы просмотра соответствующая функция окна должна вернуть нулевое значение.
Для сообщений SB_THUMBTRACK и SB_THUMBPOSITION младшее слово параметра lParam определяет текущую позицию ползунка на полосе просмотра. Для других сообщений полосы просмотра младшее слово параметра lParam не используется. Старшее слово параметра lParam содержит идентификатор окна для полосы просмотра (если временное окно имеет полосу просмотра, старшее слово параметра lParam не используется).
Несмотря на то, что в файле windows.h определены константы SB_LEFT, SB_TOP, SB_RIGHT, SB_BOTTOM, полоса просмотра никогда не посылает сообщений со значением параметра wParam, равным этим константам. Однако приложению имеет

смысл предусмотреть обработчик для таких сообщений. Это нужно для подключения к полосе просмотра клавиатурного интерфейса.
Если запустить любое стандартное приложение Windows, работающее с текстовыми документами, можно убедится в том, что для свертки окна, отображающего документ, можно использовать не только полосу просмотра, но и клавиши. Обычно это клавиши перемещения курсора и клавиши <PgUp>, <PgDn>. Как правило, с помощью клавиш <Home> и <End> вы можете перейти, соответственно, в начало и в конец документа.
Так как действия, выполняемые при свертке, одинаковы для полосы просмотра и дублирующих ее клавиш, имеет смысл предусмотреть единый обработчик сообщений от полосы просмотра. Для добавления клавиатурного интерфейса обработчик клавиатурного сообщения WM_KEYDOWN может посылать в функцию окна сообщения полосы просмотра. Например, если обработчик сообщения WM_KEYDOWN обнаружил, что вы нажали клавишу <PgUp>, он может послать в функцию окна сообщение WM_VSCROLL со значением wParam, равным SB_PAGEUP. Результат будет в точности такой же, как будто для свертки документа на одну страницу вверх вы воспользовались полосой просмотра, а не клавиатурой.
Если же обработчик клавиатурного сообщения WM_KEYDOWN обнаружил, что вы нажали клавишу <Home> или <End>, он может послать в функцию окна, соответственно, сообщение WM_VSCROLL со значением wParam, равным SB_TOP или SB_BOTTOM. Если в приложении имеются обработчики этих сообщений, они выполнят переход в начало или в конец документа.
Такой подход позволяет локализовать всю логику свертки в обработчике сообщений полосы просмотра. При этом сильно упрощается процедура подключения клавиатурного интерфейса - обработчик клавиатурного сообщения WM_KEYDOWN должен послать в функцию окна сообщение полосы просмотра, соответствующее коду нажатой клавиши. Но ему не надо выполнять свертку окна.

Инициализация полосы просмотра

Для полосы просмотра определены понятия текущая позиция и диапазон изменения значений позиции. При передвижении ползунка вдоль полосы просмотра текущая позиция принимает дискретные значения внутри диапазона изменения значений позиции. Если ползунок находится в самом левом (для горизонтальной полосы просмотра) или самом верхнем (для вертикальной полосы просмотра) положении, текущая позиция равна минимальной. Если же ползунок находится в самом правом или самом нижнем положении, текущая позиция равна максимальной.
После того как вы создали полосу просмотра одним из описанных выше способов, ее необходимо проинициализировать, указав диапазон изменений значений позиции. Для этого следует вызвать функцию SetScrollRange:
void WINAPI SetScrollRange(HWND wnd, int fnBar, int nMin, int nMax, BOOL fRedraw);
Параметр hwnd определяет идентификатор окна, имеющего полосу просмотра, или идентификатор полосы просмотра, созданного как орган управления.
Параметр fnBar определяет тип полосы просмотра, для которой выполняется установка диапазона изменения значений позиции:

Значение	Описание

SB_CTL	Установка диапазона для полосы просмотра, созданной как орган управления класса "scrollbar". В этом случае параметр hwnd функции SetScrollRange должен указывать идентификатор органа управления, полученный при его создании от функции CreateWindow.
SB_HORZ	Установка диапазона горизонтальной полосы просмотра для окна, при создании которого был использован стиль окна WS_HSCROLL. Параметр hwnd функции SetScrollRange должен указывать идентификатор окна, имеющего полосу просмотра
SB_VERT	Установка диапазона вертикальной полосы просмотра для окна, при создании которого был использован стиль окна WS_VSCROLL. Параметр hwnd функции SetScrollRange должен указывать идентификатор окна, имеющего полосу просмотра

Параметры nMin и nMax определяют, соответственно, минимальное и максимальное значение для диапазона. Разность между nMax и nMin не должна превышать 32666.
Параметр fRedraw определяет, следует ли перерисовывать полосу просмотра для отражения изменений. Если указано значение TRUE, после установки диапазона полоса просмотра будет перерисована, а если FALSE - не будет.
В любой момент времени вы можете узнать диапазон для полосы просмотра, вызвав функцию GetScrollRange:
void WINAPI GetScrollRange(HWND hwnd, int fnBar, int FAR* lpnMinPos, int FAR* lpnMaxPos);
Параметры hwnd и fnBar аналогичны параметрам функции SetScrollRange. Первый из них определяет идентификатор окна или органа управления, второй - тип полосы просмотра.
Параметры lpnMinPos и lpnMaxPos - указатели на переменные, в которые после возврата из функции будут записаны, соответственно, минимальное и максимальное значение диапазона.

Управление полосой просмотра

Для установки ползунка в заданную позицию следует использовать функцию SetScrollPos: int WINAPI SetScrollPos(HWND hwnd, int fnBar, int nPos, BOOL fRepaint);
Параметры hwnd и fnBar определяют, соответственно, идентификатор окна или органа управления и тип полосы просмотра.
Параметр nPos определяет новое положение ползунка. Значение этого параметра должно находиться в пределах установленного диапазона.
Параметр fRepaint определяет, нужно ли перерисовывать полосу просмотра после установки новой позиции. Если указано TRUE, полоса будет перерисована, если FALSE - нет.
Функция SetScrollPos возвращает значение предыдущей позиции или 0 в случае ошибки. Для определения текущей позиции надо вызвать функцию GetScrollPos:
int WINAPI GetScrollPos(HWND hwnd, int fnBar);
Параметры этой функции определяют, соответственно, идентификатор окна или органа управления и тип полосы просмотра

Функция возвращает текущую позицию или 0, если идентификатор окна указан неправильно или окно не имеет полосы просмотра.
Иногда бывает нужно убрать из окна одну или обе полосы просмотра. Это нужно, например, в тех случаях, когда, например, после изменения размера окна документ поместился в нем целиком. С помощью функции ShowScrollBar вы можете скрывать или показывать полосы просмотра:
void WINAPI ShowScrollBar(HWND hwnd, int fnBar, BOOL fShow);
Параметр hwnd определяет идентификатор окна, имеющего полосу просмотра, или идентификатор полосы просмотра, созданного как орган управления.
Параметр fnBar определяет тип полосы просмотра, для которой выполняется установка диапазона изменения значений позиции. Кроме описанных нами ранее констант SB_CTL, SB_HORZ и SB_VERT вы можете использовать константу SB_BOTH. Эта константа предназначена для работы сразу с обеими полосами просмотра, определенными в стиле окна.
Параметр fShow определяет действие, выполняемое функцией. Если этот параметр равен TRUE, полоса просмотра (или обе полосы просмотра, если указано SB_BOTH) появляются в окне. Если же указать значение FALSE, полоса просмотра исчезнет.
Программный интерфейс операционной системы Windows версии 3.1 имеет в своем составе функцию EnableScrollBar, позволяющую разрешать или запрещать работу полосы просмотра:
BOOL WINAPI EnableScrollBar(HWND hwnd, int fnBar, UINT fuArrowFlag);
Первые два парамера этой функции аналогичны параметрам функции ShowScrollBar.
Параметр fuArrowFlag определяет, какие из кнопок полосы просмотра должны быть заблокированы или разблокированы:

Значение	Описание
ESB_ENABLE_BOTH	Обе кнопки полосы просмотра разблокированы
ESB_DISABLE_BOTH	Обе кнопки полосы просмотра заблокированы
ESB_DISABLE_LEFT, ESB_DISABLE_UP, ESB_DISABLE_LTUP	Заблокирована левая кнопка горизонтальной полосы просмотра или верхняя кнопка вертикальной полосы просмотра
ESB_DISABLE_RIGHT, ESB_DISABLE_DOWN, ESB_DISABLE_RTDN	Заблокирована правая кнопка горизонтальной полосы просмотра или нижняя кнопка вертикальной полосы просмотра

Функция возвращает значение TRUE при успешном завершении или FALSE при ошибке (если, например, кнопки уже находятся в требуемом состоянии).

5.1.4. Редактор текста

В операционной системе Microsoft Windows зарегистрирован класс окна с именем "edit", на базе которого вы можете создать однострочный или многострочный текстовый редактор. Такой редактор может быть использован для ввода значений текстовых или числовых переменных, а также для создания и редактирования текстовых файлов (без функций форматирования текста). Встроенный текстовый редактор умеет выполнять функции выделения текста, может работать с универсальным буфером обмена Clipboard.

Для того чтобы в среде Windows сделать свой собственный текстовый редактор, вам достаточно создать на базе класса "edit" орган управления, вызвав функцию CreateWindow. После этого функция родительского окна будет получать от редактора сообщение с кодом WM_COMMAND (как и от других аналогичных органов управления). Вместе с этим сообщением в функцию окна будут передаваться коды извещения, отражающие изменения состояния редактора текста. Вы также можете с помощью функции SendMessage посылать текстовому редактору около трех десятков различных управляющих сообщений, с помощью которых можно изменять редактируемый текст, получать отдельные строки, выделять фрагменты текста, копировать выделенный фрагмент текста в Clipboard и т. д.
Использование предопределенного класса "edit" - самый простой способ создания в приложении редактора текста. Фактически в этом случае вы будете использовать готовый текстовый редактор. Вам не придется самостоятельно обрабатывать сообщения от клавиатуры, управлять текстовым курсором, учитывать ширину букв в пропорциональном шрифте, выполнять свертку текста в окне редактирования по вертикали или горизонтали, заниматься выделением фрагментов текста, работать с Clipboard или решать другие задачи, возникающие при создании текстовых редакторов.

Создание редактора текста

Для создания редактора текста (однострочного или многострочного) следует вызвать функцию CreateWindow, передав ей в качестве первого параметра указатель на строку "edit":

```
hwndEdit = CreateWindow("edit", NULL, WS_CHILD | WS_VISIBLE | S_BORDER |
ES_LEFT,        30, 30, 300, 30, hwnd, (HMENU) ID_EDIT, hInst, NULL);
```

Заголовок окна не используется, поэтому второй параметр следует указать как NULL.
Если при создании текстового редактора не указать стиль окна WS_BORDER, область редактирования не будет выделена. Это неудобно для пользователя, особенно если в окне имеется несколько редакторов. При использовании стиля WS_BORDER вокруг редактора будет нарисована рамка.
Кроме обычных стилей окна для текстового редактора указывают стили, символическое имя которых начинается с префикса ES_. Это стили редактора текста. Они влияют на внешний вид редактора и выполняемые им функции. Подробно стили редактора текста будут описаны в следующем разделе.
Остальные параметры функции CreateWindow указываются так же, как и для других органов управления. Параметры с четвертого по седьмой используются для определения расположения и размеров текстового редактора. Восьмой параметр - идентификатор родительского окна, в функцию которого будет поступать сообщение WM_COMMAND. Девятый параметр определяет идентификатор редактора текста. Десятый указывает идентификатор копии приложения. Последний параметр должен быть задан как NULL.

Стили редактора текста

Приведем список стилей окна, которые используются при создании редактора текста.

Стиль	Описание
ES_AUTOHSCROLL	Выполняется автоматическая свертка текста по горизонтали. Когда при наборе текста достигается правая граница окна ввода, весь текст сдвигается влево на 10 символов
ES_AUTOVSCROLL	Выполняется автоматическая свертка текста по вертикали. Когда при наборе текста достигается нижняя граница окна ввода, весь текст сдвигается вверх на одну строку
ES_CENTER	Центровка строк по горизонтали в многострочном текстовом редакторе
ES_LEFT	Выравнивание текста по левой границе окна редактирования
ES_LOWERCASE	Выполняется автоматическое преобразование введенных символов в строчные (маленькие)
ES_MULTILINE	Создается многострочный редактор текста
ES_NOHIDESEL	Если редактор текста теряет фокус ввода, при использовании данного стиля выделенный ранее фрагмент текста отображается в инверсном цвете. Если этот стиль не указан, при потере фокуса ввода выделение фрагмента пропадает и появляется вновь только тогда, когда редактор текста вновь получает фокус ввода
ES_OEMCONVERT	Выполняется автоматическое преобразование кодировки введенных символов из ANSI в OEM и обратно. Обычно используется для ввода имен файлов
ES_PASSWORD	Этот стиль используется для ввода паролей или аналогичной информации. Вместо вводимых символов отображается символ "*" или другой, указанный при помощи сообщения EM_SETPASSWORDCHAR (см. ниже раздел, посвященный сообщениям для редактора текста)
ES_READONLY	Создаваемый орган управления предназначен только для просмотра текста, но не для редактирования. Этот стиль можно использовать в версии 3.1 операционной системы Windows или в более поздней версии
ES_RIGHT	Выравнивание текста по правой границе окна редактирования
ES_UPPERCASE	Выполняется автоматическое преобразование введенных символов в заглавные (большие)
ES_WANTRETURN	Стиль используется в комбинации со стилем ES_MULTILINE. Используется только в диалоговых панелях. При использовании этого стиля клавиша <Enter> действует аналогично кнопке диалоговой панели, выбранной по умолчанию. Этот стиль можно использовать в версии 3.1 операционной системы Windows или в более поздней версии

Для создания однострочного редактора текста достаточно указать стиль ES_LEFT (который, кстати, определен в файле windows.h как 0). Для обеспечения свертки текста по горизонтали используйте дополнительно стиль ES_AUTOHSCROLL.

Если вам нужен многострочный редактор текста, укажите стиль ES_MULTILINE. Для обеспечения автоматической свертки текста по горизонтали и вертикали следует также указать стили ES_AUTOHSCROLL и ES_AUTOVSCROLL.
Если в многострочном редакторе текста не указан стиль ES_AUTOHSCROLL, но указан стиль ES_AUTOVSCROLL, при достижении в процессе ввода текста правой границы окна ввода выполняется автоматический перенос слова на новую строку. Если свертка не используется, в описанной выше ситуации будет выдан звуковой сигнал.
Многострочный редактор текста может иметь вертикальную и горизонтальную полосы просмотра. Для создания полос просмотра достаточно в стиле редактора указать константы WS_HSCROLL и WS_VSCROLL.

Коды извещения

Текстовый редактор посылает в родительское окно сообщение WM_COMMAND с параметром wParam, равным идентификатору редактора. Этот идентификатор можно использовать для того чтобы различать сообщения, поступающие от разных органов управления (в частности, от разных текстовых редакторов, если в одном окне их создано несколько штук).
Младшее слово параметра lParam содержит идентификатор окна, полученный от функции CreateWindow при создании редактора.
Старшее слово параметра lParam содержит код извещения. Анализируя этот код, приложение может определить событие, послужившее причиной появления сообщения WM_COMMAND.
Приведем список кодов извещений.

Код извещения	Описание
EN_CHANGE	Изменилось содержимое текста в окне редактирования
EN_ERRSPACE	Произошла ошибка при попытке получить дополнительную память
EN_HSCROLL	Выполнена свертка текста по горизонтали. Пользователь использовал горизонтальную полосу просмотра для свертки текста, но изменения в окне редактирования еще не произошли
EN_KILLFOCUS	Текстовый редактор потерял фокус ввода
EN_MAXTEXT	При вводе очередного символа произошло переполнение, так как было превышен максимально допустимый для редактора размер текста
EN_SETFOCUS	Текстовый редактор получил фокус ввода
EN_UPDATE	Содержимое текстового редактора будет изменено. Пользователь ввел один символ текста или выполнил другую операцию редактирования, но выполнение этой операции еще не отразилось на содержимом окна редактирования. После этого извещения после отображения изменений придет извещение с кодом EN_CHANGE
EN_VSCROLL	Выполнена свертка текста по вертикали. Пользователь использовал вертикальную полосу просмотра для свертки текста, но изменения в окне редактироания еще не произошли

Ваше приложение должно обрабатывать, по крайней мере, извещение с кодом EN_ERRSPACE, которое приходит в том случае, если редактор текста не смог заказать для себя дополнительную память.

Сообщения для редактора текста

С помощью функции SendMessage вы можете посылать в редактор текста различные сообщения. Коды сообщений, специально предназначенных для текстового редактора, имеют символические имена с префиксом EM_. Приведем список таких сообщений.

EM_CANUNDO С помощью этого сообщения можно проверить, поддерживает ли редактор текста операцию отмены последнего действия редактирования. Эта операция выполняется по сообщению WM_UNDO, когда оно посылается в редактор текста.

Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: TRUE, если операция поддерживается, FALSE - если нет

EM_EMPTYUNDOBUFFER

Сброс содержимого буфера, используемого для отмены последнего действия редактирования.Параметры и возвращаемое значение не используются.

EM_FMTLINES

Управление режимом добавления или удаления символов конца строки в процессе переноса слов на новую строку.
Параметры: wParam = (WPARAM)(BOOL)fAddEOL; lParam = 0L;
Значение флага fAddEOL: TRUE - вставка, FALSE - удаление.
Возвращаемое значение:TRUE - вставка, FALSE - удаление

EM_GETFIRSTVISIBLELINE

Получение номера самой верхней видимой строки в окне редактирования.
Параметры: не используются. Возвращаемое значение: Номер строки. Первой строке соответствует значение 0

EM_GETHANDLE

Получение идентификатора локальной памяти, используемой редактором для хранения текста.Параметры: не используются.
Возвращаемое значение: Идентификатор блока памяти

EM_GETLINE

Копирование строки из редактора текста в буфер.
Параметры: wParam = (WPARAM)nLine; lParam = (LPARAM)(LPSTR)lpCh;
nLine - номер строки, lpCh - адрес буфера для строки.
Возвращаемое значение: Номер скопированных в буфер байт данных или 0, если указанный номер строки превосходит количество строк в тексте

EM_GETLINECOUNT

Определение количества строк в тексте.
Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: Количество строк текста или 1, если окно редактирования не содержит ни одной строки текста

EM_GETMODIFY

Определение значения флага обновления.
Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: TRUE, если текст был изменен или FALSE - если нет

EM_GETPASSWORDCHAR

Получение символа, используемого для вывода при вводе пароля.
Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: Код символа

EM_GETRECT

Определение координат прямоугольной области, используемой для редактирования текста. Эта область по своим размерам не обязательно совпадает с областью, занятой самим органом управления.
Параметры: wParam = 0; lParam = (LPARAM)(RECT FAR *)lprc;
lprc - указатель на структуру RECT, в которую будут записаны искомые координаты.
Возвращаемое значение: не используется

EM_GETSEL

Определение положения первого и последнего символа в выделенном фрагменте .
Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: Двойное слово. Младшее слово содержит положение первого символа в выделенном фрагменте, старшее - положение символа, следующего за выделенным фрагментом текста

EM_GETWORDBREAKPROC

Получение адреса текущей функции, которая используется для переноса слов с одной строки на другую. Используется в Windows версии 3.1 и более поздних версиях.
Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: Адрес функции или NULL, если такая функция не существует

EM_LIMITTEXT

Определение максимального количества символов, которое можно ввести в окно .
Параметры: wParam = (WPARAM)cCmax;lParam = 0L; cCMax - размер текста.
Возвращаемое значение: не используется

EM_LINEFROMCHAR

Определение номера строки, содержащей символ в заданной позиции.
Параметры: wParam = (WPARAM) iChar; lParam = 0L;
iChar - номер позиции. Можно задать как -1, в этом случае используется текущая строка (строка, в которой установлен текстовый курсор), или строка, в которой начинается выделенный фрагмент текста (если в тексте есть выделенный фрагмент).
Возвращаемое значение: Номер строки. Первой строке соответствует значение 0

EM_LINEINDEX

Определение смещения в байтах от начала текста заданной строки.
Параметры: wParam = (WPARAM) nLine; lParam = 0L;

nLine - номер строки. Можно задать как -1, в этом случае используется текущая строка.
Возвращаемое значение: Смещение в байтах или -1, если указана строка с номером, превосходящим количество строк в окне редактирования

EM_LINELENGTH

Определение размера строки в байтах.
Параметры: wParam = (WPARAM) iChar; lParam = 0L;
iChar - номер позиции символа, который находится в строке. Можно задать как -1, в этом случае используется текущая строка, для которой возвращается количество невыбранных символов
Возвращаемое значение: Размер строки в байтах

EM_LINESCROLL

Свертка заданного количества строк.
Параметры: wParam = 0; lParam = MAKELPARAM(dv, dh);
dv - количество сворачиваемых строк по вертикали,dh - количество символов для свертки по горизонтали, не используется для текста, выравненного по правой границе или центрированного.
Возвращаемое значение: TRUE, если сообщение был послано многострочному редактору, или FALSE - если однострочному

EM_REPLACESEL

Заменить выделенный фрагмент текста. Если в тексте нет выделенных фрагментов, строка будет вставлена в текущей позиции.
Параметры: wParam = 0; lParam = (LPARAM)(LPCSTR)lpszStr
;lpszStr - адрес строки, которая должна заместить собой выделенный текст
Возвращаемое значение: не используется

EM_SETHANDLE

Назначение буфера для хранения текста.
Параметры: wParam = (WPARAM)(HLOCAL)hLoc; lParam = 0L;
hLoc - идентификатор локального блока памяти, полученный с помощью функции LocalAlloc. Возвращаемое значение: не используется

EM_SETMODIFY

Установка флага обновления.
Параметры:wParam = (WPARAM)(UINT)fMod; lParam = 0L;
fMod - новое значение для флага обновления. TRUE, если текст надо отметить, как обновленный, FALSE - если как необновленный.
Возвращаемое значение: не используется

EM_SETPASSWORDCHAR

Установка символа, который используется для вывода текста (в редакторе, имеющим стиль ES_PASSWORD).
Параметры: wParam = (WPARAM)(UINT)chChar; lParam = 0L;
chChar - код символа.
Возвращаемое значение: TRUE, если сообщение посылается редактору

EM_SETREADONLY

Установка или сброс состояния редактора, в котором пользователю позволяется только просматривать текст, но не редактировать (режим "только чтение").
Параметры: wParam = (WPARAM)(UINT)fReadOnly; lParam = 0L;
fReadOnly - TRUE для установки режима "только чтение", FALSE - для сброса.
Возвращаемое значение: TRUE, если установка выполнена без ошибок или FALSE при ошибке

EM_SETRECT

Изменение размеров или расположения области, используемой для редактирования текста. Эта область находится внутри окна органа управления и сразу после создания совпадает с этим окном по размерам и расположению.
Параметры: wParam = 0; lParam = (LPARAM)(RECT FAR *)lprc;
lprc - указатель на структуру RECT, в которую будут записаны новые координаты области.
Возвращаемое значение: не используется

EM_SETRECTNP

Аналогично предыдущему, за исключением того что окно редактирования не перерисовывается

EM_SETSEL

Выделение заданных символов в окне редактирования.
Параметры: wParam = (WPARAM)(UINT)fScroll; lParam = MAKELPARAM(ichStart, ichEnd);
fScroll - если этот параметр равен 1, текстовый курсор сворачивается, если 0 - нет.
ichStart - начальная позиция.
ichEnd - конечная позиция. Если начальная позиция равна 0, а конечная -1, выбирается весь текст. Если начальная позиция равна -1, выделение фрагмента (если оно было) исчезает.
Возвращаемое значение: TRUE, если сообщение посылается редактору, созданному как орган управления

EM_SETTABSTOPS

Установка позиций табуляции.
Параметры: wParam = (WPARAM)cTabs; lParam = (LPARAM)(const int FAR *) lpTabs;
cTabs - расстояние для табуляции. Если этот параметр указан как 0, используется значение по умолчанию - 32.
lpTabs - массив беззнаковых целых чисел, определяющих расположение позиций табуляции в единицах, используемых для диалоговых панелей. Эти единицы будут описаны в следующей главе.
Возвращаемое значение: TRUE, если позиции табуляции были установлены ил FALSE при ошибке

EM_SETWORDBREAKPROC

Установка новой функции для переноса слов с одной строки на другую. Используется в Windows версии 3.1 и более поздних версиях.

Параметры: wParam = 0; lParam = (LPARAM)(EDITWORDBREAKPROC) ewpbrc; ewpbrc - адрес переходника для новой функции, которая будет использована для переноса слов. Этот адрес необходимо получить при помощи функции MakeProcInstance, указав последней адрес функции переноса слов.
Возвращаемое значение: не используется

EM_UNDO

Отмена последней операции редактирования текста.
Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: не используется
Помимо описанных выше, текстовому редактору можно посылать некоторые сообщения, символические имена которых начинаются с префикса WM_. Это сообщения WM_COPY, WM_PASTE, WM_CUT, WM_CLEAR. Приведем краткое описание этих сообщений.

WM_COPY

Копирование выделенного фрагмента текста в универсальный буфер обмена Clipboard.
Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: не используется

WM_PASTE

Вставка текста из буфера обмена Clipboard в текущую позицию редактируемого текста.
Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: не используется

WM_CUT

Удаление выделенного текста с записью его в Clipboard. Удаленный текст можно восстановить, если послать в редактор сообщение EM_UNDO.
Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: не используется

WM_CLEAR

Удаление выделенного текста без записи в Clipboard. Удаленный текст можно восстановить, если послать в редактор сообщение EM_UNDO.
Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: не используется

1.1..5. Список класса LISTBOX

Перед программистом часто встает задача организации списка, предназначенного для выбора строки из некоторого определенного заранее набора строк. Например, вам может потребоваться список файлов из текущего каталога, список названий цветов для раскраски какого-либо объекта приложения, список режимов работы приложения и т. д. Стандартные диалоговый панели "Open" и "Save As" содержат списки файлов, каталогов и дисковых устройств.
Операционная система Windows имеет мощное средство организации списков - органы управления класса "listbox" и "combobox". В этом разделе мы рассмотрим спи-

сок, созданный на базе класса "listbox". О том, как создавать и использовать список класса "combobox", вы узнаете из раздела с названием "Список класса COMBOBOX".
С помощью класса "listbox" вы можете создавать одноколоночные и многоколоночные списки, имеющие вертикальную (для одноколоночных списков) и горизонтальную (для многоколоночных списков) полосу просмотра. Родительское окно может само рисовать элементы списка, аналогично тому, как оно рисует кнопки.

Создание списка

Для создания списка приложение должно вызвать функцию CreateWindow, передав в качестве первого параметра указатель на строку "listbox":

```
hListBox = reateWindow("listbox", NULL,   WS_CHILD | WS_VISIBLE |
LBS_STANDARD |   LBS_WANTKEYBOARDINPUT,   30, 30, 200, 100,   hwnd,
(HMENU) ID_LIST, hInst, NULL);
```

Второй параметр функции должен быть указан как NULL.
Дополнительно к стилям окна WS_CHILD и WS_VISIBLE при создании списка указываются специальные стили списка, символические имена которых имеют префикс LBS_.
Остальные параметры функции CreateWindow указываются так же, как и для других органов управления. Параметры с четвертого по седьмой используются для определения расположения и размеров списка. Восьмой параметр - идентификатор родительского окна, в функцию которого будет поступать сообщение WM_COMMAND. Девятый параметр определяет идентификатор списка. Десятый указывает идентификатор копии приложения. Последний параметр должен быть задан как NULL.

Стили списка

Имя стиля	Описание
LBS_DISABLENOSCROLL	Если в одноколоночном списке помещаются все строки, вертикальная полоса просмотра изображается в неактивном состоянии. Без указания стиля LBS_DISABLENOSCROLL в аналогичной ситуации вертикальная полоса просмотра пропадает. Этот стиль можно указывать для Windows версии 3.1 и более поздних версий
LBS_EXTENDEDSEL	Можно выделять не только отдельные строки, но и сразу несколько расположенных рядом строк. Для этого можно использовать клавишу <Shift> или мышь
LBS_HASSTRINGS	Создание списка, содержащего строки. Этот стиль используется для всех списков, за исключением тех, которые рисуются родительским окном
LBS_MULTICOLUMN	Создание многоколоночного списка. Для того чтобы задать количество колонок, в список необходимо послать сообщение LB_SETCOLUMNWIDTH
LBS_MULTIPLESEL	Можно выделять в списке несколько строк сразу.

	Выделенные строки могут находиться в любом месте списка, а не только рядом (как при использовании стиля LBS_EXTENDEDSEL)
LBS_NOINTEGRALHEIGHT	Допустимо частичное отображение строк (например, в нижней части списка можно отображать верхнюю половину строки)
LBS_NOREDRAW	Для списка не выполняется перерисовка содержимого при добавлении или удалении строк. Этот стиль может быть динамически добавлен или удален посылкой списку сообщения WM_SETREDRAW
LBS_NOTIFY	Родительское окно, создавшее список, получит извещение, если пользователь выполнит в списке двойной щелчок мышью по строке
LBS_OWNERDRAWFIXED	Создается список, который рисуется родительским окном, причем все элементы в списке имеют одинаковую высоту
LBS_OWNERDRAWVARIABLE	Аналогично предыдущему, но элементы списка могут иметь разную высоту
LBS_SORT	Строки списка будут отсортированы
LBS_STANDARD	Комбинация наиболее употребительных стилей списка: LBS_NOTIFY, LBS_SORT, WS_BORDER и WS_VSCROLL
LBS_USETABSTOPS	При выводе строк списка будет выполняться преобразование символов табуляции. По умолчанию один символ табуляции расширяется на 32 единицы ширины (эти единицы используются в диалоговых панелях)
LBS_WANTKEYBOARDINPUT	При использовании этого стиля родительское окно, создавшее список, будет получать сообщения WM_VKEYTOITEM или WM_CHARTOITEM, если список имеет фокус ввода и пользователь работает со списком при помощи клавиатуры

Для создания простейшего одноколоночного списка имеет смысл использовать стиль LBS_STANDARD. В этом случае, если все строки списка не помещаются в окне, у списка появится вертикальная полоса просмотра. Если при удалении из списка некоторого количества строк размеры окна списка станут достаточны для одновременного отображения всех строк, полоса просмотра исчезнет.
В некоторых случаях такое поведение списка нежелательно, так как оно приводит к изменению внешнего вида списка. Если для списка указать стиль LBS_DISABLENOSCROLL, полоса просмотра будет существовать всегда. Если все строки списка помещаются в окне, эта полоса перейдет в неактивное состояние, но останется на экране.

Иногда нужно сделать так, чтобы добавляемые в список строки отображались в порядке добавления, а не в алфавитном порядке. Для этого не следует указывать стиль LBS_SORT.
Небольшое замечание относительно использования стиля LBS_WANTKEYBOARDINPUT. Если указан этот стиль, то сообщения WM_KEYDOWN и WM_CHAR, получаемые списком (имеющим фокус ввода), создают сообщения WM_VKEYTOITEM или WM_CHARTOITEM. Эти сообщения попадают в функцию родительского окна, благодаря чему последнее может отслеживать операции, выполняемые пользователем над списком при помощи клавиатуры.
Если список имеет стиль LBS_HASSTRINGS, родительское окно будет получать сообщение WM_VKEYTOITEM, а если не имеет - сообщение WM_CHARTOITEM. Параметр wParam сообщения WM_VKEYTOITEM содержит виртуальный код нажатой клавиши. Например, если пользователь выделит строку в списке и нажмет клавишу <Enter>, родительское окно получит сообщение WM_VKEYTOITEM со значением параметра wParam, равным VK_RETURN. При этом оно может получить из списка выбранную строку и выполнить над ней необходимые действия.
Если родительское окно получает сообщение WM_CHARTOITEM, параметр wParam содержит код символа, соответствующего нажатой клавише.

Коды извещения

Так же как редактор текста, список посылает в родительское окно сообщение WM_COMMAND (если он создан со стилем LBS_NOTIFY). Параметр wParam этого сообщения содержит идентификатор органа управления (в данном случае, идентификатор списка). Младшее слово параметра lParam содержит идентификатор окна списка, а старшее - код извещения.
Приведем список кодов извещения, поступающих от органа управления класса "listbox".

Код извещения	Описание
LBN_DBLCLK	Двойной щелчок левой клавишей мыши по строке списка
LBN_ERRSPACE	Ошибка при попытке заказать дополнительную память
LBN_KILLFOCUS	Список теряет фокус ввода
LBN_SELCANCEL	Пользователь отменил выбор в списке. Это извещение используется в Windows версии 3.1 и более поздних версий
LBN_SELCHANGE	Изменился номер выбранной строки (т. е. пользователь выбрал другую строку)
LBN_SETFOCUS	Список получает фокус ввода

Сообщения для списка

Для управления списком приложение посылает ему сообщения, вызывая функцию SendMessage. Эта функция возвращает значение, которое зависит от выполняемой функции или коды ошибок LB_ERRSPACE (ошибка при получении дополнительной памяти), LB_ERR (затребованная операция не может быть выполнена).

В файле windows.h определены сообщения, специально предназначенные для работы со списком. Символические имена этих сообщений имеют префикс LB_. Приведем список таких сообщений.

LB_ADDSTRING

Добавление строки в список.
Параметры: wParam = 0; lParam = (LPARAM)(LPCSTR)lpszStr;
lpszStr - указатель на добавляемую строку.
Возвращаемое значение: Номер строки в списке или код ошибки.

LB_DELETESTRING

Удаление строки из списка.
Параметры: wParam = (WPARAM)nIndex; lParam = 0L;
nIndex - номер удаляемой строки. Первая строка имеет номер 0.
Возвращаемое значение: Количество строк, оставшихся в списке, или код ошибки.

LB_DIR

Заполнение списка именами файлов и каталогов, расположенных в текущем каталоге, а также именами дисков.
Параметры:
wParam = (WPARAM)(UINT)uAttr; lParam = (LPARAM)(LPCSTR)lpszFileSpec;
uAttr - атрибуты файлов;
lpszFileSpec - указатель на строку, содержащую имя файла или шаблон имени файла.
Возвращаемое значение: Номер последнего имени файла или код ошибки.

LB_FINDSTRING

Поиск строки в списке, имеющей заданный префикс. Будет найдена строка, начальная часть которой совпадает с текстовой строкой, определенной в качестве префикса.
Параметры:
wParam = (WPARAM)nIndexStart; lParam = (LPARAM)(LPCSTR)lpszStr;
nIndexStart - номер строки, с которой начинается поиск;lpszStr- адрес префикса строки, которую нужно найти в списке.
Возвращаемое значение: Номер найденной строки, или код ошибки (если строки в списке нет).

LB_FINDSTRINGEXACT

Поиск строки в списке.
Параметры: wParam = (WPARAM)nIndexStart; lParam = (LPARAM)(LPCSTR)lpszStr;
nIndexStart - номер строки, с которой начинается поиск;lpszStr- адрес строки, которую нужно найти в списке.
Возвращаемое значение: Номер найденной строки, или код ошибки .

LB_GETCARETINDEX

Определение номера строки, имеющей фокус ввода.
Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: Номер строки, имеющей фокус ввода или код ошибки.

LB_GETCOUNT

Определение количества строк в списке.
Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: Количество строк в списке или код ошибки.

LB_GETCURSEL

Определение номера выделенной строки.
Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: Номер выделенной строки или код ошибки.

LB_GETHORIZONTALEXTENT

Определение ширины сворачиваемой области списка.
Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: Ширина сворачиваемой области списка в пикселях.

LB_GETITEMDATA

Получение 32-битового значения, соответствующего заданной строке. Напомним, что каждому элементу списка ставится в соответствие некоторое число, занчение которого можно определить с помощью этого сообщения.
Параметры: wParam = (WPARAM)nIndex; lParam = 0L;
nIndex - номер строки, для которой нужно получить значение.
Возвращаемое значение: Двойное слово искомого значения или код ошибки.

LB_GETITEMHEIGHT

Определение высоты заданной строки в списке, который рисуется родительским окном и имеет переменную высоту элементов.
Параметры: wParam = (WPARAM)nIndex; lParam = 0L;
nIndex - номер строки, для которой нужно получить значение.
Возвращаемое значение: Высота строки в пикселях или код ошибки.

LB_GETITEMRECT

Определение координат внутренней области окна, соответствующей заданной строке.
Параметры: wParam = (WPARAM)nIndex; lParam = (LPARAM)(RECT FAR *)lprc;
nIndex - номер строки, для которой нужно получить значение координат.
lprc - адрес структуры типа RECT, в которую будут записаны искомые координаты.
Возвращаемое значение: Код ошибки.

LB_GETSEL

С помощью этого сообщения можно определить, выбрана ли указанная строка списка.
Параметры: wParam = (WPARAM)nIndex; lParam = 0L;
nIndex - номер строки, о которой нужно получить информацию.
Возвращаемое значение: Положительное число, если строка выбрана, 0, если не выбрана или код ошибки.

LB_GETSELCOUNT

С помощью этого сообщения можно определить количество выбранных строк.
Параметры: wParam = 0; lParam = 0L;

Возвращаемое значение: Количество выбранных строк или код ошибки.

LB_GETSELITEMS

Заполнение буфера идентификаторами выбранных строк для списка, в котором можно выбирать несколько строк сразу.
Параметры: wParam = (WPARAM)cItems; lParam = (LPARAM)(int FAR *)lpItems ; cItems - максимальное количество выбранных строк.
lpItems - указатель на буфер для записи идентификаторов выбранных строк.
Возвращаемое значение: Количество идентификаторов или код ошибки.

LB_GETTEXT

Копирование текста, соответствующего заданной строке, в буфер. Если список не содержит строк (определен без стиля LBS_HASSTRING), в буфер копируется двойное слово, соответствующее указанному элементу списка.
Параметры: wParam = (WPARAM)nIndex; lParam = (LPARAM)(int FAR *)lpszBuffer ;nIndex - номер строки.l
pszBuffer - адрес буфера.
Возвращаемое значение: Длина строки в байтах (с учетом нуля), или код ошибки.

LB_GETTEXTLEN

Определение длины строки, содержащейся в списке.
Параметры: wParam = (WPARAM)nIndex; lParam = 0L;nIndex - номер строки.
Возвращаемое значение: Длина строки в байтах (с учетом нуля,), или код ошибки.

LB_GETTPOINDEX

Определение номера первой отображаемой строки.
Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: Номер строки или код ошибки.

LB_INSERTSTRING

Вставка элемента в заданную позицию списка. На расположение строки не влияет стиль LBS_SORT.Параметры:wParam = (WPARAM)nIndex;lParam = (LPARAM)(int FAR *)lpszBuffer;nIndex - номер позиции, в которую будет вставлена строка.lpszBuffer - адрес буфера.Возвращаемое значение:Номер позиции, в которую вставлена строка, или код ошибки.

LB_RESETCONTENT

Удаление всех строк из списка.
Параметры: wParam = 0; lParam = 0L; Возвращаемое значение: не используется.

LB_SELECTSTRING

Поиск строки в списке, которая начинается с символов, соответствующих образцу. Найденная строка становится выбранной.
Параметры: wParam = (WPARAM)nIndexStart; lParam = (LPARAM)(int FAR *)lpszBuffer;
nIndexStart - номер строки, с которой начинается поиск.lpszBuffer - адрес буфера, содержащего образец.
Возвращаемое значение: Номер найденной строки или код ошибки.

LB_SELITEMRANGE

Выделение одной или нескольких расположенных рядом строк.
Параметры: wParam = (WPARAM)(BOOL)fSelect; lParam = MAKELPARAM(wFirst, wLast);
fSelect - если TRUE, указанные строки выбираются и выделяются, если FALSE - выбор и выделение отменяются.
wFirst - номер первой выделяемой строки.
wLast - номер последней выделяемой строки.
Возвращаемое значение: Код ошибки.

LB_SETCARETINDEX

Передача фокуса ввода указанной строке. Если данная строка находится вне окна отображения, список сворачивается таким образом, чтобы строка стала видимой. Это сообщение используется в Windows версии 3.1 и более поздних версий.
Параметры: wParam = (WPARAM)nIndex; lParam = MAKELPARAM(fScroll, 0);
nIndex - номер строки.
fScroll - если TRUE, свертка выполняется до тех пор, пока указанная строка не будет видна хотя бы частично, если FALSE - до тех пор, пока строка не будет видна полностью.
Возвращаемое значение: Код ошибки.

LB_SETCOLUMNWIDTH

Установка ширины колонки в многоколоночном списке.
Параметры: wParam = (WPARAM)cxColumn; lParam = 0L;
cxColumn - ширина колонок списка в пикселях.
Возвращаемое значение: не используется.

LB_SETCURSEL

Выбор указанной строки. Ранее выделенная строка становится невыделенной. Если данная строка находится вне окна отображения, список сворачивается таким образом, чтобы строка стала видимой.
Параметры: wParam = (WPARAM)nIndex; lParam = 0L;
nIndex - номер строки. Если указать -1, выделение всех строк будет отменено. При этом функция SendMessage вернет значение LB_ERR, что в данном случае не говорит об ошибке.
Возвращаемое значение: Код ошибки (если значение nIndex не равно -1).

LB_SETHORIZONTALEXTENT

Установка ширины, на которую может быть свернут список, имеющий стиль WS_HSCROLL.
Параметры: wParam = (WPARAM)cxExtent; lParam = 0L; cxExtent - ширина в пикселях.
Возвращаемое значение: не используется.

LB_SETITEMDATA

Установка значения двойного слова, связанного с указанным элементом списка.
Параметры: wParam = (WPARAM)nIndex; lParam = (LPARAM)dwData;nIndex - номер строки.dwData - значение двойного слова.

Возвращаемое значение: Код ошибки.

LB_SETITEMHEIGHT

Установка высоты элемента в списке, который рисует родительское окно и имеет переменную высоту элементов. Это сообщение используется в Windows версии 3.1 и более поздних версий.
Параметры: wParam = (WPARAM)nIndex; lParam = MAKELPARAM(cyItem, 0);
nIndex - номер строки. Если список не имеет стиль LBS_OWNERDRAWVARIABLE, значение этого параметра должно быть равно 0.cyItem - высота элемента в пикселах.
Возвращаемое значение: Код ошибки.

LB_SETSEL

Установка высоты элемента в списке, который рисует родительское окно и имеет переменную высоту элементов. Это сообщение используется в Windows версии 3.1 и более поздних версий.
Параметры: wParam = (WPARAM)(BOLL)fSelect; lParam = MAKELPARAM(nIndex, 0);
fSelect - если TRUE, строка выбирается и выделяется, если FALSE - выделение и выбор строки отменяется.nIndex - номер строки. Можно указать как -1, в этом случае действие распространяется на все строки списка.
Возвращаемое значение: Код ошибки.

LB_SETTABSTOPS

Установка позиции табуляции в списке.
Параметры: wParam = (WPARAM)cTabs; lParam = (LPARAM)(int FAR *)lpTabs;
cTabs - количество позиций табуляции.lpTabs - адрес массива целых чисел, содержащих позиции табуляции.
Возвращаемое значение: Ненулевое значение, если позиции табуляции были установлены, в противном случае возвращается 0.

LB_SETTOPINDEX

Свертка списка до тех пор, пока указанная строка не станет видимой.
Параметры: wParam = (WPARAM)nIndex; lParam = 0L; nIndex - номер строки.
Возвращаемое значение: Код ошибки.

1.1.6. Список класса COMBOBOX

В этом разделе мы рассмотрим орган управления, создаваемый на базе предопределенного класса "combobox". Этот орган является комбинацией списка и однострочного редактора текста, поэтому для списка "combobox" используются стили, коды извещения и сообщения, аналогичные списку "listbox", а также некоторые сообщения, специфические для редактора текста класса "edit".

Создание списка COMBOBOX

Для того чтобы создать список класса "combobox" приложение должно вызвать функцию CreateWindow, передав в качестве первого параметра указатель на строку "combobox":

```
hComboBox = CreateWindow("ComboBox", NULL,  WS_CHILD | WS_VISIBLE |
WS_VSCROLL |  CBS_AUTOHSCROLL | CBS_SIMPLE,  30, 30, 200, 200,  hwnd,
(HMENU) ID_LIST, hInst, NULL);
```

Второй параметр функции должен быть указан как NULL.
При создании списка "combobox" указываются специальные стили списка, символические имена которых имеют префикс CBS_.
Остальные параметры функции CreateWindow указываются так же, как и для списка класса "listbox".

Стили списка

Приведем список стилей, которые используются для создания органа управления класса "combobox". Многие из этих стилей вам уже знакомы.

Имя стиля	Описание
CBS_AUTOHSCROLL	Выполняется автоматическая свертка текста по горизонтали в окне редактирования
CBS_DISABLENOSCROLL	Если в одноколоночном списке помещаются все строки, вертикальная полоса просмотра изображается в неактивном состоянии. Этот стиль можно указывать для Windows версии 3.1 и более поздних версий
CBS_DROPDOWN	Список остается в невидимом состоянии до тех пор, пока пользователь не нажмет пиктограмму, специально предназначенную для отображения списка
CBS_DROPDOWNLIST	Аналогично предыдущему, но однострочный текстовый редактор может быть использован только для отображения текста, но не для редактирования
CBS_HASSTRINGS	Создание списка, содержащего строки, который рисуется родительским окном
CBS_NOINTEGRALHEIGHT	Допустимо частичное отображение строк
CBS_OEMCONVERT	При вводе символов в окне редактирования выполняется их преобразование из кодировки ANSI в OEM и обратно. Этот стиль используется только совместно со стилями CBS_SIMPLE и CBS_DROPDOWN
CBS_OWNERDRAWFIXED	Создается список, который рисуется родительским окном, причем все элементы в списке имеют одинаковую высоту
CBS_OWNERDRAWVARIABLE	Аналогично предыдущему, но элементы списка могут иметь разную высоту
CBS_SIMPLE	Создается список, который всегда виден и расположен под окном однострочного редактора текста, содержащего выделенную в списке строку.
CBS_SORT	Строки списка будут отсортированы

Среди описанных выше стилей можно выделить три базовых.
Стиль CBS_SIMPLE соответствует списку с окном редактирования (или, как его еще называют, окном выбора).

Если в окне редактирования вводить строку символов (образец), то по мере ввода в списке будут появляться (и выделяться) строки, совпадающие по начальным символам с образцом. Например, если ввести букву "a", в списке окажется выделенной строка, начинающаяся с этой буквы. Если вслед за буквой "a" набрать букву "b", в списке будет выделена строка, начинающаяся с букв "ab" и т. д. Это очень удобно, например, для поиска строки по известным вам начальным буквам.
Если список имеет стиль CBS_DROPDOWN, в исходном состоянии он состоит из окна редактирования и расположенной справа от этого окна пиктограммы со стрелкой (кнопкой, предназначенной для отображения списка).
Стиль CBS_DROPDOWNLIST аналогичен стилю CBS_DROPDOWN, но окно редактирования можно использовать только для просмотра выделенной строки, а не для редактирования или ввода.

Коды извещения

Список "combobox" посылает в родительское окно сообщение WM_COMMAND. Параметр wParam этого сообщения содержит идентификатор списка. Младшее слово параметра lParam содержит идентификатор окна списка, а старшее - код извещения.
Приведем список кодов извещения, поступающих от органа управления класса "combobox".

Код извещения	Описание
CBN_CLOSEUP	Список исчез (стал невидим)
CBN_DBLCLK	Двойной щелчок левой клавишей мыши по строке списка, имеющего стиль CBS_SIMPLE
CBN_DROPDOWN	Список стал видимым
CBN_EDITCHANGE	Пользователь изменил содержимое окна редактирования, причем изменения уже отображены
CBN_EDITUPDATE	Пользователь изменил содержимое окна редактирования, изменения еще не отображены
CBN_ERRSPACE	Ошибка при попытке заказать дополнительную память
CBN_KILLFOCUS	Список теряет фокус ввода
CBN_SELENDCANCEL	Пользователь отменил выбор в списке.
CBN_SELENDOK	Пользователь выбрал строку в списке.
CBN_SELCHANGE	Изменился номер выбранной строки (т. е. пользователь выбрал другую строку)
CBN_SETFOCUS	Список получает фокус ввода

Сообщения для списка

Для управления списком "combobox" используется набор сообщений, аналогичный набору сообщений для списка "listbox" и редактора текста "edit". Функция SendMessage, посылающая сообщения списку "combobox", возвращает значение, которое зависит от выполняемой функции или коды ошибок CB_ERRSPACE (ошибка при получении дополнительной памяти), CB_ERR (затребованная операция не мо-

жет быть выполнена). Если операция выполнена без ошибок, возвращается значение CB_OKAY.
В файле windows.h определены сообщения, специально предназначенные для работы со списком "combobox". Символические имена этих сообщений имеют префикс CB_. Приведем список таких сообщений.

CB_ADDSTRING

Добавление строки в список.
Параметры: wParam = 0; lParam = (LPARAM)(LPCSTR)lpszStr;
lpszStr - указатель на добавляемую строку.
Возвращаемое значение: Номер строки в списке (первая строка имеет номер 0), или код ошибки.

CB_DELETESTRING

Удаление строки из списка.
Параметры: wParam = (WPARAM)nIndex; lParam = 0L;
nIndex - номер удаляемой строки. Первая строка имеет номер 0.
Возвращаемое значение: Количество строк, оставшихся в списке, или код ошибки.

CB_DIR

Заполнение списка именами файлов и каталогов, расположенных в текущем каталоге, а также именами дисков.
Параметры: wParam = (WPARAM)(UINT)uAttr; lParam = (LPARAM) (LPCSTR) lpszFileSpec; uAttr - атрибуты файлов; lpszFileSpec - указатель на строку, содержащую имя файла или шаблон имени файла.
Возвращаемое значение: Номер последнего имени файла, добавленного в список, или код ошибки.

CB_FINDSTRING

Поиск строки в списке, имеющей заданный префикс.
Параметры: wParam = (WPARAM)nIndexStart; lParam = (LPARAM)(LPCSTR)lpszStr;
nIndexStart - номер строки, с которой начинается поиск;
lpszStr- адрес префикса строки, которую нужно найти в списке.
Возвращаемое значение: Номер найденной строки, или код ошибки (если строки в списке нет).

CB_GETCOUNT

Определение количества строк в списке.
Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: Количество строк в списке или код ошибки.

CB_GETCURSEL

Определение номера выделенной строки.
Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: Номер выделенной строки или код ошибки.

CB_GETDROPPEDCONTROLRECT

Определение экранных координат видимой части списка. Используется в Windows версии 3.1 и более поздних версий.

Параметры: wParam = 0; lParam = (LPARAM)(RECT FAR *) lprc
;lprc - указатель на структуру RECT, в которую будут записаны искомые координаты.
Возвращаемое значение: Всегда возвращается CB_OKAY.

CB_GETDROPPEDSTATE

С помощью этого сообщения можно определить, находится список в видимом или невидимом состоянии.
Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: TRUE, если список виден, FALSE - если нет.

CB_GETEDITSEL

Определение положения первого и последнего символа в выделенном фрагменте .
Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: Двойное слово. Младшее слово содержит положение первого символа в выделенном фрагменте, старшее - положение символа, следующего за выделенным фрагментом текста

CB_GETEXTENDUI

С помощью этого сообщения можно определить, использует ли список расширенный интерфейс пользователя. Это сообщение используется в Windows версии 3.1 и более поздних версий.При использовании расширенного интерфейса щелчок в окне редактора текста для стиля CBS_DROPDOWMLIST приводит к отображению списка. Список также отображается, когда пользователь нажимает клавишу перемещения курсора вниз <Down>. Если список находится в невидимом состоянии, свертка окна редактирования не выполняется.
Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: TRUE, если расширенный интерфейс пользователя используется, FALSE - если нет.

CB_GETITEMDATA

Получение 32-битового значения, соответствующего заданной строке.
Параметры: wParam = (WPARAM)nIndex; lParam = 0L;
nIndex - номер строки, для которой нужно получить значение.
Возвращаемое значение: Двойное слово - искомое значение, или код ошибки.

CB_GETITEMHEIGHT

Определение высоты заданной строки в списке, который рисуется родительским окном и имеет переменную высоту элементов. Это сообщение используется в Windows версии 3.1 и более поздних версий.
Параметры: wParam = (WPARAM)nIndex; lParam = 0L;
nIndex - номер строки, для которой нужно получить значение.
Возвращаемое значение: Высота строки в пикселях или код ошибки.

CB_GETLBTEXT

Копирование текста, соответствующего заданной строке, в буфер.
Параметры: wParam = (WPARAM)nIndex; lParam = (LPARAM)(int FAR *)lpszBuffer;
nIndex - номер строки. lpszBuffer - адрес буфера.

Возвращаемое значение: Длина строки в байтах (с учетом двоичного нуля, закрывающего строку), или код ошибки.

CB_GETLBTEXTLEN

Определение длины строки, содержащейся в списке.
Параметры: wParam = (WPARAM)nIndex; lParam = 0L; nIndex - номер строки.
Возвращаемое значение: Длина строки в байтах (с учетом двоичного нуля, закрывающего строку), или код ошибки.

CB_INSERTSTRING

Вставка элемента в заданную позицию списка. На расположение строки не влияет стиль LBS_SORT.
Параметры: wParam = (WPARAM)nIndex; lParam = (LPARAM)(int FAR *)lpszBuffer;
nIndex - номер позиции, в которую будет вставлена строка.lpszBuffer - адрес буфера.
Возвращаемое значение: Номер позиции, в которую вставлена строка, или код ошибки.

CB_LIMITTEXT

Определение максимального количества символов, которое можно ввести в окно .
Параметры: wParam = (WPARAM)cCmax; lParam = 0L; cCMax - размер текста.
Возвращаемое значение: не используется

CB_RESETCONTENT

Удаление всех строк из списка.
Параметры: wParam = 0; lParam = 0L;
Возвращаемое значение: не используется.

CB_SELECTSTRING

Поиск строки в списке, которая начинается с символов, соответствующих образцу. Найденная строка становится выбранной.
Параметры: wParam = (WPARAM)nIndexStart; lParam = (LPARAM)(int FAR *)lpszBuffer;
nIndexStart - номер строки, с которой начинается поиск.lpszBuffer - адрес буфера, содержащего образец.
Возвращаемое значение: Номер найденной строки или код ошибки.

CB_SETCURSEL

Выбор указанной строки. Ранее выделенная строка становится невыделенной. Если данная строка находится вне окна отображения, список сворачивается таким образом, чтобы строка стала видимой.
Параметры: wParam = (WPARAM)nIndex; lParam = 0L;
nIndex - номер строки. Если указать -1, выделение всех строк будет отменено. При этом функция SendMessage вернет значение CB_ERR, что в данном случае не говорит об ошибке.
Возвращаемое значение: Код ошибки (если значение nIndex не равно -1).

CB_SETEDITSEL

Выделение заданных символов в окне редактирования.
Параметры:

wParam = (WPARAM)(UINT)fScroll; lParam = MAKELPARAM(ichStart, ichEnd);
fScroll - если этот параметр равен 1, текстовый курсор сворачивается, если 0 - нет.ichStart - начальная позиция. ichEnd - конечная позиция.
Если начальная позиция равна 0, а конечная -1, выбирается весь текст. Если начальная позиция равна -1, выделение фрагмента (если оно было) исчезает.
Возвращаемое значение: TRUE, если сообщение посылается операция выполнена без ошибок или код ошибки.

CB_SETEXTENDEDUI

Установка режима использования расширенного интерфейса пользователя. Это сообщение используется в Windows версии 3.1 и более поздних версий.
Параметры: wParam = (WPARAM)(BOOL)fExtended; lParam = 0L;
fExtended - TRUE для установки режима, FALSE - для сброса.
Возвращаемое значение: CB_OKAY, если сообщение посылается операция выполнена без ошибок или код ошибки CB_ERR.

CB_SETITEMDATA

Установка значения двойного слова, связанного с указанным элементом списка.
Параметры: wParam = (WPARAM)nIndex; lParam = (LPARAM)dwData;
nIndex - номер строки. dwData - значение двойного слова.
Возвращаемое значение: Код ошибки.

CB_SETITEMHEIGHT

Установка высоты элемента в списке, который рисует родительское окно и имеет переменную высоту элементов. Это сообщение используется в Windows версии 3.1 и более поздних версий.
Параметры: wParam = (WPARAM)nIndex; lParam = MAKELPARAM(cyItem, 0);
nIndex - номер строки.
Если список не имеет стиль LBS_OWNERDRAWVARIABLE, значение этого параметра должно быть равно 0.cyItem - высота элемента в пикселах.
Возвращаемое значение: Код ошибки.

CB_SHOWDROPDOWN

Переключение списка в отображаемое или неотображаемое состояние.
Параметры: wParam = (WPARAM)(BOOL)fExtended; lParam = 0L;
fExtended - TRUE для отображения списка, FALSE - для переключения списка в неотображаемое состояние.
Возвращаемое значение: всегда не равно 0

6.Меню в окнах Windows.

6.1. Классификация типов меню

При создании окна в приложении Windows вы можете указать, что окно должно иметь меню. Обычно меню создается в главном окне приложения. Такое меню мы будем называть меню приложения .
Для активизации строки меню вам надо установить на нее курсор и сделать щелчок левой клавишей мыши, либо нажать клавишу <Alt> и затем клавишу, соответствующую подчеркнутой букве. Например, для активизации строки "File" следует ис-

пользовать клавиши <Alt> и <F>. Если нажать, а затем отпустить клавишу <Alt>, нужную строку в меню приложения можно будет выбрать клавишами перемещения курсора по горизонтали <Left> и <Right>. Для активизации строки в последнем случае после выбора следует нажать клавишу <Enter>. Для отказа от выбора можно воспользоваться клавишей <Esc>.

Строки меню могут быть использованы либо для выбора команд, либо для активизации дополнительных временных меню (pop-up menu). Как правило, строки меню приложения используются только для активизации временных меню, но не для выполнения команд. Некоторые строки меню могут отображаться серым цветом. Это заблокированые строки, которые не могут быть выбраны.

Временное меню содержит строки, расположенные в столбец. Для выбора строки из временного меню вы можете воспользоваться мышью или клавишами перемещения курсора по вертикали <Up> и <Down>. В последнем случае для завершения выбора следует нажать клавишу <Enter>. Можно также воспользоваться клавишей <Alt> и клавишей, соответствующей подчеркнутой букве в нужной строке.

Строки временного меню могут быть отмечены галочкой . Такие строки обычно используются как переключатели, изменяющие режим работы приложения.

Мы уже говорили, что режимы работы приложения обычно задаются при помощи диалоговых панелей. Однако в простых случаях можно воспользоваться и строками меню.

Если выбор строки меню приводит к выполнению команды (например, команды создания документа, завершения работы приложения, копирования фрагмента документа в универсальный буфер обмена Clipboard и т. д.), строка меню содержит краткое название выполняемой команды, например, "New", "Copy", и т. д. Если же при выборе строки на экране появляется диалоговая панель, к слову справа добавляется многоточие . Последнее соглашение не является обязательным, однако вы должны ему следовать для обеспечения стандартного пользовательского интерфейса.

Вы можете создавать многоуровневые меню . Можно использовать многократную вложенность меню. Однако мы не советуем вам увлекаться сложными многоуровневыми меню, так как ими трудно пользоваться.

Каждое стандартное приложение Windows имеет системное меню , которое можно вызвать щелчком левой клавиши мыши по окну активизации системного меню, расположенному слева от заголовка окна либо при помощи комбинации клавиши <Alt> и клавиши пробела .

С помощью системного меню вы можете минимизировать (строка "Minimize") или максимизировать ("Maximize") главное окно приложения, восстанавливать размер этого окна ("Restore"), перемещать окно ("Move") или изменять его размер ("Size"), закрывать окно ("Close") и переключаться на другие приложения ("Switch To...").

Ваше приложение может изменять системное меню, дополняя его новыми строками или горизонтальными разделительными линиями, удалять строки из существующего меню.

Приложение может создать меню в любом месте экрана. Такое меню называют плавающим (floating menu), подчеркивая тот факт, что меню может появится в любом месте экрана или окна приложения.

В некоторых случаях плавающие меню удобнее обычных. Вы можете создавать плавающее меню двойным щелчком левой клавиши мыши или щелчком правой кла-

виши мыши, а также любым другим аналогичным способом. Если плавающее меню появится вблизи курсора мыши, из него будет легче выбрать нужную строку, чем из обычного меню, так как не надо перемещать курсор в верхнюю часть экрана. Кроме того, создавая плавающее меню щелчком мыши, вы можете изменять внешний вид этого меню в зависимости от объекта, по изображению которого был сделан щелчок.
Это позволит реализовать объектно-ориентированный подход в работе пользователя с приложением - в зависимости от того, для какого объекта было создано плавающее меню, изменяется содержимое меню. Таким образом, для того чтобы выполнить какую-либо операцию над объектом, пользователю будет достаточно щелкнуть по нему мышью. Около объекта появится плавающее меню операций, определенных для данного объекта.
Меню не обязательно должно содержать только текстовые строки. Вы можете создать меню из графических изображений или из комбинации графических изображений и текста.
Для создания меню с графическими изображениями можно использовать методы, аналогичные применяемым при создании органов управления, рисуемых родительским окном, или специальные функции из программного интерфейса Windows.

6.2. Создание меню при помощи шаблона

Для создания меню вы можете использовать три метода.
1)Можно описать шаблон меню в файле ресурсов приложения (аналогично шаблону диалоговой панели, но с использованием других операторов). Этот способ больше всего подходит для создания статических меню, не меняющихся или меняющихся не очень сильно в процессе работы приложения.
2)Можно создать меню "с нуля" при помощи специальных функций программного интерфейса Windows. Этот способ хорош для приложений, меняющих внешний вид меню, когда вы не можете создать заранее подходящий шаблон. Разумеется, второй метод пригоден и для создания статических меню.
3)Можно подготовить шаблон меню непосредственно в оперативной памяти и создать меню на базе этого шаблона.

Создание шаблона меню

Шаблон меню можно создать в текстовом виде либо при помощи редактора ресурсов, либо обычным текстовым редактором. В любом случае перед сборкой приложения текстовое описание шаблона меню должно находиться в файле ресурсов с расширением имени .rc, указанном в проекте приложения (или в файле, включаемом в файл проекта оператором #include).
Описание шаблона меню имеет следующий вид:

```
nameID MENU   [load] [mem]
BEGIN
  . . .
  . . .
  . . .
END
```

Поле nameID используется для идентификации шаблона меню. Оно может указываться либо в виде текстовой строки, либо в виде числа от 1 до 65535.
Параметр load - необязательный. Он используется для определения момента загрузки меню в память. Если этот параметр указан как PRELOAD, меню загружается в память сразу после запуска приложения. По умолчанию используется значение LOADONCALL, в этом случае загрузка шаблона в память происходит только при отображении меню.
Параметр mem также необязательный. Он влияет на тип памяти, выделяемой для хранения шаблона, и может указываться как FIXED (ресурс всегда остается в фиксированной области памяти), MOVEABLE (при необходимости ресурс может перемещаться в памяти, это значение используется по умолчанию) или DISCARDABLE (если ресурс больше не нужен, занимаемая им память может быть использована для других задач). Значение DISCARDABLE может использоваться вместе со значением MOVEABLE.
Между строками BEGIN и END в описании шаблона располагаются операторы описания строк MENUITEM и операторы описания временных меню POPUP.
Оператор MENUITEM имеет следующий формат:
MENUITEM text, id [, param]
Параметр text определяет имя строки меню. Вы должны указать текстовую строку в двойных кавычках, например, "File". Текстовая строка может содержать символы &, \t, \a.
Если в текстовой строке перед буквой стоит знак &, при выводе меню данная буква будет подчеркнута. Например, строка "&File" будет отображаться как "File". Клавиша, соответствующая подчеркнутой букве, может быть использована в комбинации с клавишей <Alt> для ускоренного выбора строки. Для того чтобы записать в строку сам символ &, его следует повторить дважды. Аналогично, для записи в строку меню символа двойной кавычки " его также следует повторить дважды.
Символ \t включает в строку меню символ табуляции и может быть использован при выравнивании текста в таблицах. Этот символ обычно используется только во временных и плавающих меню, но не в основном меню приложения, расположенном под заголовком главного окна.
Символ \a выравнивает текст по правой границе временного меню или полосы меню.
Параметр id представляет собой целое число, которое должно однозначно идентифицировать строку меню. Приложение получит это число в параметре wParam сообщения WM_COMMAND, когда вы выберете данную строку.
Необязательный параметр param указывается как совокупность атрибутов, разделенных запятой или пробелом. Эти атрибуты определяют внешний вид и поведение строки меню:

Атрибут	Описание
CHECKED	При выводе меню на экран строка меню отмечается галочкой " "
GRAYED	Строка меню отображается серым цветом и находится в неактивном состоянии. Такую строку нельзя выбрать. Этот атрибут несовместим с атрибутом INACTIVE

HELP	Слева от текста располагается разделитель в виде вертикальной линии
INACTIVE	Строка меню отображается в нормальном виде (не серым цветом), но находится в неактивном состоянии. Этот атрибут несовместим с атрибутом GRAYED
MENUBREAK	Если описывается меню верхнего уровня, элемент меню выводится с новой строки. Если описывается временное меню, элемент меню выводится в новом столбце
MENUBARBREAK	Аналогично атрибуту MENUBREAK, но дополнительно новый столбец отделяется вертикальной линией (используется при создании временных меню)

Для описания временных меню используется оператор POPUP :

```
POPUP text [, param]
BEGIN
  . . .
  . . .
  . . .
END
```

Между строками BEGIN и END в описании временного меню располагаются операторы описания строк MENUITEM и операторы описания вложенных временных меню POPUP.

Параметры text и param указываются так же, как и для оператора MENUITEM .

Для того чтобы создать в меню горизонтальную разделительную линию, используется специальный вид оператора MENUITEM : MENUITEM SEPARATOR

Подключение меню к окну приложения

Следующий этап после создания меню - подключение меню к окну приложения. Обычно меню определяется для класса окна при регистрации или для отдельного окна при его создании функцией CreateWindow.

Подключение меню при регистрации класса окна

Если при регистрации класса окна в поле lpszMenuName структуры типа WNDCLASS указать адрес текстовой строки, содержащей имя шаблона меню в файле ресурсов, все перекрывающиеся и временные окна, создаваемые на базе этого класса, будут иметь меню, определенное данным шаблоном. Дочерние окна (child window) не могут иметь меню. Например, пусть в файле описания ресурсов шаблон меню определен под именем APP_MENU:

```
APP_MENU MENU
BEGIN
....
....
....
END
```

В этом случае для подключения меню при регистрации класса вы должны записать адрес текстовой строки "APP_MENU" в поле lpszMenuName структуры wc, имеющей тип WNDCLASS: wc.lpszMenuName = "APP_MENU";

Вы можете использовать для идентификации шаблона меню целые числа (как и для идентификации ресурсов других типов). В этом случае необходимо использовать макрокоманду MAKEINTRESOURCE.
Например, пусть в файле описания ресурсов и в файле исходного текста приложения определена константа: #define APP_MENU 123
В этом случае ссылка на меню при регистрации класса окна должна выполняться следующим образом: wc.lpszMenuName = MAKEINTRESOURCE(APP_MENU);
Когда для класса окна определено меню, все перекрывающиеся и временные окна, создаваемые на базе этого класса, будут иметь меню, если при создании окна функцией CreateWindow идентификатор меню указан как 0.

Подключение меню при создании окна

Если при регистрации класса окна было определено меню, вы можете создавать окна с этим меню, или можете указать для создаваемого окна другое меню. Для подключения меню, отличного от указанного в классе окна, вам необходимо задать идентификатор нужного меню при создании окна функцией CreateWindow. Короче говоря, окно может иметь меню, определенное в классе, или свое собственное.
Девятый параметр функции CreateWindow используется для подключения меню к создаваемому окну:

```
hwnd = CreateWindow(
   szClassName,                  // имя класса окна
   szWindowTitle,                // заголовок окна
   WS_OVERLAPPEDWINDOW, // стиль окна
   CW_USEDEFAULT,                // размеры и расположение окна
   CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
   0,                                  // идентификатор родительского окна
   hmenu,                              // идентификатор меню
   hInstance,                          // идентификатор приложения
   NULL);                              // указатель на дополнительные параметры
```

Значение параметра идентификатора меню может быть получено, например, от функции LoadMenu , определенной в программном интерфейсе Windows:
HMENU WINAPI LoadMenu(HINSTANCE hInstance,   LPCSTR lpszMenuName);
Параметр hInstance должен содержать идентификатор текущей копии приложения, полученный через соответствующий параметр функции WinMain.
Параметр lpszMenuName является указателем на строку символов, закрытую двоичным нулем, содержащую имя загружаемого шаблона меню. Если для идентификации шаблона меню используется целое число, необходимо сформировать этот указатель при помощи макрокоманды MAKEINTRESOURCE.
Функция LoadMenu возвращает идентификатор загруженного меню или NULL при ошибке.
Перед завершением своей работы приложение должно уничтожить загруженное меню функцией DestroyMenu : BOOL WINAPI DestroyMenu(HMENU hmenu);
В качестве единственного параметра функции DestroyMenu необходимо указать идентификатор уничтожаемого меню. Функция DestroyMenu возвращает в случае успеха значение TRUE, при ошибке - FALSE.

6.3. Сообщения, поступающие от меню

Меню посылает сообщения в функцию создавшего его окна.
Сообщение WM_INITMENU посылается перед отображением меню и может быть использовано для инициализации. Сообщение WM_COMMAND посылается после того, как пользователь выберет одну из активных строк меню. Системное меню посылает в окно приложения сообщение WM_SYSCOMMAND, которое обычно не обрабатывается приложением (передается функции DefWindowProc). В процессе выбора строки из меню, когда курсор перемещается по строкам меню, функция окна, создавшего меню, получает сообщение WM_MENUSELECT. Перед инициализацией временного меню функция окна получает сообщение WM_INITMENUPOPUP. Из всех этих сообщений наибольший интерес представляют сообщения WM_INITMENU, M_INITMENUPOPUP, WM_COMMAND, WM_SYSCOMMAND.

Сообщение WM_INITMENU

Сообщение WM_INITMENU посылается окну, создавшему меню, в момент отображения меню. Это происходит, когда вы нажимаете на строку в полосе меню или активизируете временное меню при помощи клавиатуры. Вместе с этим сообщением в параметре wParam передается идентификатор активизируемого меню. Параметр lParam не используется. Если приложение обрабатывает сообщение WM_INITMENU, соответствующий обработчик должен вернуть нулевое значение. Обработка может заключаться в активизации или деактивизации строк меню, изменении состояния строк (отмеченное галочкой или не отмеченное) и т. п.

Сообщение WM_INITMENUPOPUP

Сообщение WM_INITMENUPOPUP посылается окну, когда операционная система Windows готова отобразить временное меню. Младшее слово параметра lParam содержит порядковый номер временного меню в меню верхнего уровня, старшее слово содержит 1 для системного меню или 0 для обычного меню. Это сообщение можно использовать для активизации или блокирования отдельных строк временного меню.

Сообщение WM_COMMAND

Сообщение WM_COMMAND посылается функции окна приложения, создавшего меню, когда вы выбираете нужную вам строку. Параметр wParam содержит идентификатор строки, определенный в шаблоне меню. Задача функции окна, обрабатывающей сообщения от меню, заключается в проверке значения параметра wParam и выполнении соответствующей функции.

Сообщение WM_SYSCOMMAND

Сообщение WM_SYSCOMMAND приходит в функцию окна приложения, когда пользователь выбирает строку из системного меню. Параметр wParam, как и для сообщения WM_COMMAND, содержит идентификатор строки меню, в данном случае, идентификатор строки системного меню. Параметр lParam не используется (за исключением идентификатора SC_HOTKEY).
Приведем список идентификаторов с кратким описанием.

Идентификатор	Описание

SC_CLOSE	Удаление окна (строка "Close")
SC_HOTKEY	Активизация окна, связанного с комбинацией клавиш, определенной приложением. Младшее слово параметра lParam содержит идентификатор активизируемого окна
SC_HSCROLL	Свертка по горизонтали
SC_KEYMENU	Выбор из меню при помощи комбинации клавиш
SC_MAXIMIZE или SC_ZOOM	Максимизация окна (строка "Maximize")
SC_MINIMIZE или SC_ICON	Минимизация окна (строка "Minimize")
SC_MOUSEMENU	Выбор из меню при помощи мыши
SC_MOVE	Перемещение окна (строка "Move")
SC_NEXTWINDOW	Переключение на следующее окно
SC_PREVWINDOW	Переключение на предыдущее окно
SC_RESTORE	Восстановление нормального положения и размера окна
SC_SCREENSAVE	Запуск приложения, предназначенного для предохранения экрана дисплея от преждевременного выгорания (screen-saver application), определенного в разделе [boot] файла system.ini
SC_SIZE	Изменение размера окна (строка "Size")
SC_TASKLIST	Запуск или активизация приложения Task Manager
SC_VSCROLL	Свертка по вертикали

При анализе параметра wParam учтите, что младшие четыре бита этого параметра могут принимать любые значения и должны игнорироваться:

```
if((wParam & 0xfff0) == SC_SIZE){  return 0;}
```

Можно добавлять строки в системное меню. При добавлении строк в системное меню вы должны указывать идентификатор строки. Этот идентификатор (с учетом сказанного выше относительно младших четырех битов) вы получите в параметре wParam сообщения WM_SYSCOMMAND при выборе добавленной вами строки. Создав собственный обработчик для сообщений, приходящих от системного меню, вы можете блокировать отдельные или все строки этого меню. Для блокировки какой-либо строки соответствующий обработчик должен вернуть нулевое значение, как в приведенном выше фрагменте кода, блокирующем изменение размера окна.

6.4. Функции для работы с меню

В программном интерфейсе операционной системы Windows есть функции, специально предназначенные для работы с меню. С помощью этих функций приложение может создавать меню (даже не имея его шаблона), добавлять или удалять строки или временные меню, активизировать или блокировать отдельные строки меню, изменять состояние строк (отмеченное или не отмеченное) и т. д.

Создание меню

Даже если в файле описания ресурсов нет определения шаблона меню, приложение может создать меню "с нуля" для любого своего перекрывающегося или временного

окна (но не для дочернего). Для создания пустого меню (то есть меню, не содержащего ни одной строки и ни одного временного меню) можно воспользоваться функцией CreateMenu : HMENU WINAPI CreateMenu(void);
Функция возвращает идентификатор созданного меню или NULL при ошибке.
Как правило, в меню верхнего уровня (в меню приложения) создаются временные меню. Для создания временного меню воспользуйтесь функцией CreatePopupMenu:
HMENU WINAPI CreatePopupMenu (void);
В дальнейшем вы можете добавить в меню верхнего уровня созданные функцией CreatePopupMenu временные меню или отдельные строки, вызвав функцию AppendMenu.
Перед завершением работы приложение должно удалить созданные описанным выше способом меню, для чего следует воспользоваться функцией DestroyMenu.
Для подключения к окну с идентификатором hwnd меню с идентификатором hmenu вы можете воспользоваться функцией SetMenu:
BOOL WINAPI SetMenu (HWND hwnd, HMENU hmenu);
Перед вызовом этой функции вы должны загрузить меню и получить его идентификатор, например, при помощи функции LoadMenu.
Функция SetMenu возвращает TRUE при успешном завершении и FALSE при ошибке.

Добавление строк

Для добавления строк в созданные функциями CreateMenu и CreatePopupMenu пустые меню можно воспользоваться функцией AppendMenu :
BOOL WINAPI AppendMenu(HMENU hmenu, UINT fuFlags, UINT idNewItem, LPCSTR lpszNewItem);
Параметр hmenu указывает идентификатор меню, к которому добавляется строка или временное меню. Вы должны использовать значение, полученное от функций CreateMenu и CreatePopupMenu.
Параметр fuFlags определяет атрибуты создаваемого элемента меню. Можно указывать следующие значения (соответствующие символические константы описаны в файле windows.h):

Константа	Описание
MF_BITMAP	Для изображения строки меню используется графическое изображение bitmap. Если указан этот параметр, младшее слово параметра lpszNewItem содержит идентификатор изображения
MF_CHECKED	При выводе меню на экран строка меню отмечается галочкой ""
MF_DISABLED	Строка меню отображается в нормальном виде (не серым цветом), но находится в неактивном состоянии
MF_ENABLED	Строка меню разблокирована и отображается в нормальном виде
MF_GRAYED	Строка меню отображается серым цветом и находится в неактивном состоянии. Такую строку нельзя выбрать
MF_MENUBREAK	Если описывается меню верхнего уровня, элемент меню

	выводится с новой строки. Если описывается временное меню, элемент меню выводится в новом столбце
MF_MENUBARBREAK	Аналогично MF_MENUBREAK, но дополнительно новый столбец отделяется вертикальной линией (используется при создании временных меню)
MF_OWNERDRAW	Строка меню рисуется окном, создавшем меню. Когда меню отображается в первый раз, функция этого окна получает сообщение WM_MEASUREITEM, в ответ на которое функция окна должна сообщить размеры области, занимаемой изображением строки меню. Рисовать изображение строки надо тогда, когда в функцию окна придет сообщение WM_DRAWITEM. Флаг MF_OWNERDRAW можно указывать только для временных меню
MF_POPUP	С данным элементом меню связывается временное меню. Если используется этот флаг, параметр idNewItem должен содержать идентификатор временного меню, связанного с данным элементом
MF_SEPARATOR	Используется для создания горизонтальной разделительной линии во временных меню. Если указан этот флаг, параметры lpszNewItem и idNewItem не используются
MF_STRING	Элемент меню является строкой символов. Параметр lpszNewItem должен указывать на строку символов, закрытую двоичным нулем
MF_UNCHECKED	При выводе меню на экран строка не отмечается галочкой ""

Вы можете указывать сразу несколько флагов, объединив их операцией логического ИЛИ, однако следует иметь в виду, что существует четыре группы взаимно несовместимых флагов:
MF_DISABLED, MF_ENABLED, MF_GRAYED
MF_BITMAP, MF_OWNERDRAW, MF_STRING
MF_MENUBREAK, MF_MENUBARBREAK
MF_CHECKED, MF_UNCHECKED
Назначение параметра idNewItem зависит от параметра fuFlags. Если значение параметра fuFlags не равно MF_POPUP, через idNewItem вы должны передать идентификатор создаваемой строки меню. Этот идентификатор будет записан в параметр wParam сообщения WM_COMMAND при выборе данной строки. Если же значение параметра fuFlags равно MF_POPUP, через параметр idNewItem вы должны передать функции AppendMenu идентификатор временного меню.
Назначение параметра lpszNewItem также зависит от параметра fuFlags. Если этот параметр равен MF_STRING, параметр lpszNewItem должен указывать на строку символов, закрытую двоичным нулем, если MF_BITMAP - младшее слово параметра lpszNewItem содержит идентификатор изображения, а если параметр fuFlags равен MF_OWNERDRAW, приложение должно передать через параметр lpszNewItem 32-битовое значение, идентифицирующее строку меню.

Еще одна функция, предназначенная для добавления элементов в меню, называется InsertMenu . Эта функция может добавить элемент в середину меню, сдвинув вниз уже существующие элементы. Прототип функции InsertMenu:
BOOL WINAPI InsertMenu(HMENU hmenu, UINT idItem, UINT fuFlags, UINT idNewItem, LPCSTR lpszNewItem);
Параметры этой функции аналогичны параметрам функции AppendMenu, за исключением параметров idItem и fuFlags.
Параметр idItem определяет элемент меню, перед которым должен быть вставлен новый элемент. Интерпретация этого параметра зависит от значения параметра fuFlags.
В дополнение к возможным значениям параметра fuFlags, описанным нами для функции AppendMenu, вместе с функцией InsertMenu вы можете использовать еще два - MF_BYCOMMAND и MF_BYPOSITION.
Если указан флаг MF_BYCOMMAND, параметр idItem определяет идентификатор элемента меню, перед которым будет вставлен новый элемент.
Если указан флаг MF_BYPOSITION, параметр idItem определяет порядковый номер элемента меню, перед которым будет вставлен новый элемент. Для того чтобы добавить элемент в конец меню, для параметра idItem можно указать значение -1.
После внесения всех изменений в меню приложение должно вызвать функцию DrawMenuBar : void WINAPI DrawMenuBar(HWND hwnd);
Эта функция перерисовывает полосу меню для указанного параметром hwnd окна. В качестве параметра функции следует передать идентификатор окна, создавшего меню.

Изменение строк

Для изменения строк (элементов) существующего меню вы можете воспользоваться функцией ModifyMenu :
BOOL WINAPI ModifyMenu(HMENU hmenu, UINT idItem, UINT fuFlags, UINT idNewItem, LPCSTR lpszNewItem);
Параметры этой функции идентичны параметрам функции InsertMenu.
Функция ModifyMenu заменяет указанный элемент меню на новый. При замещении временного меню оно уничтожается и все связанные с ним ресурсы освобождаются.
После того как вы изменили меню, не забудьте вызывать функцию DrawMenuBar, описанную выше.
В программном интерфейсе Windows для изменения существующего меню есть следующие пять функций:

Функция	Описание
AppendMenu	Добавление элемента в меню
DeleteMenu	Удаление элемента из меню
InsertMenu	Вставка элемента в меню
ModifyMenu	Изменение элемента меню
RemoveMenu	Удаление элемента меню без освобождения ресурсов, занимаемых этим элементом

Удаление строк

Для удаления элементов меню, таких, как строки и временные меню, предназначена функция DeleteMenu :
BOOL WINAPI DeleteMenu(HMENU hmenu, UINT idItem, UINT fuFlags);
Параметр hmenu определяет меню, из которого будет удален элемент.
Параметр idItem определяет удаляемый элемент, причем его интерпретация зависит от значения параметра fuFlags.
Если в параметре fuFlags указан флаг MF_BYCOMMAND, параметр idItem определяет идентификатор удаляемого элемента меню. Если указан флаг MF_BYPOSITION, параметр idItem определяет порядковый номер удаляемого элемента меню.
При удалении временного меню все связанные с ним ресурсы освобождаются.
Для отображения результата удаления меню следует вызвать функцию DrawMenuBar.
В программном интерфейсе Windows определена функция RemoveMenu , имеющая параметры, аналогичные параметрам функции DeleteMenu:
BOOL WINAPI RemoveMenu(HMENU hmenu, UINT idItem, UINT fuFlags);
Эта функция удаляет указанный ей элемент из меню, но не уничтожает связанные с ним ресурсы, поэтому вы можете вновь воспользоваться удаленным элементом меню (если знаете его идентификатор, о том как получить идентификатор временного меню мы расскажем немного позже).
Для уничтожения меню используется функция DestroyMenu :
BOOL WINAPI DestroyMenu(HMENU hmenu);
В качестве параметра функции передается идентификатор уничтожаемого меню.
Функция освобождает все ресурсы, связанные с уничтоженным меню.

Активизация и блокирование строк меню

Для изменения состояния элемента меню удобно использовать функцию EnableMenuItem :
BOOL WINAPI EnableMenuItem(HMENU hmenu, UINT idItem, UINT uEnable);
Параметр hmenu указывает идентификатор меню, над элементом которого будет выполняться операция активизации или блокирования.
Параметр idItem определяет элемент меню, над которым выполняется операция. Интерпретация этого параметра зависит от значения параметра uEnable.
Параметр uEnable может принимать значения MF_DISABLED, MF_ENABLED или MF_GRAYED в комбинации с одним из значений: MF_BYCOMMAND или MF_BYPOSITION.
Для блокирования элемента меню необходимо использовать значение MF_DISABLED. Если заблокированный элемент меню нужно изобразить серым цветом, вместо MF_DISABLED используйте значение MF_GRAYED.
Для активизации заблокированного ранее элемента меню укажите значение MF_ENABLED.
Если в параметре fuFlags указан флаг MF_BYCOMMAND, параметр idItem определяет идентификатор элемента меню, состояние которого будет изменено. Если указан флаг MF_BYPOSITION, параметр idItem определяет порядковый номер элемента меню, состояние которого будет изменено.

Как и после выполнения других операций по изменению меню, после изменения состояния элемента меню необходимо вызвать функцию DrawMenuBar, которая отобразит внесенные изменения на экране.

Отметка строк

Элементы временного меню могут быть отмечены галочкой. Для включения и выключения такой отметки можно использовать функцию CheckMenuItem :
BOOL WINAPI CheckMenuItem(HMENU hmenu, UINT idItem, UINT fuCheck);
Параметр hmenu указывает идентификатор меню, над элементом которого будет выполняться операция включения или выключения отметки.
Параметр idItem определяет элемент меню, над которым выполняется операция. Интерпретация этого параметра зависит от значения параметра fuCheck.
Параметр fuCheck может принимать значения MF_CHECKED или MF_UNCHECKED в комбинации с одним из значений: MF_BYCOMMAND или MF_BYPOSITION.
Для включения отметки элемента меню необходимо использовать значение MF_CHECKED. Для выключения отметки элемента меню укажите значение MF_UNCHECKED. Если в параметре fuCheck указан флаг MF_BYCOMMAND, параметр idItem определяет идентификатор элемента меню, отметка которого будет изменена. Если указан флаг MF_BYPOSITION, параметр idItem определяет порядковый номер элемента меню, отметка которого будет изменена.

Выделение строк

Для выделения строк меню верхнего уровня, расположенных в полосе меню ниже заголовка окна, можно использовать функцию HiliteMenuItem :
BOOL WINAPI HiliteMenuItem(HWND hwnd, HMENU hmenu, UINT idItem, UINT fuHilite);
Параметр hwnd должен содержать идентификатор окна, для которого выполняется операция выделения. Через параметр hMenu необходимо передать идентификатор соответствующего меню верхнего уровня. Параметр idItem определяет элемент меню, над которым выполняется операция выделения. Интерпретация этого параметра зависит от значения параметра fuHilite. Параметр fuHilite может принимать значения MF_HILITE или MF_UNHILITE в комбинации с одним из значений: MF_BYCOMMAND или MF_BYPOSITION.
Для выделения строки меню необходимо использовать значение MF_HILITE. Для отмены выделения строки меню укажите значение MF_UNHILITE.
Если в параметре fuHilite указан флаг MF_BYCOMMAND, параметр idItem определяет идентификатор строки меню, для которого выполняется операция выделения или отмены выделения. Если указан флаг MF_BYPOSITION, параметр idItem определяет порядковый номер этой строки.

Получение информации

В программном интерфейсе операционной системы Windows существует несколько функций для получения различной информации о меню и о состоянии строк меню.

Идентификатор меню

С помощью функции GetMenu вы можете определить идентификатор меню, связанного с окном: HMENU WINAPI GetMenu(HWND hwnd);
Идентификатор окна задается при помощи параметра hwnd.
Функция возвращает идентификатор меню или NULL, если окно не имеет меню. Дочернее окно не может иметь меню, однако в документации к SDK говорится, что если вы вызовете данную функцию для дочернего окна, возвращенное значение будет неопределено.

Идентификатор временного меню

Для определения идентификатора временного меню следует вызвать функцию GetSubMenu : HMENU WINAPI GetSubMenu(HMENU hmenu, int nPos);
Эта функция для меню верхнего уровня с идентификатором hmenu возвращает идентификатор временного меню, порядковый номер которого задается параметром nPos. Первому временному меню соответствует нулевое значение параметра nPos. Если функция GetSubMenu вернула значение NULL, то меню верхнего уровня не содержит в указанной позиции временное меню.

Проверка идентификатора меню

Для того чтобы убедиться, что идентификатор не является идентификатором меню, вы можете использовать функцию IsMenu :BOOL WINAPI IsMenu(HMENU hmenu);
Функция возвращает значение FALSE, если переданный ей через параметр hmenu идентификатор не является идентификатором меню. Можно было бы ожидать, что если функция IsMenu вернула значение TRUE, то проверяемый идентификатор является идентификатором меню, однако в описании функции сказано, что это не гарантируется.

Количество элементов в меню

Функция GetMenuItemCount возвращает количество элементов в меню верхнего уровня или во временном меню, заданном параметром hmenu:
int WINAPI GetMenuItemCount(HMENU hmenu);

Идентификатор элемента меню

Для получения идентификатора элемента меню, расположенного в указанной позиции, вы можете воспользоваться функцией GetMenuItemID :
UINT WINAPI GetMenuItemID(HMENU hmenu, int nPos);
Параметр hmenu задает меню, идентификатор элемента которого требуется определить. Порядковый номер элемента определяется параметром nPos, причем первому элементу соответствует нулевое значение. В случае ошибки (если параметр hmenu указан как NULL или указанный элемент является временным меню) функция GetMenuItemID возвращает значение -1. Если вы попытаетесь определить идентификатор горизонтальной разделительной линии (сепаратора), функция вернет нулевое значение.

Текст строки меню

С помощью функции GetMenuString вы можете переписать в буфер текстовую строку, соответствующую элементу меню.

int WINAPI GetMenuString(HMENU hmenu, UINT idItem, LPSTR lpsz, int cbMax, UINT fuFlags);
Параметр hmenu определяет меню, для которого будет выполняться операция. Параметр idItem определяет элемент меню, над которым выполняется операция. Интерпретация этого параметра зависит от значения параметра fuFlags.
Если в параметре fuFlags указан флаг MF_BYCOMMAND, параметр idItem определяет идентификатор строки меню, для которого выполняется операция. Если указан флаг MF_BYPOSITION, параметр idItem определяет порядковый номер этой строки. Адрес буфера, в который будет выполняться копирование, задается параметром lpsz, размер буфера без учета двоичного нуля, закрывающего строку, - оператором cbMax. Символы, не поместившиеся в буфер, будут обрезаны.
Функция GetMenuString возвращает количество символов, скопированных в буфер, без учета двоичного нуля, закрывающего строку.

Флаги состояния элемента меню

Функция GetMenuState возвращает флаги состояния для заданного элемента меню:
UINT WINAPI GetMenuState(HMENU hmenu, UINT idItem, UINT fuFlags);
Параметр hmenu определяет меню, для которого будет выполняться операция. Параметр idItem определяет элемент меню, для которого будут получены флаги состояния. Интерпретация этого параметра зависит от значения параметра fuFlags.
Если в параметре fuFlags указан флаг MF_BYCOMMAND, параметр idItem определяет идентификатор строки меню, для которого выполняется операция. Если указан флаг MF_BYPOSITION, параметр idItem определяет порядковый номер этой строки. Для временного меню старший байт возвращаемого функцией значения содержит количество элементов во временном меню, а младший - набор флагов, описывающих временное меню. Для меню верхнего уровня возвращаемое значение является набором флагов, описывающих указанный элемент меню:

Флаг	Описание
MF_BITMAP	Для изображения строки меню используется графическое изображение bitmap
MF_CHECKED	Строка меню отмечена галочкой ""
MF_DISABLED	Строка меню находится в неактивном состоянии
MF_ENABLED	Строка меню разблокирована и отображается в нормальном виде. Этому состоянию соответствует возвращаемое функцией GetMenuState значение, равное нулю
MF_GRAYED	Строка меню отображается серым цветом и находится в неактивном состоянии. Такую строку нельзя выбрать
MF_MENUBREAK	Для меню верхнего уровня элемент меню выводится с новой строки. Для временного меню элемент выводится в новом столбце
MF_MENUBARBREAK	Аналогично MF_MENUBREAK, но дополнительно столбец отделен вертикальной линией
MF_SEPARATOR	Строка является горизонтальной разделительной линией во временных меню

MF_UNCHECKED	Строка не отмечена галочкой ""

Если указанный элемент меню не существует, функция GetMenuState возвращает значение -1.

6.5. Системное меню

При необходимости вы можете изменить системное меню , добавив в него новые строки или горизонтальные разделительные линии. Прежде всего вам надо получить идентификатор системного меню. Это можно сделать при помощи функции GetSystemMenu : HMENU WINAPI GetSystemMenu(HWND hwnd, BOOL fRevert);
Параметр hwnd является идентификатором окна, к системному меню которого требуется получить доступ. Параметр fRevert определяет действия, выполняемые функцией GetSystemMenu. Если этот параметр указан как FALSE, функция GetSystemMenu возвращает идентификатор используемой на момент вызова копии системного меню. Если же значение этого параметра равно TRUE, функция восстанавливает исходный вид системного меню, используемый в Windows по умолчанию и уничтожает все созданные ранее копии системного меню. В последнем случае возвращаемое значение не определено.
После того как вы получили идентификатор системного меню, вы можете использовать функции AppendMenu, InsertMenu или ModifyMenu для изменения внешнего вида системного меню.
Есть одна особенность, которую нужно учитывать при добавлении собственной строки в системное меню. Младшие четыре бита в сообщении WM_SYSCOMMAND могут иметь любые значения. С учетом этого обстоятельства следует выбирать идентификатор для добавляемой в системное меню строки. Очевидно, что значение этого идентификатора должно быть больше 15 и не должно конфликтовать с идентификаторами других строк меню приложения.

6.6. Плавающее меню

При необходимости ваше приложение может создать временное плавающее меню , расположенное в любом месте экрана
Процедура создания меню выглядит следующим образом:

```
if(msg == WM_RBUTTONDOWN)
{
 HMENU hmenuPopup;
 POINT pt;

 pt = MAKEPOINT(lParam);
 ClientToScreen(hwnd, &pt);

 hmenuPopup = CreatePopupMenu();

 AppendMenu(hmenuPopup, MF_BYCOMMAND | MF_ENABLED,
    CM_FILENEW, "&New");
 AppendMenu(hmenuPopup, MF_BYCOMMAND | MF_ENABLED,
    CM_FILEOPEN, "&Open");
 AppendMenu(hmenuPopup, MF_BYCOMMAND | MF_ENABLED,
    CM_FILESAVE, "&Save");
```

```
  AppendMenu(hmenuPopup, MF_SEPARATOR, 0, 0);
  AppendMenu(hmenuPopup, MF_BYCOMMAND | MF_ENABLED,
      CM_FILEEXIT, "E&xit");

  TrackPopupMenu(hmenuPopup,
    TPM_CENTERALIGN | TPM_LEFTBUTTON,
    pt.x, pt.y, 0, hwndMain, NULL);

  DestroyMenu(hmenuPopup);
}
```

Обработчик сообщения WM_RBUTTONDOWN, которое приходит, если вы нажимаете правую клавишу мыши, прежде всего преобразует координаты курсора мыши в экранные. Для этого он вызывает функцию ClientToScreen .
Далее при помощи функции CreatePopupMenu создается пустое временное меню. Это меню наполняется обычным образом с помощью функции AppendMenu, но оно не привязывается к главному меню приложения или какому-либо другому меню. Вместо этого создается плавающее меню. Для этого идентификатор созданного и наполненного временного меню передается функции TrackPopupMenu :
TrackPopupMenu(hmenuPopup, TPM_CENTERALIGN | TPM_LEFTBUTTON, pt.x, pt.y, 0, hwndMain, NULL);
Эта функция выводит на экран плавающее меню и создает свой собственный цикл обработки сообщений, завершающий работу после выбора строки. Поэтому функция TrackPopupMenu не возвращает управление до тех пор, пока работа с меню не будет завершена либо выбором строки, либо отказом от выбора.
После этого созданное временное меню уничтожается: DestroyMenu(hmenuPopup);
Приведем прототип функции TrackPopupMenu :
BOOL WINAPI TrackPopupMenu(HMENU hmenu, UINT fuFlags, int x, int y, int nReserved, HWND hwnd, const RECT FAR* lprc);
Параметр hmenu должен содержать идентификатор отображаемого временного меню. Вы можете создать новое меню при помощи функции CreatePopupMenu или получить идентификатор существующего временного меню, вызвав функцию GetSubMenu.
Параметр fuFlags определяет расположение плавающего меню и клавиши мыши, с помощью которых должен выполняться выбор.
Для определения расположения меню вы можете указать один из трех флагов:

Флаг	Описание
TPM_CENTERALIGN	Центровка относительно координаты, заданной параметром x
TPM_LEFTALIGN	Выравнивание по левой границе относительно координаты, заданной параметром x
TPM_RIGHTALIGN	Выравнивание по правой границе относительно координаты, заданной параметром x

Дополнительно к перечисленным выше флагам вы можете указать один из двух флагов, определяющий клавишу мыши, предназначенную для выбора строки из плавающего меню:

Флаг	Описание
TPM_LEFTBUTTON	Левая клавиша мыши
TPM_RIGHTBUTTON	Правая клавиша мыши

Параметр nReserved зарезервирован, для совместимости со следующими версиями операционной системы Windows его значение должно быть равно 0.
Параметр hwnd задает идентификатор окна, которое получит сообщение WM_COMMAND после того как пользователь сделает выбор из плавающего меню. В операционной системе Windows версии 3.1 это сообщение попадает в функцию указанного окна после того как функция TrackPopupMenu возвратит управление. В версии 3.0 сообщение WM_COMMAND попадало в функцию окна до возврата управления функцией TrackPopupMenu.
Параметр lprc является указателем на структуру типа RECT, определяющую координаты прямоугольной области, в которой пользователь может выполнять выбор из меню. Если сделать щелчок мышью за пределами этой области, плавающее меню исчезнет с экрана. Такие действия эквивалентны отказу от выбора. Если задать для этого параметра значение NULL, размеры и расположение указанной выше прямоугольной области будут совпадать с размерами плавающего меню.

6.7. Использование плавающего меню в органе управления EDIT

Орган управления, созданный на базе предопределенного класса "edit", является простым редактором текста. В 13-м томе БСП Фроловых приводится интересный прием, позволяющий вызвать на экран плавающее меню простым нажатием правой клавиши мыши внутри окна редактирования. Причем меню окажется как раз около курсора мыши, так что для работы с меню вам не придется передвигать мышь на большое расстояние.
Для редактора текста внутри операционной системы Windows определена функция окна, выполняющая всю работу по редактированию текста, выделению фрагментов текста, копирование выделенного фрагмента в универсальный буфер обмена Clipboard и т. д. Когда вы устанавливаете курсор мыши в окно редактирования и нажимаете правую клавишу мыши, сообщение WM_RBUTTONDOWN попадает в функцию окна редактора текста.
Однако функция родительского окна, создавшая редактор текста, получает только сообщение с кодом WM_COMMAND, да и то только при выполнении определенных операций с текстом. Поэтому сколько бы вы не нажимали правую кнопку мыши в окне редактора текста, родительское окно об этом никогда не узнает.
Нам же надо не только определить момент, в который пользователь нажал правую кнопку мыши, но и узнать текущие координаты курсора мыши, чтобы создать плавающее меню в нужном месте экрана (недалеко от курсора мыши).
Так как встроенная функция окна, используемая редактором текста, перехватывает сообщение WM_RBUTTONDOWN и "не выпускает" его наружу, нам надо вставить собственный обработчик сообщений перед стандартным для класса окна "edit". Программный интерфейс Windows позволяет нам это сделать.
Определим в программе две переменные:
WNDPROC lpfnEditOldWndProc;

WNDPROC lpfnEditWndProc;
Эти переменные будут использоваться для хранения, соответственно, указателя на старую функцию окна редактора текста и указателя на новую функцию окна редактора текста.
Для получения адреса функции окна редактора текста мы воспользуемся функцией GetWindowLong :

```
lpfnEditOldWndProc =  (WNDPROC)GetWindowLong(hEdit, GWL_WNDPROC);
```

Если в качестве второго параметра этой функции передать константу GWL_WNDPROC , функция вернет адрес функции окна, идентификатор которого задан первым параметром. Возвращенное функцией GetWindowLong значение мы сохраним в переменной lpfnEditOldWndProc, так как наша функция окна, встроенная до стандартной, после выполнения своей задачи должна вызвать стандартную функцию окна (иначе редактор текста не будет работать).
Итак, адрес старой функции окна мы узнали. Теперь надо подготовить новую функцию окна, которая, если пользователь нажмет на правую клавишу мыши, будет выводить на экран плавающее меню. Вот эта функция:

```
// Новая функция окна для редактора текста
LRESULT CALLBACK _export
EditWndProc(HWND hwnd, UINT msg, WPARAM wParam,  LPARAM lParam)
{ // Если в окне редактора текста пользователь нажал правую клавишу мыши, выводим      //в позиции курсора мыши плавающее меню
  if(msg == WM_RBUTTONDOWN)
  {  HMENU hmenuPopup;    POINT pt;
    // Преобразуем координаты курсора мыши в экранные
    pt = MAKEPOINT(lParam);   ClientToScreen(hwnd, &pt);
    hmenuPopup = CreatePopupMenu();// Создаем пустое временное меню
    // Заполняем временное меню
    AppendMenu(hmenuPopup, MF_BYCOMMAND | MF_ENABLED, CM_FILENEW,
"&New");
    AppendMenu(hmenuPopup, MF_BYCOMMAND | MF_ENABLED, CM_FILEOPEN,
"&Open");
    AppendMenu(hmenuPopup, MF_BYCOMMAND | MF_ENABLED, CM_FILESAVE,
"&Save");
    AppendMenu(hmenuPopup, MF_SEPARATOR, 0, 0);
    AppendMenu(hmenuPopup, MF_BYCOMMAND | MF_ENABLED,
        CM_FILEEXIT, "E&xit");
    // Выводим плавающее меню в позиции курсора мыши
    TrackPopupMenu(hmenuPopup, TPM_CENTERALIGN | TPM_LEFTBUTTON,
      pt.x, pt.y, 0, hwndMain, NULL);
     DestroyMenu(hmenuPopup); // Удаляем временное меню
  }
  // Вызываем старую функцию окна редактора текста
  return CallWindowProc(lpfnEditOldWndProc, hwnd, msg, wParam, lParam);
}
```

Обратите внимание, что после завершения работы новая функция окна вызывает старую функцию окна. Так как ваше приложение не может вызывать функцию окна

непосредственно, мы вызываем старую функцию окна при помощи функции CallWindowProc .
Таким образом, мы сделали то, что нам нужно - новая функция окна обрабатывает сообщение от правой клавиши мыши, выводит плавающее меню и затем вызывает стандартную функцию окна текстового редактора.
Однако для того чтобы вместо стандартной функции окна вызывалась наша, ее необходимо подключить при помощи функции SetWindowLong :

```
lpfnEditWndProc = (WNDPROC)MakeProcInstance((FARPROC)EditWndProc,hInst);
SetWindowLong(hEdit, GWL_WNDPROC,  (LONG)lpfnEditWndProc);
```

Перед вызовом функции мы создаем переходник и сохраняем его адрес в переменной lpfnEditWndProc. Сразу после возвращения управления из функции SetWindowLong наша новая функция окна включается в работу, пропуская через себя все сообщения, предназначенные для стандартной функции окна редактора текста.
Описанная выше методика обычно используется в тех случаях, когда нужно изменить поведение стандартного органа управления или любого стандартного окна Windows с известным идентификатором (зная который можно "добраться" до функции окна).

6.8. Акселераторы

Для ускорения доступа к строкам меню при помощи клавиатуры (а также для назначения тех или иных функций, не связанных с меню, комбинациям клавиш), используется так называемая таблица акселераторов (accelerator table).
Таблица акселераторов находится в ресурсах приложения и определяет соответствие между комбинациями клавиш и значением параметра wParam сообщения WM_COMMAND, попадающего в функцию окна, когда вы нажимаете эти комбинации клавиш.
Например, вы можете определить, что комбинации клавиш <Control+Insert> соответствует значение wParam, равное CM_EDITCUT. В этом случае если нажать указанную выше комбинацию клавиш, в функцию окна попадет сообщение WM_COMMAND с параметром wParam, равным CM_EDITCUT.
Обычно комбинации клавиш, используемые для ускоренного выбора (или просто акселераторы) обозначаются в правом столбце меню .
Однако такое обозначение, сделанное при помощи символа \t в шаблоне меню не распознается Windows, а служит лишь для удобства пользователя. Для того чтобы комбинация клавиш стала работать как акселератор, она должна быть описана в таблице акселераторов. Кроме этого, приложение должно загрузить таблицу акселераторов из ресурсов приложения и изменить цикл обработки сообщений.

Описание таблицы акселераторов

Таблица акселераторов определяется в файле описания ресурсов приложения в следующем виде:

```
<Id> ACCELERATORS
BEGIN
.......
.......
```

```
.....
END
```
Для ссылки на таблицу акселераторов используется идентификатор Id, который не должен совпадать с идентификаторами других ресурсов приложения, таких как строки, диалоги и т. д.
Между операторами BEGIN и END располагаются строки описания акселераторов. Они имеют следующий формат (в квадратных скобках указаны необязательные параметры): Key, AccId, [KeyType[,]] [NOINVERT] [ALT] [SHIFT] [CONTROL]
Поле Key определяет клавишу, которая будет использована для создания акселератора. Вы можете использовать символ в коде ASCII, заключенный в двойные кавычки (например, "F"), комбинацию символа ASCII со знаком ^ (например, "^S", что соответствует комбинации клавиш <Control+S>), ASCII-код клавиши в виде целого значения, или виртуальный код клавиши (в символьном или цифровом виде).
Поле AccId соответствует значению параметра wParam сообщения WM_COMMAND, которое попадет в функцию окна при использовании акселератора.
Поле KeyTab может принимать значения ASCII или VIRTKEY. В первом случае поле Key определяет клавишу с использованием кода ASCII, во втором - с использованием кода виртуальной клавиши. По умолчанию используется значение ASCII.
Если указан параметр NOINVERT, при использовании акселератора соответствующая строка меню не выделяется. По умолчанию строка меню выделяется инвертированием цвета.
Если поле KeyTab содержит значение VIRTKEY, можно указывать параметры ALT, SHIFT или CONTROL. В этом случае для акселератора используется комбинация клавиши, указанной параметром Key, и клавиши ALT, SHIFT или CONTROL, соответственно.
Приведем пример описания таблицы акселераторов из приложения SMARTPAD:
```
APP_ACCELERATORS ACCELERATORS
BEGIN
  "N", CM_FILENEW,        VIRTKEY, CONTROL
  "S", CM_FILESAVE,       VIRTKEY, CONTROL
  "O", CM_FILEOPEN,       VIRTKEY, CONTROL
  "Z", CM_EDITUNDO,       VIRTKEY, CONTROL
  "X", CM_EDITCUT,        VIRTKEY, CONTROL
  "C", CM_EDITCOPY,       VIRTKEY, CONTROL
  "V", CM_EDITPASTE,      VIRTKEY, CONTROL
  VK_DELETE, CM_EDITCLEAR, VIRTKEY, CONTROL
  VK_F1, CM_HELPINDEX,    VIRTKEY
END
```
Здесь описана таблица акселераторов APP_ACCELERATORS, в которой определены девять акселераторов, т. е. девять комбинаций клавиш ускоренного выбора.
Для того чтобы акселератор, состоящий из комбинации символьной клавиши (такой, как "N") и клавиши <Control>, работал вне зависимости от состояния клавиши <Caps Lock>, мы использовали виртуальные коды. Если бы мы использовали коды ASCII, наш акселератор активизировался бы только при использовании заглавных

букв (мы могли бы указать строчные буквы, например, "n", в этом случае для активизации акселератора следовало бы использовать строчные буквы).
Из-за того что клавиша <Caps Lock> может находиться в любом состоянии, лучше работать с виртуальными кодами клавиш, не зависящих от того, являются буквы строчными или прописными.

Загрузка таблицы акселераторов

Для загрузки таблицы акселераторов следует использовать функцию LoadAccelerators : HACCEL WINAPI LoadAccelerators(HINSTANCE hInst, LPCSTR lpszTableName);
Параметр hInst определяет идентификатор копии приложения, из ресурсов которого будет загружена таблица акселераторов.
Параметр lpszTableName является указателем на строку, содержащую идентификатор таблицы акселераторов. Если для идентификации ресурса используется целое значение, оно должно быть преобразовано макрокомандой MAKEINTRESOURCE.
Функция LoadAccelerators возвращает идентификатор загруженной таблицы акселераторов или NULL при ошибке.
Загруженная таблица акселераторов автоматически уничтожается при завершении работы приложения.

Изменения в цикле обработки сообщений

Для использования акселераторов цикл обработки сообщений должен выглядеть следующим образом:

```
while(GetMessage(&msg, 0, 0, 0))
{
 if(!haccel || !TranslateAccelerator(hwnd, haccel, &msg))
 {
  TranslateMessage(&msg);
  DispatchMessage(&msg);
 }
}
```

В этом фрагменте кода переменная haccel содержит идентификатор загруженной таблицы акселераторов. Если идентификатор не равен NULL, вызывается функция TranslateAccelerator . Эта функция ищет в очереди сообщений сообщения от клавиатуры, соответствующие определенным в ресурсах приложения акселераторам, преобразуя такие сообщения в сообщения WM_COMMAND и WM_SYSCOMMAND (если сообщение соответствует системному меню), передаваемые непосредственно в функцию окна, минуя очередь сообщений приложения .
Содержимое параметра wParam в последних двух сообщениях равно идентификатору, указанному в таблице акселераторов для данной комбинации клавиш.
Старшее слово параметр lParam содержит 1 для сообщений, которые пришли от акселераторов и 0 для сообщений, которые пришли от меню.
Приведем прототип функции TranslateAccelerator:
int WINAPI TranslateAccelerator(HWND hwnd, HACCEL haccel, MSG FAR* lpmsg);
Параметр hwnd определяет идентификатор окна, для которого выполняется преобразование клавиатурных сообщений.

Параметр haccel должен содержать идентификатор загруженной при помощи функции LoadAccelerators таблицы акселераторов.
Последний параметр lpmsg является указателем на структуру типа MSG, в которую должно быть записано обрабатываемое сообщение.
Если функция TranslateAccelerator выполнила преобразование сообщения, она возвращает ненулевое значение. В противном случае возвращается 0. Обработанное сообщение не следует передавать функциям TranslateMessage и DispatchMessage.

6.9. Орган управления TOOLBAR

В современных приложениях Windows широко используется еще один важный элемент пользовательского интерфейса, облегчающий работу с меню (и в некоторых случаях даже полностью заменяющий меню). Речь идет об органе управления, который часто называется Toolbar.
Toolbar с точки зрения пользователя представляет собой ни что иное, как набор кнопок с нарисованными на их поверхности пиктограммами. Каждая такая кнопка соответствует определенной строке в том или ином временном меню приложения.
С точки зрения программиста орган управления Toolbar может представлять собой отдельный объект в виде дочернего окна с расположенными на нем кнопками или совокупность кнопок, созданных на поверхности главного окна приложения.

Подготовка изображений для кнопок.

Проще всего взять файл toolbar.bmp в среде разработки MS Visual C++ и при необходимости заменить пиктограммы на кнопках своими. В любом случае это изображение придется добавить в файл ресурсов например так:
IDB_TBBITMAP BITMAP “toolbar.bmp”

Описание кнопок

Далее надо создать массив структур TBBUTTON :

```
typedef struct _TBBUTTON { \\ tbb
   int iBitmap;          //номер кнопки
   int idCommand;    //Идентификатор для сообщения WM_COMMAND
   BYTE fsState;       //Флаг исходного состояния кнопки
   BYTE fsStyle;       //Стиль кнопки
   DWORD dwData;
   int iString;           //Номер текстовой строки для надписи на кнопке
} TBBUTTON
```

Образец массива кнопочных структур может выглядеть так:

```
TBBUTTON tbButtons[] =
{
  { 0, ID_FILE_NEW,  TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0},
  { 1, ID_FILE_OPEN, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0},
  { 2, ID_FILE_SAVE, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0},
  { 0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0L, 0},
  { 3, ID_EDIT_CUT,   TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0},
  { 4, ID_EDIT_COPY,  TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0},
```

```
  { 5, ID_EDIT_PASTE, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0},
  { 0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0L, 0},
  { 6, ID_FILE_PRINT, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0},
  { 0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0L, 0},
  { 7, ID_HELP_ABOUT, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0}
};
```

Вызов функции создания окна Toolbar

Покажем на примере:

```
hwndTb = CreateToolbarEx(hWnd,
   WS_CHILD | WS_BORDER | WS_VISIBLE | TBSTYLE_TOOLTIPS |
    CCS_ADJUSTABLE,
   IDT_TOOLBAR,          // идентификатор органа Toolbar
   8,                    // количество пиктограмм
   hInst,                // идентификатор приложения
   IDB_TBBITMAP,         // идентификатор битового изображения кнопок
   (LPCTBBUTTON)&tbButtons,// адрес описания кнопок
   11,                   // количество кнопок
   16,16,                // ширина и высота кнопок
   16,16,                // ширина и высота пиктограмм
   sizeof(TBBUTTON));    // размер структуры в байтах
```

Обработка извещений от Toolbar

Этот орган посылает родительскому окну сообщения WM_COMMAND иWM_NOTIFY.

На обработке WM_COMMAND нет смысла останавливаться еще раз. Рассмотрим обработку WM_NITIFY – wParam содержит идентификатор органа управления (в нашем случае Toolbar), а lParam – указатель на структуру TBNOTIFY, первым полем которой является структура NMHDR:

```
typedef struct { \\ tbn
   NMHDR hdr;
   int iItem;                //Номер кнопки пославшей сообщение
   TBBUTTON tbButton; //структура описания кнопки
   int cchText;              //Длина текстовой строки
   LPTSTR pszText;           //Адрес строки
} TBNOTIFY, FAR* LPTBNOTIFY;

typedef struct tagNMHDR {
   HWND hwndFrom; //окно
   UINT idFrom;          //орган
   UINT code;            //код сообщения
} NMHDR;
```

Приводим комментированный фрагмент кода, выполняющий обработку извещений от Toolbar; остальную информацию об извещениях вы можете извлечь из Win32 OnLine Help компилятора Borland или MSDN студииразработчика MS Visual C++:

```
LRESULT WndProc_OnNotify(HWND hWnd, int idFrom, NMHDR* pnmhdr)
{  LPTOOLTIPTEXT lpToolTipText;  LPTBNOTIFY lptbn;  int nItem;
  static CHAR szBuf[128];
```

```
 switch(pnmhdr->code)
 {// Если получили сообщение от ToolTips, загружаем из ресурсов
  // соответствующую текстовую строку
  case TTN_NEEDTEXT:
    lpToolTipText = (LPTOOLTIPTEXT)pnmhdr;
         LoadString(hInst, lpToolTipText->hdr.idFrom,
           szBuf, sizeof(szBuf));
         lpToolTipText->lpszText = szBuf;
    break;
// Возвращаем окну Toolbar х-ки кнопки, с номером, заданным в lptbn->iItem
  case TBN_GETBUTTONINFO:
    lptbn = (LPTBNOTIFY)pnmhdr;
         nItem = lptbn->iItem;
         lptbn->tbButton.iBitmap   = tbButtons[nItem].iBitmap;
         lptbn->tbButton.idCommand = tbButtons[nItem].idCommand;
         lptbn->tbButton.fsState   = tbButtons[nItem].fsState;
         lptbn->tbButton.fsStyle   = tbButtons[nItem].fsStyle;
         lptbn->tbButton.dwData    = tbButtons[nItem].dwData;
         lptbn->tbButton.iString   = tbButtons[nItem].iString;
// Если запрашиваются х-ки несуществующей  кнопки, возвращаем FALSE
    return ((nItem < sizeof(tbButtons)/sizeof(tbButtons[0]))? TRUE : FALSE);
    break;
// Разрешаем удаление любой кнопки, кроме самой первой
  case TBN_QUERYDELETE:
    lptbn = (LPTBNOTIFY)pnmhdr;    nItem = lptbn->iItem;
    return (nItem == 0)? FALSE : TRUE;
    break;
// Разрешаем вставку любой кнопки, кроме самой первой
  case TBN_QUERYINSERT:
    lptbn = (LPTBNOTIFY)pnmhdr;    nItem = lptbn->iItem;
    return (nItem == 0)? FALSE : TRUE;
    break;
 // В ответ на завершение операции перемещения  перерисовываем Toolbar
  case TBN_TOOLBARCHANGE:
    SendMessage(hwndTb, TB_AUTOSIZE, 0L, 0L);
    return TRUE;
         break;
   default:   break;
 }  return FALSE;}
```

6.10.Орган управления Statusbar (стрка статуса).

Создается либо с помощью функции CreateWindowEx на основе класса окна STATUSCLASSNAME, либо специальной функцией CreateStatusWindow:

```
HWND CreateStatusWindow(

  LONG style,            //Стиль – WS_VISIBLE | WS_CHILD
  LPCTSTR lpszText, // текст
```

```
   HWND hwndParent, //Идентификатор родительского окна
   UINT wID         //Идентификатор Statusbar
  );
```
В области Statusbar можно выводить текст или графику. Дпя этого используется обычная функция посылки сообщений SendMessage с кодом SB_TEXT:
```
SendMessage(hwndSb,SB_SETTEXT,0,(LPARAM)"Выводимый текст");
```
При выводе графики номер области вывода в wParam надо скомбинировать с константой SBT_OWNERDRAW, что даст возможность родительскому окну нарисовать там любое изображение или вывести текст любым шрифтом:
```
hSbBmp=LoadBitMap(hInst,IDB_BITMAPSB);
SendMessage(hwndSb,SB_SETTEXT,0 | SBT_OWERDRAW,(LPARAM)hSbBmp);
```
Фрагмент кода, рисующий в строке статуса маленькое графическое изображение при обработке сообщения WM_DRAWITEM:
```
void WndProc_OnDrawItem(HWND hwnd,  const DRAWITEMSTRUCT * lpDrawItem)
{  // Необходимо выполнить перерисовку области Statusbar,
  // которая была определена как SBT_OWNERDRAW
  if(lpDrawItem->CtlID == IDS_STATUSBAR)
  { LPDRAWITEMSTRUCT lpDis;
    HDC hdcMem;
    HBITMAP hbmOld;
    BITMAP bm;
    // Рисуем битовое изображение внутри области
    lpDis = (LPDRAWITEMSTRUCT)lpDrawItem;
    hdcMem = CreateCompatibleDC(lpDis->hDC);
    hbmOld = SelectObject(hdcMem, hSbLogoBmp);
    GetObject(hSbLogoBmp, sizeof(bm), &bm);
    BitBlt(lpDis->hDC,
      lpDis->rcItem.left, lpDis->rcItem.top,
      bm.bmWidth, bm.bmHeight,
      hdcMem, 0, 0, SRCCOPY);
    SelectObject(hdcMem, hbmOld);
    DeleteDC(hdcMem);
  }
  return FORWARD_WM_DRAWITEM(hwnd, lpDrawItem, DefWindowProc);}
```
С набором сообщений для строки статуса вы можете ознакомиться через систему помощи вашей инструментальной среды разработки.

7.Панели диалога

Обычно органы управления создаются не на поверхности главного окна - для объединения органов управления используются временные (pop-up) окна, созданные на базе предопределенного внутри Windows класса окон - класса диалоговых панелей (dialog window class). На поверхности такого окна располагаются дочерние окна - органы управления. Функция окна для класса диалоговых панелей, определенная в Windows, выполняет практически всю работу, необходимую для организации взаимодействия органов управления с приложением.

Диалоговые панели значительно упрощают использование органов управления, так как функция окна, соответствующая классу диалоговых панелей и расположенная внутри Windows, обеспечивает как взаимодействие органов управления между собой, так и их взаимодействие с приложением. В частности, эта функция обеспечивает передачу фокуса ввода от одного органа управления к другому при помощи клавиши <Tab> и клавиш перемещения курсора <Up> и <Down>, выполняет обработку сообщений от клавиш <Enter> и <Esc>.
Что же касается расположения органов управления на поверхности диалоговой панели, то для этого можно использовать три способа.
Первый способ предполагает включение в файл ресурсов приложения текстового описания шаблона диалоговой панели. Это описание можно создать при помощи любого текстового редактора:

```
DIALOG_BOX DIALOG 25, 34, 152, 66
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Приложение DIALOG"

BEGIN
  CTEXT "Microsoft Windows Application\n"    "Приложение DIALOG",
    -1, 28, 9, 116, 26, WS_CHILD | WS_VISIBLE | WS_GROUP  ICON "APPICON", -1,
6, 14, 16, 16, WS_CHILD | WS_VISIBLE  DEFPUSHBUTTON "OK", IDOK, 56, 43, 36,
14,
    WS_CHILD | WS_VISIBLE | WS_TABSTOP
END
```

В приведенном описании определен шаблон диалоговой панели с идентификатором DIALOG_BOX . Расположение и размеры диалоговой панели определяются цифрами, стоящими после оператора DIALOG.
Вторая строка описания (оператор STYLE) предназначена для определения стиля временного окна, на поверхности которого будут расположены органы управления. Далее следует описание заголовка диалоговой панели (оператор CAPTION).
Между строками BEGIN и END находится описание органов управления и пиктограмм. В нашем случае это статический орган управления (оператор CTEXT), пиктограмма (ICON) и кнопка (DEFPUSHBUTTON). Для каждого органа управления приводится вся необходимая информация - заголовок, стиль, расположение и размеры.
Так как шаблон диалоговой панели описан в файле ресурсов, вы можете влиять на внешний вид и поведение диалоговой панели, изменяя файл ресурсов, но не исходный текст приложения. При использовании диалоговых панелей задача создания и размещения (а также взаимодействия) органов управления сильно упрощается. Отделяя описание органов управления от исходного текста приложения, вы не только упрощаете процесс программирования, но и получаете возможность создать локализованную версию приложения для любого национального языка без внесения изменений в исходные тексты приложения.
Второй способ размещения органов управления на поверхности диалоговой панели предполагает использование специального редактора ресурсов. Эти редакторы позволяют нарисовать диалоговую панель и сохранить ее текстовое описание в файле ресурсов приложения.

Такой подход в создании приложений носит зачатки визуального программирования, когда внешний вид и поведение приложения определяется с помощью специальных графических средств проектирования без традиционного программирования на каком-либо алгоритмическом языке.
Третий способ предполагает создание шаблона диалоговой панели в памяти во время работы приложения. Этот способ используется редко, обычно только в тех случаях, когда внешний вид диалоговой панели нельзя определить на этапе создания приложения.
В операционной системе Windows есть встроенные диалоговые панели, реализованные в виде библиотеки динамической загрузки commdlg.dll. Приложения могут вызывать стандартные диалоговые панели для работы с файлами , для выбора цветов и шрифтов, для работы с принтерами и текстовыми строками (поиск и замена строк в текстовых редакторах и текстовых процессорах).
Функция MessageBox использует встроенные диалоговые панели, в которых могут находиться пиктограмма, статический орган управления (для отображения текста), а также одна или несколько кнопок.

7.1. Создание диалоговой панели

Диалоговая панель обычно представляет собой временное (pop-up) окно, хотя допустимо использовать и перекрывающиеся (overlapped) окна. Для создания диалоговой панели вам не требуется вызывать функцию CreateWindow, так как в программном интерфейсе Windows определены функции, специально предназначенные для создания диалоговых панелей.
Разумеется, перед вызовом этих функций необходимо создать шаблон диалоговой панели. Это можно сделать, использовав один из описанных выше способов.
Помимо шаблона, **перед созданием диалоговой панели вам следует определить специальную функцию диалога, в которую будут поступать сообщения от функции окна диалоговой панели (которые, в свою очередь, поступают туда от органов управления)**. Функция диалога похожа на функцию таймера (получающую сообщения от таймера). Так же, как и функция таймера (а также функция окна), функция диалога является функцией обратного вызова .
Итак, для создания диалоговой панели вы должны предпринять следующие действия:
создать шаблон диалога;
определить функцию диалога;
вызвать одну из функций создания диалога.

Создание шаблона диалога

Для создания шаблона диалога лучше всего воспользоваться редактором ресурсов.

Использование текстового редактора

Вы можете создать диалоговую панель без применения редакторов диалога, создав описание шаблона при помощи любого текстового редактора, сохраняющего текст без атрибутов форматирования.

Описание шаблона

Описание шаблона имеет следующий вид:

```
nameID DIALOG [load] [mem] x, y, width, height
CAPTION "Заголовок диалоговой панели"
STYLE   Стиль
 BEGIN     . . . . . .
END
```

В этом описании nameID используется для идентификации шаблона диалоговой панели и может указываться либо в виде текстовой строки, либо в виде числа .

Параметр load - необязательный. Он используется для определения момента загрузки диалоговой панели в память. Если этот параметр указан как PRELOAD, диалоговая панель загружается в память сразу после запуска приложения. По умолчанию используется значение LOADONCALL, при использовании которого загрузка шаблона в память происходит только при отображении диалоговой панели.

Параметр mem также необязательный. Он влияет на тип выделяемой для хранения шаблона памяти и может указываться как FIXED (ресурс всегда остается в фиксированной области памяти), MOVEABLE (при необходимости ресурс может перемещаться в памяти, это значение используется по умолчанию) или DISCARDABLE (если ресурс больше не нужен, занимаемая им память может быть использована для других задач). Значение DISCARDABLE может использоваться вместе со значением MOVEABLE.

Параметры x и y определяют, соответственно, x-координату левой границы диалоговой панели и y-координату верхней стороны диалоговой панели. Координаты могут принимать значения от 0 до 65535.

Параметры width и height определяют, соответственно, ширину и высоту диалоговой панели. Эти параметры могут находиться в диапазоне от 1 до 65535.

Для описания шаблонов диалоговых панелей используется специальная координатная система, в которой размер единицы длины в пикселах зависит от размера системного шрифта. Такая координатная система позволяет создавать диалоговые панели, размер которых не зависит от режима работы видеоадаптера. Это возможно благодаря тому, что размер системного шрифта в пикселах зависит от разрешения - в режиме 800x600 точек размеры системного шрифта больше, чем, например, в режиме 640x480 точек.

Одна единица длины в координатной системе, используемой при описании ширины элементов шаблонов диалоговых панелей, составляет четверть средней ширины символов системного шрифта, а при описании высоты (или вертикальных размеров) - восьмую часть высоты символов системного шрифта. Так как высота символов системного шрифта примерно в два раза больше средней ширины этих символов, единица длины в этой системе координат имеет одинаковый размер по вертикали и горизонтали. Эта единица называется диалоговая единица (dialog unit).

Размер единицы измерения dialog unit можно получить при помощи функции GetDialogBaseUnits: DWORD WINAPI GetDialogBaseUnits(void);

Младшее слово возвращаемого значения представляет собой ширину в пикселях диалоговой единицы длины, старшее - высоту.

Оператор CAPTION предназначен для определения заголовка диалоговой панели.

Оператор STYLE используется для назначения стиля окну диалоговой панели. В качестве стиля вы можете использовать комбинацию символических имен, определен-

ных в файле windows.h и имеющих префикс WS_. Специально для диалоговых панелей в этом файле определены несколько констант с префиксом DS_.

Имя константы	Описание
DS_LOCALEDIT	При использовании этого стиля редакторы текста, созданные в диалоговой панели, будут использовать память в сегменте данных приложения. В этом случае можно использовать сообщения EM_GETHANDLE и EM_SETHANDLE
DS_MODALFRAME	Создается модальная диалоговая панель (см. ниже)
DS_NOIDLEMSG	Если этот стиль не указан, когда диалоговая панель переходит в видимое состояние (отображается), Windows посылает родительскому окну (создавшему диалоговую панель), сообщение WM_ENTERIDLE
DS_SYSMODAL	Создается системная модальная диалоговая панель

Для создания стандартной диалоговой панели используются стили WS_POPUP, WS_BORDER, WS_SYSMENU, WS_CAPTION, DS_MODALFRAME. Если нужно создать диалоговую панель с рамкой, но без заголовка, используется стиль WS_DLGFRAME.

Диалоговые панели бывают трех типов: **модальные, системные модальные, и немодальные**.

При выводе на экран модальной диалоговой панели работа приложения приостанавливается. Функции главного окна приложения и всех дочерних окон перестают получать сообщения от мыши и клавиатуры. Все эти сообщения попадают в временное (pop-up) окно диалоговой панели. Когда работа пользователя с диалоговой панелью будет завершена, главное окно приложения и его дочерние окна будут разблокированы. Заметьте, что **диалоговая панель не должна создаваться как дочернее окно** - в этом случае при активизации диалоговой панели она будет заблокирована наряду с остальными дочерними окнами и приложение "зависнет".

Модальная диалоговая панель, тем не менее, позволяет пользователю переключиться на работу с другими приложениями. Если вам требуется запретить такое переключение, используйте системные модальные диалоговые панели.

Немодальная диалоговая панель не блокирует работу основного окна приложения и его дочерних окон. Вы можете работать как с диалоговой панелью, так и с окном приложения. Разумеется, вам также доступна возможность переключения на другие запущенные приложения.

Помимо операторов STYLE и CAPTION, описание шаблона может содержать операторы CLASS и FONT. Оператор CLASS используется в тех случаях, когда диалоговая панель использует свой собственный класс, а не тот, который определен для диалоговых панелей операционной системой Windows: CLASS "PrivateDlgClass"

Мы не будем рассматривать диалоговые панели на базе собственных классов.

Перед созданием диалоговой панели с собственным классом этот класс должен быть зарегистрирован. При этом в структуре WNDCLASS, используемой для регистрации, поле cbWndExtra должно иметь значение DLGWINDOWEXTRA.
Оператор FONT позволяет задать шрифт, с использованием которого Windows будет писать текст в диалоговой панели: FONT 10, "MS Serif"
Первый параметр оператора FONT указывает размер шрифта в пунктах, второй - название шрифта, определенного в файле win.ini. Отметим, что единственный шрифт, присутствие которого гарантируется - это системный шрифт. Все остальные шрифты можно отключить при помощи приложения Control Panel. Указывая шрифт, отличный от системного, вы не можете быть уверены, что этот шрифт будет установлен у пользователя. В этом случае перед выводом диалоговой панели имеет смысл убедиться в том, что в системе зарегистрирован требуемый шрифт (о том, как это сделать, вы узнаете позже). Если нужный шрифт не установлен, можно выдать предупреждающее сообщение.
Описание всех органов управления, расположенных на поверхности диалоговой панели, должно находиться между строками BEGIN и END.
Для описания органов управления используются три формата строк.
Первый формат можно использовать для всех органов управления, кроме списков, редакторов текста и полосы просмотра:

CtlType "Текст", ID, x, y, width, height [,style]

Вместо CtlType в приведенной выше строке должно находиться обозначение органа управления.Параметр "Текст" определяет текст, который будет написан на органе управления.Параметр ID - идентификатор органа управления. Этот идентификатор передается вместе с сообщением WM_CONTROL. Параметры x и y определяют координаты органа управления относительно левого верхнего угла диалоговой панели. Используется единица длины dialog unit. Параметры width и height определяют, соответственно, ширину и высоту органа управления в единицах длины dialog unit. Параметр style определяет стиль органа управления (необязательный параметр). Это тот самый стиль, который указывается при вызове функции CreateWindow.
Приведем список обозначений органов управления и возможных стилей для первого формата.

Обозначение органа управления	Класс окна	Описание и стиль, используемый по умолчанию
CHECHBOX	button	Переключатель в виде прямоугольника BS_CHECKBOX, WS_TABSTOP
CTEXT	static	Строка текста, выровненная по центру SS_CENTER, WS_GROUP
DEFPUSHBUTTON	button	Кнопка, выбираемая в диалоговой панели по умолчанию BS_DEFPUSHBUTTON, WS_TABSTOP
GROUPBOX	button	Прямоугольник, объединяющий группу органов управления BS_GROUPBOX
ICON	static	ПиктограммаSS_ICON Параметры width, height и style можно не указывать
LTEXT	static	Строка текста, выровненная по левой границе

		органа управления SS_LEFT, WS_GROUP
PUSHBUTTON	button	КнопкаBS_PUSHBUTTON, WS_TABSTOP
RADIOBUTTON	button	Переключатель в виде кружка (радиопереключатель) BS_RADIOBUTTON, WS_TABSTOP
RTEXT	static	Строка текста, выровненная по правой границе органа управления SS_RIGHT, WS_GROUP

Стили WS_TABSTOP и WS_GROUP будут описаны позже.
Второй формат используется для описания списков, редакторов текста и полос просмотра: CtlType ID, x, y, width, height [,style]
В этом формате нет параметра "Текст", остальные параметры используются так же, как и в первом формате. Список обозначений органов управления и возможных стилей для второго формата:

Обозначение органа управления	Класс окна	Описание и стиль, используемый по умолчанию
COMBOBOX	combobox	Список с окном редактирования CBS_SIMPLE, WS_TABSTOP
LISTBOX	listbox	Список LBS_NOTIFY, WS_BORDER
EDITTEX	edit	Редактор текста ES_LEFT, WS_BORDER, WS_TABSTOP
SCROLLBARS	scrollbar	Полоса просмотраSBS_HORZ

Третий формат описания органов управления наиболее универсальный:
CONTROL "Текст", ID, class, style, x, y, width, height
Этот формат позволяет описать орган управления, принадлежащий классу class, который указывается в виде строки символов. Вы можете использовать третий формат для описания предопределенных классов органов управления, таких как "button", "combobox", "edit", "listbox", "scrollbar", "static". Данный формат описания можно использовать для любых органов управления.

Функция диалога

Перед созданием диалоговой панели, помимо шаблона диалога, программисту необходимо подготовить функцию диалога, предназначенную для обработки сообщений, поступающих от диалоговой панели. Эта функция должна быть описана следующим образом:
BOOL CALLBACK DlgProc (HWND hdlg, UINT msg, WPARAM wParam, LPARAM lParam);
Вы можете выбрать для функции диалога любое имя.
Параметры функции диалога напоминают параметры функции окна. Все они, за исключением первого, имеют аналогичное назначение. Через первый параметр функции диалога передается идентификатор диалога hdlg, а не идентификатор окна hwnd. В отличие от функции окна, функция диалога не должна вызывать функцию DefWindowProc для тех сообщений, которые она не желает обрабатывать. Если функция диалога обрабатывает сообщение, она должна вернуть значение TRUE, а если нет - FALSE. Функция диалога не обрабатывает сообщения WM_CREATE, WM_PAINT, WM_DESTROY. При инициализации диалога в функцию диалога вме-

сто сообщения WM_CREATE передается сообщение WM_INITDIALOG. Как правило, функция диалога всегда обрабатывает сообщения WM_INITDIALOG и WM_COMMAND. Сообщение WM_INITDIALOG использует параметры wParam и lParam. Параметр wParam содержит идентификатор органа управления, который первым получит фокус ввода после отображения диалоговой панели. Это первый орган управления, описанный в шаблоне диалога со стилем WM_TABSTOP. Параметр lParam содержит значение, передаваемое приложением при создании диалоговой панели. Если в ответ на сообщение WM_INITDIALOG функция диалога возвращает значение TRUE, после создания диалоговой панели фокус ввода передается органу управления, идентификатор которого был записан в параметре wParam.
Если при инициализации диалоговой панели обработчик сообщения WM_INITDIALOG устанавливает фокус ввода на другой орган управления (вызывая функцию SetFocus), функция диалога должна вернуть значение FALSE.
Сообщение WM_COMMAND, поступающее в функцию диалога, передает сообщения или извещения от органов управления, расположенных в диалоговой панели. Крое этого, функция диалога может получить это сообщение с параметром wParam, равным константам IDOK и IDCANCEL, описанным в файле windows.h.
Сообщение с параметром IDOK поступает в функцию диалога в том случае, если пользователь нажал клавишу <Enter> в момент, когда ни одна из кнопок, расположенных в диалоговой панели, не имеет фокус ввода, и ни одна из кнопок не имеет стиль WS_DEFPUSHBUTTON. Если в диалоговой панели есть кнопка со стилем WS_DEFPUSHBUTTON, в описанной ситуации в функцию диалога поступает сообщение WM_COMMAND с параметром wParam, равным идентификатору этой кнопки.
Сообщение с параметром IDCANCEL появится тогда, когда пользовательзакроет диалоговую панель с помощью системного меню или клавиши <Esc>.
Обычно в диалоговой панели всегда создается одна клавиша, имеющая стиль WS_DEFPUSHBUTTON. Как правило, на этой клавише пишется слово "OK" и она используется для нормального завершения работы диалоговой панели. Для этой клавиши имеет смысл использовать идентификатор IDOK.
Еще одна клавиша, присутствующая практически во всех диалоговых панелях, имеет надпись "Cancel" и используется для отмены диалоговой панели. Если определить идентификатор этой клавиши как IDCANCEL, вы сможете использовать единый обработчик сообщения для отмены диалоговой панели при помощи кнопки и при помощи системного меню или клавиши <Esc>.

Функции для создания диалоговой панели

В программном интерфейсе Windows определены восемь функций, предназначенных для создания модальных и немодальных диалоговых панелей.
Для создания модальной диалоговой панели чаще всего используется функция DialogBox: int WINAPI DialogBox(HINSTANCE hInstance, LPCSTR lpszTemplate, HWND hwndOwner, DLGPROC dlgprc);
Через параметр hInstance необходимо передать идентификатор текущей копии приложения. Параметр lpszTemplate представляет собой указатель на строку имени шаблона, указанном в операторе DIALOG текстового описания шаблона. Параметр

hwndOwner - идентификатор окна, создавшего диалоговую панель. Последний параметр, dlgprc, представляет собой адрес функции диалога.
Если при создании диалоговой панели ей необходимо передать параметр, воспользуйтесь функцией DialogBoxParam:
int WINAPI DialogBoxParam(HINSTANCE hInstance, LPCSTR lpszTemplate, HWND hwndOwner, DLGPROC dlgprc, LPARAM lParamInit);
Эта функция полностью аналогична функции DialogBox, за исключением дополнительного параметра lParamInit. Значение этого параметра передается через параметр lParam сообщения WM_INITDIALOG и может быть проанализировано на этапе создания диалоговой панели.
Как мы уже говорили, существует редко используемая возможность создания диалоговой панели с помощью шаблона, сформированного непосредственно в памяти (а не загруженного из ресурсов приложения). Для создания таких диалоговых панелей предназначены функции DialogBoxIndirect и DialogBoxIndirectParam.
Функция DialogBoxIndirect аналогична функции DialogBox, но в качестве второго параметра в ней используется не указатель на строку имени шаблона, а идентификатор глобального блока памяти, в котором подготовлен шаблон:
int WINAPI DialogBoxIndirect(HINSTANCE hInstance, HGLOBAL hglbDlgTemplate, HWND hwndOwner, DLGPROC dlgprc);
Функция DialogBoxIndirectParam аналогична функции DialogBoxIndirect, но имеет дополнительный параметр lParamInit:
int WINAPI DialogBoxIndirectParam(HINSTANCE hInstance, HGLOBAL hglbDlgTemplate, HWND hwndOwner, DLGPROC dlgprc, LPARAM lParamInit);
Для создания немодальных диалоговых панелей используются функции CreateDialog, CreateDialogParam, CreateDialogIndirect, CreateDialogIndirectParam. Эти функции имеют параметры, аналогичные параметрам функций DialogBox, DialogBoxParam, DialogBoxParamIndirect:
HWND WINAPI CreateDialog(HINSTANCE hInstance, LPCSTR lpszTemplate, HWND hwndOwner, DLGPROC dlgprc);

HWND WINAPI CreateDialogParam(HINSTANCE hInstance, LPCSTR lpszTemplate, HWND hwndOwner, DLGPROC dlgprc, LPARAM lParamInit);

HWND WINAPI CreateDialogIndirect(HINSTANCE hInstance, HGLOBAL hglbDlgTemplate, HWND hwndOwner, DLGPROC dlgprc);

HWND WINAPI CreateDialogIndirectParam(HINSTANCE hInstance, HGLOBAL hglbDlgTemplate, HWND hwndOwner, DLGPROC dlgprc, LPARAM lParamInit);
Функции DialogBox, DialogBoxParam, DialogBoxIndirect, и DialogBoxIndirectParam возвращают значение, передаваемое при завершении работы диалоговой панели с помощью функции EndDialog.
Функция EndDialog имеет следующий прототип:
void WINAPI EndDialog(HWND hdlg, int nResult);
В качестве первого параметра функции необходимо указать идентификатор завершаемой диалоговой панели, который передается в функцию диалога через параметр hdlg.

Функции CreateDialog, CreateDialogParam, CreateDialogIndirect, и CreateDialogIndirectParam возвращают идентификатор окна для созданной диалоговой панели.

7.2. Сообщения для органов управления

Органы управления, расположенные на поверхности диалоговой панели, посылают в функцию диалога сообщение WM_COMMAND. В свою очередь, приложение может посылать различные сообщения органам управления, вызывая функцию SendMessage.

Использование функции SendMessage

Первый параметр функции SendMessage является идентификатором окна, функция которого должна получить сообщение. Если вы создаете модальную диалоговую панель, функция диалога получает идентификатор окна диалоговой панели. Вам же нужны идентификаторы окон отдельных органов управления.
Программный интерфейс Windows содержит специальную функцию, предназначенную для определения идентификаторов окна органов управления по идентификатору окна диалога и идентификатору самого органа управления. Эта функция называется GetDlgItem: HWND WINAPI GetDlgItem(HWND hdlg, int idControl);
В качестве параметра hdlg этой функции необходимо передать идентификатор окна диалоговой панели. Для модальных диалоговых панелей этот идентификатор можно определить только внутри функции диалога (он передается через первый параметр функции диалога). Идентификатор окна немодальной диалоговой панели возвращается функциями, создающими такую панель, например, функцией CreateDialog.
Второй параметр является идентификатором органа управления, указанным в шаблоне диалоговой панели.
Для того чтобы установить переключатель с идентификатором IDC_SWITCH во включенное состояние, вы можете вызывать функцию SendMessage следующим образом: SendMessage(GetDlgItem(hdlg, IDC_SWITCH), BM_SETCHECK, TRUE, 0L);
Зная идентификатор окна органа управления, вы можете получить идентификатор самого органа управления, т. е. решить задачу, обратную выполняемой функцией DetDlgItem. Для этого следует воспользоваться функцией GetWindowWord, передав ей в качестве второго параметра константу GWW_ID:

```
nIDControl = GetWindowWord(hwndControl, GWW_ID);
```

Эта функция возвращает значения из области дополнительной памяти, определенной при регистрации класса окна. Напомним, что размер дополнительной области памяти задается значением, записанным в элементе cbWndExtra структуры WNDCLASS.

Использование специальных функций

Для упрощения работы с органами управления, расположенными в диалоговых панелях, в программном интерфейсе Windows определены специальные функции. При использовании этих функций вам не требуется указывать идентификаторы окна органов управления, достаточно знать идентификаторы самих органов управления, определенные в описании шаблона.
Для посылки сообщения органу управления удобно использовать функцию SendDlgItemMessage:

LRESULT WINAPI SendDlgItemMessage(HWND hdlg, int idDlgItem, UINT uMsg, WPARAM wParam, LPARAM lParam);
В качестве параметра hdlg этой функции необходимо указать идентификатор окна диалоговой панели. Параметр idDlgItem определяет идентификатор органа управления, которому предназначается сообщение. Остальные три параметра этой функции содержат, соответственно, код сообщения и параметры, передаваемые вместе с сообщением.
Для выполнения некоторых часто использующихся операций с органами управления в программном интерфейсе Windows определены специальные функции.
В частности, для заполнения списка LISTBOX именами файлов, каталогов и дисковых устройств предназначена функция DlgDirList:
int WINAPI DlgDirList(HWND hdlg, LPSTR lpszPath, int idListBox, int idStatic, UINT uFileType);
Первый параметр этой функции указывает идентификатор окна диалоговой панели. Параметр lpszPath - указатель на строку, содержащую шаблон для имен файлов. Параметр idListBox перед вызовом функции должен содержать идентификатор заполняемого списка.
В качестве параметра idStatic вы должны указать идентификатор статического органа управления, в который будет записана строка полного пути к текущему каталогу, или NULL, если статический орган управления не используется.
И, наконец, последний параметр этой функции определяет тип файлов, имена которых заносятся в список, а также указывают на необходимость записи в список имен каталогов и дисковых устройств. Этот параметр должен быть указан как логическая комбинация констант с префиксом имени DDL_ (вы уже знакомы с этими константами).
Аналогичная функция предусмотрена и для списка COMBOBOX:
int WINAPI lgDirListComboBox (HWND hdlg, LPSTR lpszPath, int idListBox, int idStatic, UINT uFileType);
Назначение параметров этой функции полностью аналогично назначению параметров функции DlgDirList.
Функция DlgDirSelect предназначена для получения из списка LISTBOX (подготовленного с помощью функции DlgDirList) строки, выбранной пользователем:
BOOL WINAPI DlgDirSelect(HWND hdlg, LPSTR lpszBuffer, int idListBox);
Параметр hdlg определяет диалоговую панель. Нужный список задается параметром idListBox. Выбранная строка будет записана в буфер, адрес которой указан с помощью параметра lpszBuffer. Размер буфера должен быть не меньше 128 байт.
Аналогичная функция предусмотрена для списка COMBOBOX:
BOOL WINAPI DlgDirSelectComboBox (HWND hdlg, LPSTR lpszBuffer, int idListBox);
Если ваше приложение будет работать в среде Windows версии 3.1 или более старшей версии, для получения выбранной пользователем строки вы можете использовать функции DlgDirSelectEx и DlgDirSelectComboBoxEx:
BOOL WINAPI lgDirSelectEx(HWND hdlg, LPSTR lpszBuffer, int cbBufSize, int idListBox);
BOOL WINAPI DlgDirSelectComboBoxEx(HWND hdlg, LPSTR lpszBuffer, int cbBufSize, int idListBox);

Эти функции позволяют получить в буфер lpszBuffer размером cbBufSize байт строку, выбранную пользователем из списка с идентификатором idListBox, расположенном в диалоговой панели hdlg. Однако для выбранной строки выполняется дополнительная обработка, а именно: если выбрано имя каталога или дискового устройства, функция удаляет из строки квадратные скобки и символы "-".
В программном интерфейсе Windows имеется несколько функций, облегчающих работу с редакторами текстов, расположенных в диалоговых панелях.
Функция SetDlgItemText позволяет изменить заголовок органа управления или записать текст в текстовый редактор:
void WINAPI SetDlgItemText(HWND hdlg, int idControl, LPCSTR lpszText);
Текстовая строка lpszText записывается в орган управления с идентификатором idControl, расположенным в диалоговой панели hdlg.
Функция SetDlgItemInt позволяет записать в заголовок органа управления или текстовый редактор текстовую строку, полученную после преобразования целого числа в формат строки символов:
void WINAPI SetDlgItemInt(HWND hdlg, int idControl, UINT uValue, BOOL fSigned);
Для диалоговой панели с идентификатором окна, равным hdlg, эта функция записывает символьное представление параметра uValue в заголовок органа управления или редактор текста с идентификатором idControl. Если параметр fSigned указан как TRUE, значение uValue интерпретируется как знаковое целое, если FALSE - как беззнаковое целое.
Для получения строки, связанной с органом управления, расположенном в диалоговой панели, можно использовать функцию GetDlgItemText:
int WINAPI GetDlgItemText(HWND hdlg, int idControl, LPSTR lpszBuffer, int cbBufferSize);
Эта функция записывает текст, связанный с органом управления idControl, в буфер lpszBuffer, имеющий размер cbBufferSize байт.
Предусмотрена также функция, получающая из органа управления текстовую строку и выполняющая преобразование этой строки в целое число:
UINT WINAPI GetDlgItemInt (HWND hdlg, int idControl, BOOL FAR* lptTranslated, BOOL fSigned);
Эта функция возвращает целое число, которое образуется после преобразования текста, связанного с органом управления idControl в диалоговой панели hdlg. Если параметр fSigned указан как TRUE, преобразуемая строка интерпретируется как символьное представление знакового целого, если FALSE - как беззнакового целого. В переменную, адрес которой передается через параметр lptTranslated, записывается код ошибки. Если преобразование выполнено без ошибок, в переменную записывается значение TRUE, в противном случае - FALSE.
Есть также функции, предназначенные для работы с переключателями.
Функция CheckDlgButton предназначена для изменения состояния переключателя CHECKBOX (включения или выключения):
void WINAPI CheckDlgButton(HWND hdlg, int idButton, UINT uState);
Для переключателя с идентификатором idButton, расположенного в диалоговой панели hdlg, устанавливается новое состояние в соответствии со значением параметра uState. Для выключения переключателя параметр uState должен иметь нулевое зна-

чение. Если этот параметр будет равен 1, переключатель будет включен, а если 2 - переведен в неактивное состояние.
Аналогичная функция предусмотрена для переключателей RADIOBUTTON:
void WINAPI CheckRadioButton (HWND hdlg, int idFirstButton, int idLastButton, int idCheckButton);
Эта функция может обслуживать сразу группу переключателей, имеющих идентификаторы от idFirstButton до idLastButton. Она включает переключатель с идентификатором idCheckButton, после чего выключает все остальные переключатели группы в указанном параметрами idFirstButton и idLastButton диапазоне идентификаторов.
Для определения текущего состояния переключателя вы можете воспользоваться функцией IsDlgButtonChecked: UINT WINAPI IsDlgButtonChecked(HWND hdlg, int idButton);
Эта функция возвращает состояние переключателя с идентификатором idButton, расположенного в диалоговой панели hdlg. Если переключатель находится в выключенном состоянии, возвращается нулевое значение. Для включенного переключателя возвращается значение 1. Значение 2 соответствует неактивному переключателю, изображенному серым цветом. В случае ошибки возвращается отрицательное значение -1.
Если ваше приложение использует собственную логику для передачи фокуса ввода между органами управления, расположенными в диалоговой панели, ему может потребоваться информация о последовательности, в которой должен передаваться фокус ввода. С помощью функции GetNextDlgGroupItemприложение может определить идентификатор окна предыдущего или следующего органа управления в группе:
HWND WINAPI GetNextDlgGroupItem(HWND hdlg, HWND hwndControl, BOOL fPrevious);
В зависимости от значения флага fPrevious функция возвращает идентификатор предыдущего или следующего органа управления группе относительно органа управления с идентификатором hwndControl. Если значение флага fPrevious равно TRUE, функция возвращает идентификатор окна для предыдущего органа управления в группе, если FALSE - для следующего.
Функция GetNextDlgTabItem позволяет определить идентификатор окна для первого органа управления, который имеет стиль WS_TABSTOP и расположен после органа управления с заданным идентификатором или перед этим органом:
HWND WINAPI GetNextDlgTabItem(HWND hdlg, HWND hwndControl, BOOL fPrevious);
Параметр hwndControl определяет орган управления, начиная с которого функция будет выполнять поиск, параметр fPrevious определяет направление поиска. Если значение параметра fPrevious равно TRUE, функция ищет предыдущий орган управления в группе, если FALSE - следующий.
Следует упомянуть еще две функции, имеющие отношение к специфической системе координат, принятой для работы с диалоговыми панелями.
Функция MapDialogRect преобразует координаты из единиц диалоговой панели (dialog units) в пиксели: void WINAPI MapDialogRect(HWND hdlg, RECT FAR* lprc);

Преобразуемые координаты необходимо записать в структуру типа RECT, адрес которой указывается во втором параметре функции. Результат преобразования будет записан в эту же структуру.
Функция GetDialogBaseUnits возвращает двойное слово, содержащее информацию о диалоговой системе координат: DWORD WINAPI GetDialogBaseUnits(void);
Младшее слово представляет собой ширину в пикселях диалоговой единицы длины, старшее - высоту.

7.3. Немодальные диалоговые панели

В отличие от модальных диалоговых панелей, блокирующих при своем появлении родительское окно и все дочерние окна родительского окна, немодальные диалоговые панели работают параллельно с другими окнами приложения без взаимных блокировок. Вы можете работать как с главным окном приложения, так и окном немодальной диалоговой панели.
Немодальные диалоговые панели очень удобны для объединения различных инструментальных средств, предназначенных для работы с объектом, расположенным в главном окне или в дочернем окне, созданным главным окном приложения.

Создание и уничтожение немодальных диалоговых панелей

Для создания немодальных диалоговых панелей используются описанные нами ранее функции CreateDialog, CreateDialogParam, CreateDialogIndirect, reateDialogIndirectParam.
Все эти функции возвращают идентификатор окна созданной диалоговой панели, который можно использовать для посылки сообщений органам управления, расположенным в этой панели.
Для обеспечения автоматического отображения немодальной диалоговой панели сразу после создания стиль панели, описанный в шаблоне, должен включать в себя константу WS_VISIBLE. В противном случае для отображения диалоговой панели придется вызывать функцию ShowWindow. Стиль DS_MODALFRAME используется только для модальных диалоговых панелей, поэтому его указывать не надо.
Завершение работы немодальной диалоговой панели следует выполнять с помощью функции DestroyWindow, указав ей в качестве параметра идентификатор окна панели, полученный от функции CreateDialog или от аналогичной функции, создающей немодальную диалоговую панель. Функция EndDialog должна использоваться только для завершения работы модальных диалоговых панелей.

Изменения в цикле обработки сообщений

При создании модальной диалоговой панели в приложении создается второй цикл обработки сообщений, который действует до завершения работы модальной диалоговой панели. Если же создается немодальная диалоговая панель, сообщения от органов управления, предназначенные для диалоговой панели, проходят через общую очередь сообщений приложения.
Для разделения этих сообщений цикл обработки должен вызывать функцию IsDialogMessage: BOOL WINAPI IsDialogMessage(HWND hwndDlg, MSG FAR* lpmsg);
Функция IsDialogMessage определяет, предназначено ли сообщение, определяемое параметром lpmsg, для немодальной диалоговой панели с идентификатором окна,

равным hwndDlg. Если предназначено, функция сама выполняет обработку такого сообщения и возвращает значение TRUE. В противном случае возвращается значение FALSE.

7.4. Функция MessageBox

int WINAPI MessageBox(HWND hwndParent, LPCSTR lpszText, LPCSTR lpszTitle, UINT fuStyle);

Эта функция создает на экране диалоговую панель с текстом, заданным параметром lpszText и заголовком, заданным параметром lpszTitle. Если заголовок указан как NULL, используется заголовок по умолчанию - строка "Error".

Параметр hwndParent указывает идентификатор родительского окна, создающего диалоговую панель. Этот параметр можно указывать как NULL, в этом случае у диалоговой панели не будет родительского окна. Вы можете вызвать функцию MessageBox из функции диалога, в этом случае первый параметр должен содержать идентификатор окна диалоговой панели.

Последний параметр fuStyle определяет стиль и внешний вид диалоговой панели. Вы можете использовать одну из следующих констант, определяющих количество кнопок, расположенных на диалоговой панели и надписи на этих кнопках.

Константа	Описание
MB_ABOTRETRYIGNORE	Диалоговая панель содержит три кнопки с надписями "Abort", "Retry", "Ignore"
MB_OK	Диалоговая панель содержит одну кнопку "OK"
MB_OKCANCEL	Две кнопки с надписями "OK", "Cancel"
MB_RETRYCANCEL	Две кнопки с надписями "Retry", "Cancel"
MB_YESNO	Две кнопки с надписями "Yes", "No"
MB_YESNOCANCEL	Три кнопки с надписями "Yes", "No", "Cancel"

К этим константам при помощи логической операции ИЛИ можно добавлять другие константы.

По умолчанию после инициализации диалоговой панели фокус ввода имеет первая кнопка. Эта кнопка будет использована по умолчанию. Вы можете определить в качестве кнопки, используемой по умолчанию любую из трех кнопок с помощью следующих констант:

Константа	Описание
MB_DEFBUTTON1	Первая кнопка используется по умолчанию
MB_DEFBUTTON2	Вторая кнопка используется по умолчанию
MB_DEFBUTTON3	Третья кнопка используется по умолчанию

С помощью следующих трех констант вы можете влиять на модальность диалоговой панели:

Константа	Описание
MB_APPLMODAL	Создается модальная диалоговая панель. Окно, указанное параметром hwndParent, переводится в неактивное состояние до тех пор, пока пользователь не завершит работу с диалоговой панелью. Пользователь может переключиться на другое приложение. Этот стиль используется по умол-

	чанию
MB_SYSTEMMODAL	До тех пор, пока пользователь не завершит работу с диалоговой панелью, все остальные приложения переводятся в неактивное состояние
MB_TASKMODAL	Аналогично MB_APPLMODAL за исключением того, что если параметр hwndParent имеет значение NULL, блокируются все окна верхнего уровня, принадлежащие данной задаче. Этот стиль используется тогда, когда идентификатор родительского окна неизвестен, но тем не менее требуется перевести все окна текущего приложения в неактивное состояние до тех пор, пока пользователь не завершит работу с диалоговой панелью

С помощью следующих шести констант вы можете создать в диалоговой панели пиктограмму: MB_ICONASTERISK, MB_ICONEXCLAMATION ,MB_ICONHAND, MB_ICONINFORMATION, MB_ICONQUESTION, MB_ICONSTOP.

8..Иллюстративные и другие ресурсы.

8.1. Включение ресурсов

Для включения ресурсов в загрузочный модуль приложения вам надо создать текстовый файл описания ресурсов. Этот файл может быть создан либо текстовым редактором, либо при помощи редактора ресурсов в среде разработки.
Файл описания ресурсов имеет расширение имени .rc. Его необходимо включить в проект приложения наряду с файлами исходных текстов и файлом определения модуля.
В процессе сборки загрузочного модуля файл описания ресурсов компилируется специальным компилятором ресурсов rc.exe. Компилятор ресурсов поставляется вместе с системой разработки приложений Windows. Он преобразует входной текстовый файл описания ресурсов в двоичный файл с расширением имени .res (вы можете указывать в проекте либо текстовый, либо двоичный вариант файла описания ресурсов, однако лучше использовать текстовый вариант, так как его можно редактировать). Перед запуском компилятора ресурсов система разработки приложений Windows запускает препроцессор текстового описания ресурсов rcpp.exe, который обрабатывает разделители комментариев и директивы препроцессора Си.
На финальном этапе сборки загрузочного модуля компилятор ресурсов rc.exe вызывается еще раз для записи ресурсов в загрузочный модуль. Дополнительно компилятор ресурсов формирует специальную таблицу ресурсов, расположенную в заголовке exe-файла. Таблица ресурсов используется Windows для поиска и загрузки ресурсов в оперативную память.
Исходные тексты приложения Windows, составленные на языках программирования C, C++ или на языке ассемблера, компилируются в объектные модули *.obj. С помощью утилиты tlib.exe объектные модули могут быть собраны в библиотеки *.lib. Далее редактор связей, входящий в систему разработки приложений Windows, соби-

рает из объектных модулей промежуточный вариант загрузочного модуля, не содержащий ресурсов. При этом используется файл определения модуля *.def. Файл описания ресурсов *.rc компилируется утилитой rc.exe в двоичный файл *.res. На последней стадии формирования загрузочного модуля промежуточный вариант exe-файла комбинируется с файлом ресурсов для получения окончательного варианта загрузочного модуля.

8.2. Таблица текстовых строк

Самый простой в использовании ресурс - таблица строк. Таблица строк содержит текстовые строки, закрытые двоичным нулем, на которые можно ссылаться по идентификатору. Идентификатор представляет собой обыкновенное целое число. Вместо чисел обычно используют символические имена, определенные с помощью директивы #define.

Создание таблицы

Для создания таблицы строк текстовый файл описания ресурсов должен содержать оператор STRINGTABLE:

```
STRINGTABLE [параметры загрузки][тип памяти]BEGIN
StringID, строка
...
...
...
END
```

В качестве параметров загрузки можно указывать значения PRELOAD или LOADONCALL (используется по умолчанию). Ресурс с параметром загрузки LOADONCALL загружается в память при обращении к нему со стороны приложения. Ресурс типа PRELOAD загружается сразу после запуска приложения.
Тип памяти, выделяемой при загрузки ресурса, может быть FIXED или MOVABLE. Дополнительно для ресурсов типа можно указать MOVABLE тип DISCARDABLE. Если указан тип FIXED, ресурс будет находиться в памяти по постоянному адресу. Ресурс типа MOVABLE может перемещаться Windows при необходимости уплотнения памяти. Если для перемещаемого ресурса указан тип DISCARDABLE, Windows может забрать у приложения память, выделенную для ресурса. Если ресурс потребуется приложению, Windows загрузит его повторно из exe-файла приложения.
Операторы BEGIN и END определяют границы таблицы строк в файле описания ресурсов. Между ними находятся строки с идентификаторами StringID:

```
STRINGTABLE
BEGIN
1, "Файл %s не найден"
2, "Ошибка при записи в файл %s"
3, "Ошибка ввода/вывода"
END
```

Загрузка строки из таблицы

Для загрузки строки в оперативную память необходимо использовать функцию LoadString:

```
int WINAPI LoadString(
  HINSTANCE hInst,      // идентификатор приложения
  UINT idResource,      // идентификатор ресурса
  LPSTR lpszBuffer,     // адрес буфера
  int cbBuffer);        // размер буфера в байтах
```

Параметр hInst определяет идентификатор запущенного приложения, из загрузочного файла которого необходимо извлечь текстовую строку.
Параметр idResource указывает идентификатор нужной строки. Вы должны использовать одно из значений, заданных в файле описания ресурсов.
Указатель lpszBuffer определяет адрес буфера, в который будет загружена строка. Размер буфера должен быть задан через параметр cbBuffer. Если размер строки окажется больше указанного размера буфера, строка будет обрезана.
Функция LoadString возвращает количество символов, записанных в буфер, или 0, если файл загрузочного модуля не содержит строки с указанным идентификатором (или если в этом файле вообще нет таблицы строк).

Функции для работы с текстовыми строками

Для работы с текстовыми строками приложения Windows могут вызывать стандартные функции библиотеки компилятора, такие как strcat или strcopy. Однако лучше использовать функции для работы с текстовыми строками, определенные в программном интерфейсе Windows. Эти функции используют дальние указатели на строки и учитывают специфику национальных алфавитов. Кроме того, указанные функции находятся в ядре Windows, поэтому при их использовании не происходит увеличения размера файла загрузочного модуля приложения.
Функция lstrcmp сравнивает строки, заданные параметрами:

```
int WINAPI lstrcmp(LPCSTR lpszString1, LPCSTR lpszString2);
```

Функция возвращает отрицательное значение, если строка lpszString1 меньше чем строка lpszString2, положительное в противоположном случае, и равное нулю при равенстве сравниваемых строк. При сравнении учитываются особенности национального алфавита для указанной при помощи приложения Control Panel страны. Функция способна сравнивать строки с двухбайтовыми кодами символов. Учитываются также заглавные и прописные буквы. Размер сравниваемых строк не может превышать 64 Кбайт.
Функция lstrcmpi предназначена для сравнения двух строк, но без учета заглавных и прописных букв:

```
int WINAPI lstrcmpi(LPCSTR lpszString1, LPCSTR lpszString2);
```

В остальном она полностью аналогична функции lstrcmp.
Учтите, что известные вам функции strcmp и strcmpi не учитывают особенности национальных алфавитов и поэтому их не следует использовать в приложениях Windows.
Для копирования текстовых строк вы должны пользоваться функцией lstrcpy:

```
LPSTR WINAPI lstrcpy(LPSTR lpszString1, LPCSTR lpszString2);
```

Эта функция копирует строку lpszString2 в строку lpszString1, возвращая указатель на первую строку или NULL при ошибке. В отличие от своего аналога из библиотеки функций MS-DOS (функции strcpy) эта функция способна работать со строками,

содержащими двухбайтовые коды символов. Размер копируемой строки не должен превышать 64 Кбайт.
В Windows есть еще одна функция, предназначенная для копирования заданного количества символов из одной строки в другую. Эта функция имеет имя lstrcpyn:
LPSTR WINAPI lstrcpyn(LPSTR lpszString1, LPCSTR lpszString2, int cChars);
Она копирует cChars символов из строки lpszString2 в строку lpszString1.
Для объединения двух строк в приложениях Windows следует применять функцию lstrcat: LPSTR WINAPI lstrcat(LPSTR lpszString1, LPCSTR lpszString2);
Функция lstrcat добавляет строку lpszString2 к строке lpszString1. Размер строки, получившейся в результате объединения, не должен превышать 64 Кбайт. Функция возвращает указатель на строку lpszString1.
Длину текстовой строки (без учета закрывающего строку двоичного нуля) можно получить при помощи функции lstrlen, аналогичной известной вам функции strlen:
int WINAPI lstrlen(LPCSTR lpszString);
Для классификации символов на строчные, прописные, буквенные или цифровые приложения должны использовать специально предназначенные для этого функции из программного интерфейса Windows.
Функция IsCharAlpha возвращает значение TRUE, если символ, заданный параметром chTest, является буквой: BOOL WINAPI IsCharAlpha(char chTest);
Функция IsCharAlphaNumeric возвращает значение TRUE, если символ, заданный параметром chTest, является буквой или цифрой:
BOOL WINAPI IsCharAlphaNumeric(char chTest);
Функция IsCharUpper возвращает значение TRUE, если символ, заданный параметром chTest, является прописным (заглавным): BOOL WINAPI IsCharUpper(char chTest);
Функция IsCharLower возвращает значение TRUE, если символ, заданный параметром chTest, является строчным: BOOL WINAPI IsCharLower(char chTest);
Windows и MS-DOS используют разные наборы символов. Приложения Windows обычно работают с наборами в стандарте ANSI, программы MS-DOS - в стандарте OEM. Для перекодировки строки символов, закрытой двоичным нулем, из набора ANSI в набор OEM предназначена функция AnsiToOem:
void WINAPI AnsiToOem(const char _huge* hpszWindowsStr, char _huge* hpszOemStr);
Параметр hpszWindowsStr представляет собой указатель типа _huge на преобразуемую строку, параметр hpszOemStr - указатель на буфер для записи результата преобразования.
Похожая по назначению функция AnsiToOemBuff выполняет преобразование массива заданного размера:
void WINAPI AnsiToOemBuff(LPCSTR lpszWindowsStr, LPSTR lpszOemStr, UINT cbWindowsStr);
Первый параметр этой функции (lpszWindowsStr) является дальним указателем на массив, содержащий преобразуемые данные, второй (lpszOemStr) - на буфер для записи результата. Третий параметр (cbWindowsStr) определяет размер входного массива, причем нулевой размер соответствует 64 Кбайт (65536 байт).
Обратное преобразование выполняется функциями OemToAnsi и OemToAnsiBuff:

void WINAPI OemToAnsi(const char _huge* hpszOemStr, char _huge* lpszWindowsStr);void WINAPI OemToAnsiBuff(LPCSTR lpszOemStr, LPSTR lpszWindowsStr, UINT cbOemStr);
Назначение параметров этих функций аналогично назначению параметров функций AnsiToOem и AnsiToOemBuff.
Для преобразований символов в строчные или прописные приложение Windows должно пользоваться функциями AnsiLower, AnsiLowerBuff, AnsiUpper, AnsiUpperBuff.
Функция AnsiLower преобразует закрытую двоичным нулем текстовую строку в строчные (маленькие) буквы:
LPSTR WINAPI AnsiLower(LPSTR lpszString);
Единственный параметр функции - дальний указатель на преобразуемую строку.
Функция AnsiUpper преобразует закрытую двоичным нулем текстовую строку в прописные (большие) буквы: LPSTR WINAPI AnsiLower(LPSTR lpszString);
Параметр функции lpszString - дальний указатель на преобразуемую строку.
Функция AnsiLowerBuff позволяет преобразовать в строчные (маленькие) буквы заданное количество символов:
UINT WINAPI AnsiLowerBuff(LPSTR lpszString, UINT cbString);
Первый параметр функции (lpszString) является указателем на буфер, содержащий преобразуемые символы, второй (cbString) определяет количество преобразуемых символов (размер буфера). Нулевой размер соответствует буферу длиной 64 Кбайт (65536 байт).
Функция возвращает количество преобразованных символов.
Функция AnsiUpperBuff позволяет преобразовать в прописные (большие) буквы заданное количество символов:
UINT WINAPI AnsiUpperBuff(LPSTR lpszString, UINT cbString);
Первый параметр функции lpszString(lpszString) является указателем на буфер, содержащий преобразуемые символы, второй (cbString) определяет количество преобразуемых символов (размер буфера). Нулевой размер соответствует буферу длиной 64 Кбайт (65536 байт).
Эта функция, как и предыдущая, возвращает количество преобразованных символов.
Функция AnsiNext возвращает новое значение для указателя, передвинутое вперед по строке на одни символ:
LPSTR WINAPI AnsiNext(LPCSTR lpchCurrentChar);
Параметр функции указывает на текущий символ. Возвращаемое значение является указателем на следующий символ в строке или на закрывающий строку двоичный ноль.
Функция AnsiPrev выполняет передвижение указателя в направлении к началу строки:
LPSTR WINAPI AnsiPrev(LPCSTR lpchStart, LPCSTR lpchCurrentChar);
Первый параметр функции указывает на начало строки (на первый символ строки). Второй параметр - указатель на текущий символ. Функция возвращает значение указателя, соответствующее предыдущему символу или первому символу в строке, если при продвижении достигнуто начало строки.

8.3. Пиктограмма

В загрузочный модуль приложения Windows можно добавить ресурс, который называется пиктограмма (Icon). Пиктограмма - это графическое изображение небольшого размера, состоящее из отдельных пикселов. Пиктограммы обычно используются для обозначения свернутого окна приложения. Окна групп приложения Program Manager содержат пиктограммы других приложений.
С помощью редактора ресурсов вы можете нарисовать свои собственные пиктограммы и использовать их для представления главного окна приложения в свернутом состоянии, для оформления внешнего вида окон приложения или для решения других задач.
Операционная система Windows содержит ряд встроенных пиктограмм, которые доступны приложениям.
Приложения Windows активно работают с графическими изображениями, состоящими из отдельных пикселов и имеющих прямоугольную или квадратную форму. Такие изображения называются bitmap (битовый образ). Для представления цвета отдельного пиксела в изображении может быть использовано различное количество бит памяти. Например, цвет пиксела черно-белого изображения может быть представлен одним битом, для представления цвета 16-цветного изображения нужно 4 бита. Цвет одного пиксела изображения в режиме TrueColor представляется 24 битами памяти.
Можно считать, что пиктограмма является упрощенным вариантом изображения bitmap. Пиктограммы хранятся в файлах с расширением имени *.ico (хотя можно использовать любое расширение). В одном файле обычно находится несколько вариантов пиктограмм. Когда Windows рисует пиктограмму, он выбирает из файла наиболее подходящий для текущего режима работы видеоадаптера.

Создание пиктограммы

Редактор ресурсов позволяет вам создать пиктограммы размером 32x16 пикселов , 32x32 пиксела и 64x64 пиксела (для режимов SVGA с большим разрешением). Вы можете создать монохромную или цветную пиктограмму. Цветная пиктограмма содержит 8, 16 или 256 цветов
В одном файле *.ico допускается хранить несколько пиктограмм различного размера и с различным количеством цветов. В этом случае при выводе пиктограммы Windows сделает правильный выбор для текущего режима работы видеоадаптера.

Включение пиктограммы в файл описания ресурсов

Для включения пиктограммы в файл описания ресурсов используется оператор ICON (для включения пиктограмм в диалоговые панели используется другой формат, который мы сейчас не будем рассматривать):
IconID ICON [параметры загрузки] [тип памяти] имя файла
Параметры загрузки и тип памяти указывается так же, как и для описанной нами ранее таблицы строк STRINGTABLE. В качестве имени файла необходимо указать имя файла, содержащего пиктограмму, например: AppIcon ICON myicon.ico
Идентификатор пиктограммы IconID можно указывать как символическое имя (см. предыдущую строку) или как целое число - идентификатор ресурса:

456 ICON great.ico
После сборки проекта файл пиктограммы будет вставлен в исполняемый файл приложения Windows.

Использование пиктограммы при регистрации класса окна

Во всех приложениях, описанных в предыдущем томе "Библиотеки системного программиста", для класса главного окна приложения мы определяли встроенную в Windows пиктограмму с идентификатором IDI_APPLICATION. Для этого мы вызывали функцию LoadIcon: wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
Эта функция имеет следующий прототип:
HICON WINAPI LoadIcon(HINSTANCE hInst, LPCSTR lpszIcon);
Параметр hInst является идентификатором текущей копии приложения.
Параметр lpszIcon - идентификатор ресурса или дальний указатель на строку, идентифицирующую ресурс.
Функция LoadIcon возвращает идентификатор загруженной пиктограммы или NULL при ошибке. Если в файле описания ресурсов идентификатор пиктограммы представлен в виде текстовой строки, в качестве второго параметра функции LoadIcon следует передать указатель на эту строку.
Следующая строка задает для класса главного окна приложения пиктограмму, определенную в файле описания ресурсов под именем AppIcon:
wc.hIcon = LoadIcon(hInstance, "AppIcon");
Если же для идентификатора пиктограммы используется целое число, второй параметр следует определить с использованием макрокоманды MAKEINTRESOURCE, определенной в файле windows.h:
#define MAKELP(sel, off) ((void FAR*)MAKELONG((off), (sel)))
#define MAKEINTRESOURCE(i) ((LPCSTR)MAKELP(0, (i)))
Например, пусть в файле описания ресурсов определена пиктограмма с целым числом в качестве идентификатора: 456 ICON great.ico
В этом случае загрузка пиктограммы должна выполняться следующим образом:
wc.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(456));
Возможен еще один вариант: wc.hIcon = LoadIcon(hInstance, "#456");
Для Windows символ "#" означает, что значение 456 является текстовой строкой.
Если пиктограмма, загруженная с помощью функции LoadIcon, вам больше не нужна, вы можете освободить занимаемую ей память, вызвав функцию DestroyIcon:
BOOL WINAPI DestroyIcon(HICON hIcon);
Функция DestroyIcon уничтожает пиктограмму, идентификатор которой задан параметром hIcon и освобождает ранее занимаемую этой пиктограммой память.

Изображение пиктограммы в окне приложения

После того как вы загрузили пиктограмму функцией LoadIcon, вы можете нарисовать ее в любом месте окна. Для этого вам достаточно вызвать функцию DrawIcon:
BOOL WINAPI DrawIcon(HDC hDC, int x, int y, HICON hIcon);
Параметр hDC - идентификатор контекста отображения, полученный для рисования.
Пиктограмма с идентификатором hIcon будет нарисована в точке с координатами (x,y), определяемыми параметрами x и y. Функция DrawIcon возвращает значение TRUE при успешном завершении и FALSE при ошибке.

8.4. Курсор мыши

Курсор мыши представляет собой ни что иное, как упрощенный вариант битового изображения (bitmap), аналогичного пиктограмме. Вы можете выбрать один из встроенных курсоров, либо создать свой собственный. Для создания своего курсора его надо нарисовать, пользуясь редактором ресурсов, и записать в файл с расширением имени cur. Далее на файл с курсором следует сделать ссылку в файле описания ресурсов. После этого приложение может загрузить курсор в память и использовать его либо при регистрации класса окна, либо для изменения формы курсора в произвольный момент времени.

Включение курсора в файл описания ресурсов

Для включения курсора в файл описания ресурсов используется оператор CURSOR, аналогичный оператору ICON:
CursorID CURSOR [параметры загрузки] [тип памяти] имя файла
В качестве параметров загрузки можно указывать значения PRELOAD или LOADONCALL (используется по умолчанию). Ресурс с параметром загрузки LOADONCALL загружается в память при обращении к нему со стороны приложения. Ресурс типа PRELOAD загружается сразу после запуска приложения.
Тип памяти, выделяемой при загрузки ресурса, может быть FIXED или MOVEABLE. Дополнительно для ресурсов типа можно указать MOVEABLE тип DISCARDABLE. Если указан тип FIXED, ресурс будет находиться в памяти по постоянному адресу. Ресурс типа MOVEABLE может перемещаться Windows при необходимости уплотнения памяти. Если для перемещаемого ресурса указан тип DISCARDABLE, Windows может забрать у приложения память, выделенную для ресурса. Если ресурс потребуется приложению, Windows загрузит его повторно из exe-файла приложения.
Идентификатор курсора CursorID можно указывать как символическое имя или как целое число - идентификатор ресурса: 789 CURSOR custom.cur
После сборки проекта файл курсора будет вставлен в исполняемый файл приложения Windows.
Для загрузки курсора следует использовать функцию LoadCursor:
HCURSOR WINAPI LoadCursor(HINSTANCE hInst,   LPCSTR lpszCursor);
Для загрузки нового курсора из ресурсов приложения в качестве параметра hInst необходимо указать идентификатор приложения, полученный через параметр hInstance функции WinMain. Параметр lpszCursor должен при этом указывать на идентификатор ресурса.
Если в файле описания ресурсов идентификатор ресурса представлен символьной строкой, адрес этой строки необходимо указать в параметре lpszCursor:
HCURSOR hCustomCursor;hCustomCursor = LoadCursor(hInstance, "AppCursor");
Если же в качестве идентификатора ресурса-курсора использовано целое число, следует использовать макрокоманду MAKEINTRESOURCE:
HCURSOR hCustomCursor;hCustomCursor = LoadCursor(hInstance, MAKEINTRESOURCE(789));

Встроенные курсоры

Приложение Windows может использовать несколько встроенных курсоров. Приведем список идентификаторов встроенных курсоров.

Идентификатор	Применение
IDC_ARROW	Стандартный курсор в виде стрелки
IDC_IBEAM	Текстовый курсор
IDC_WAIT	Курсор в виде песочных часов. Используется при выполнении длительных операций
IDC_CROSS	Курсор в виде перекрестия
IDC_UPARROW	Курсор в виде вертикальной стрелки
IDC_SIZE	Индикация изменения размера
IDC_ICON	Пустая пиктограмма
IDC_SIZENWSE	Индикация изменения размера
IDC_SIZENESW	Индикация изменения размера
IDC_SIZEWE	Индикация изменения размера
IDC_SIZENS	Индикация изменения размера

Изменение формы курсора

При регистрации класса окна мы задавали форму курсора следующим способом:
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
В качестве второго параметра функции LoadCursor вы можете указать идентификатор встроенного курсора или идентификатор курсора из файла описания ресурсов. В последнем случае через первый параметр необходимо передать идентификатор текущей копии приложения: wc.hCursor = LoadCursor(hInstance, "AppCursor");
Для динамического изменения формы курсора (например, во время обработки сообщения) следует использовать функцию SetCursor:
HCURSOR WINAPI SetCursor(HCURSOR hcur);
Параметр hcur функции SetCursor должен указывать идентификатор нового курсора, подготовленный при помощи функции LoadCursor. Если указать параметр как NULL, изображение курсора исчезнет с экрана.
Для того чтобы выключить изображение курсора мыши или вновь включить его используют функцию ShowCursor: int WINAPI ShowCursor(BOOL fShow);
Функция управляет содержимым счетчика, который используется для определения момента включения или выключения изображения курсора мыши. Первоначально содержимое счетчика равно нулю. Этот счетчик увеличивается, когда необходимо включить курсор, и уменьшается при выключении курсора. Если счетчик больше или равен нулю, курсор мыши находится во включенном (видимом) состоянии.
Для включения курсора в качестве параметра fShow функции следует передать значение TRUE, для выключения - FALSE.
Возвращаемое функцией ShowCursor значение равно новому содержимому счетчика.

Изображение курсора в окне приложения

Строго говоря, специальной функции для рисования курсора мыши нет. Это понятно - курсор мыши рисует сама операционная система Windows, отслеживая перемещения мыши по столу (некоторые видеоконтроллеры рисуют курсор мыши с помощью аппаратных средств, но опять же под руководством Windows). Однако приложение все-таки может нарисовать изображение курсора в своем окне, воспользовавшись функцией DrawIcon. Для этого в качестве второго параметра функции DrawIcon следует передать идентификатор курсора, полученный при помощи функции LoadCurosor.
Заметим, что эта особенность функции DrawIcon не нашла отражения в документации, поэтому в следующих версиях Windows она может исчезнуть. Тем не менее в Windows благодаря практически одинаковому формату данных для пиктограммы и курсора, вы можете рисовать курсор как пиктограмму.

8.5. Графическое изображение типа bitmap

В ресурсы приложения вы можете включить произвольное графическое изображение в виде битового образа (в дальнейшем мы будем называть такое изображение изображением типа bitmap или просто изображением bitmap).
С помощью графического редактора, входящего в состав редактора ресурсов, или с помощью стандартного приложения Paint Brush вы можете нарисовать прямоугольное графическое изображение типа bitmap. При этом для представления цвета одного пикселя может использоваться разное количество бит. Изображение записывается в файл с расширением имени bmp.
Мы уже сталкивались с изображениями bitmap, когда выводили в окно пиктограмму и курсор. Эти объекты являются частными случаями изображения bitmap.
Изображения bitmap удобно использовать для оформления внешнего вида окон приложения. С помощью таких изображений вы можете, например, создать органы управления любой формы (например, круглые кнопки с картинками), нарисовать фотографию, введенную сканером, заставку или эмблему фирмы. Изображения bitmap открывают широкие возможности для разработки дизайна приложения Windows.

Включение изображения bitmap в файл описания ресурсов

Для включения изображения типа bitmap в файл описания ресурсов используется такой же способ, как и для включения пиктограмм. Файл описания ресурсов должен содержать оператор BITMAP:
BitmapID BITMAP [параметры загрузки] [тип памяти] имя файла
Параметры загрузки и тип памяти указывается так же, как и для описанной нами ранее таблицы строк STRINGTABLE, пиктограммы и курсора. В качестве имени файла необходимо указать имя файла, содержащего изображение bitmap, например:
AppBitmap BITMAP mybrush.bmp
Идентификатор изображения bitmap BitmapID можно указывать как символическое имя или как целое число - идентификатор ресурса.

Загрузка изображения bitmap

Для загрузки изображения bitmap вы должны использовать функцию LoadBitmap:
HBITMAP WINAPI LoadBitmap(HINSTANCE hInst,LPCSTR lpszBitmap);
Назначение параметров этой функции аналогично назначению параметров функций LoadIcon и LoadCursor. Параметр hInst указывает идентификатор текущей копии приложения, параметр lpszBitmap - идентификатор bitmap в файле описания ресурсов.
Функция возвращает идентификатор изображения, который следует использовать для рисования bitmap.
Перед завершением работы приложение должно удалить загруженное изображение, вызвав функцию DeleteObject: BOOL WINAPI DeleteObject(HGDIOBJ hGDIObj);
В качестве параметра этой функции следует передать идентификатор изображения, полученный от функции LoadBitmap.

Создание кисти для закрашивания окна

Одно из простых применений изображений bitmap - раскрашивание фона окна. С этой целью вы можете использовать изображения размером 8x8 точек.
Изображение, которое будет использовано для закрашивания внутренней области окна, должно быть определено в файле описания ресурсов при помощи оператора BITMAP.
После загрузки изображения функцией LoadBitmap приложение должно создать из этого изображения кисть, вызвав функцию CreatePatternBrush:
HBRUSH WINAPI CreatePatternBrush(HBITMAP hBmp);
В качестве параметра функции необходимо передать идентификатор изображения bitmap, полученный от функции LoadBitmap.
Функция CreatePatternBrush возвращает идентификатор кисти, который можно использовать при регистрации класса окна. Значение этого идентификатора следует записать в поле hbrBackground структуры wndclass, используемой для регистрации класса окна.
Перед завершением работы приложения созданная кисть должна быть уничтожена. Для этого следует вызвать функцию DeleteObject, передав ей в качестве параметра идентификатор кисти.

8.6. Произвольные данные

Ресурсы приложения Windows могут включать в себя произвольные данные, такие как таблицы перекодировки, звуковые данные и т. п. При необходимости приложение может загрузить эти данные в память и делать с ними все что угодно.

Включение произвольных данных в ресурсы приложения

Для включения произвольных данных в ресурсы приложения файл описания ресурсов должен содержать один или несколько операторов следующего вида:
RId [тип ресурса] [параметры загрузки] [тип памяти] имя файла
RId является идентификатором ресурса. Можно использовать любое имя или число. Для типа ресурса вы можете выбрать любое обозначение. Разумеется, не следует выбирать названия предопределенных ресурсов, такие как ICON или CURSOR.

Параметры загрузки и тип памяти указывается так же, как и для других описанной нами ранее ресурсов. В качестве имени файла необходимо указать имя файла, содержащего данные, например: Hello SOUND hello.wav
Вся работа по подготовке файла с данными выполняется программистом перед сборкой проекта.

Загрузка произвольных данных из ресурсов приложения

Получение идентификатора произвольных данных из ресурсов приложения - трехступенчатый процесс. Вначале с помощью функции FindResource необходимо определить расположение ресурса в файле приложения:
HRSRC WINAPI FindResource(HINSTANCE hInst, LPCSTR lpszName, LPCSTR lpszType);
Параметр hInst является идентификатором модуля, содержащего ресурс. Для извлечения ресурса из приложения вы должны указать его идентификатор, передаваемый функции WinMain через параметр hInstance.
Параметр lpszName должен содержать адрес имени ресурса. Для загрузки произвольных данных в качестве этого параметра следует передать указатель на строку, содержащую идентификатор ресурса. В приведенном выше примере используется идентификатор "Hello".
Параметр lpszType - адрес строки, содержащий тип ресурса. Для нашего примера это должна быть строка "SOUND".
Таким образом, поиск ресурса, описанного как Hello SOUND hello.wav
должен выполняться следующим образом:

```
 HRSRC hRsrc;
hRsrc = FindResource(hInstance, "Hello", "SOUND");
```

Функции FindResource в качестве третьего параметра можно передавать идентификаторы предопределенных типов ресурсов, список которых приведен ниже (некоторые из перечисленных типов ресурсов вам пока незнакомы, мы расскажем о них позже).

Идентификатор ресурса	Название ресурса
RT_ACCELERATOR	Таблица акселераторов
RT_BITMAP	Изображение bitmap
RT_CURSOR	Курсор
RT_DIALOG	Диалоговая панель
RT_FONT	Шрифт
RT_FONTDIR	Каталог шрифтов
RT_ICON	Пиктограмма
RT_MENU	Меню
RT_RCDATA	Произвольные данные
RT_STRING	Таблица строк

Вы можете использовать функцию FindResource для загрузки таких ресурсов, как пиктограммы или курсоры, указав ей тип ресурса, соответственно, RT_ICON или RT_CURSOR. Однако в документации к SDK сказано, что загрузку предопределен-

ных ресурсов, таких как пиктограммы и курсоры, следует выполнять специально предназначенными для этого функциями (LoadIcon, LoadCursor и т. д.).
На втором этапе, после того как ресурс найден, его следует загрузить, вызвав функцию LoadResource:
HGLOBAL WINAPI LoadResource(HINSTANCE hinst, HRSRC hrsrc);
Параметр hinst представляет собой идентификатор модуля, из файла которого загружается ресурс. Если ресурс загружается из файла вашего приложения, используйте значение hInstance, полученное через соответствующий параметр функции WinMain.
В качестве второго параметра этой функции следует передать значение, полученное от функции FindResource.
Третий этап заключается в фиксировании ресурса в оперативной памяти функцией LockResource: void FAR* WINAPI LockResource(HGLOBAL hGlb);
В качестве параметра hGlb функции LockResource следует передать идентификатор ресурса, полученный от функции LoadResource.
Функция LockResource фиксирует данные в памяти и возвращает дальний указатель на соответствующий буфер. После фиксирования Windows не станет удалять сегмент с ресурсами из памяти, так что приложение сможет использовать данные в любой момент времени.
После того как приложение использовало ресурс и он стал ненужен, следует расфиксировать память ресурса, вызвав функцию UnlockResource. Функция определена через функцию GlobalUnlock следующим образом:
BOOL WINAPI GlobalUnlock(HGLOBAL hGlb);
#define UnlockResource(h) GlobalUnlock(h)
Эти, а так же другие функции управления памятью мы рассмотрим позже, в отдельной главе, посвященной управлению памятью в операционной системе Windows.
Перед завершением работы приложения следует освободить полученный ресурс, вызвав функцию FreeResource:
BOOL WINAPI FreeResource(HGLOBAL hGlb);
В качестве параметра hGlb следует передать идентификатор ресурса, полученный от функции LoadResource.

9.Вывод в окна Windows.

9.1. Контекст отображения.

Перед любой операцией вывода текста или графики в памяти должна быть сформирована структура, описывающая окно и содержащая такие характеристики как цвет кисти и пера для рисования, шрифт, толщина и вид линий и т.п. Эта операция называется получением контекста и может быть выполнена при обработке сообщения WM_PAINT (посылаемого окну при необходимости его перерисовки – при изменении размеров или позиции) с помощью функции

```
HDC  hdc = BeginPaint(hwnd, &ps);
….//Что-то выводим
EndPaint(hwnd,&ps);
```

(**Внимание:** функции BeginPaint и EndPaint используются только при обработке сообщения WM_PAINT , во всех остальных случаях контекст отображения получают и удаляют с помощью функций GetDC(hwnd), ReleaseDC(hwnd, hdc))
Первый аргумент BeginPaint – идентификатор окна, а второй – указатель на структуру PAINTSTRUCT:

```
typedef struct tagPAINTSTRUCT
{
  HDC  hdc;
  BOOL fErase;
  RECT rcPaint;
  BOOL fRestore;
  BOOL fIncUpdate;
  BYTE rgbReserved[16];
} PAINTSTRUCT;
```

Поле hdc после вызова функции BeginPaint будет содержать идентификатор контекста отображения (тот же самый, что и возвращаемый самой функцией BeginPaint).
Начальные координаты и размер области, подлежащей обновлению в процессе обработки сообщения WM_PAINT, передаются через поле rcPaint. Это поле представляет собой структуру типа RECT, описывающую прямоугольную область:

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

Координаты задаются в единицах измерения, называемых пикселами. Пикселы - это маленькие прямоугольники, из которых строится изображение. Приложение может определить размер любого своего окна или размер используемого шрифта при помощи специальных функций, которые мы рассмотрим позже.
Поле fErase структуры PAINTSTRUCT определяет необходимость стирания фона окна в области, подлежащей обновлению. Если это поле установлено в состояние TRUE, функция BeginPaint посылает функции окна сообщение WM_ERASEBKGND.
Как правило, сообщение WM_ERASEBKGND передается функции DefWindowProc, которая при получении этого сообщения перерисовывает фон соответствующей области окна (используя кисть, определенную при регистрации класса окна). Если поле fErase содержит значение FALSE, фон окна не изменяется.
Остальные поля структуры PAINTSTRUCT используются Windows, приложение не должно изменять их содержимое.
Если окно содержит несколько областей, подлежащих обновлению, приложение получает только одно сообщение WM_PAINT, в котором определена область, охватывающая все указанные выше области.
Рассмотрим некоторые функции, имеющие отношение к сообщению WM_PAINT.
Функция UpdateWindow имеет следующий прототип:

```
void UpdateWindow(HWND hwnd);
```

Эта функция посылает сообщение WM_PAINT функции окна, идентификатор которого задан в качестве параметра hwnd. Сообщение посылается в обход очереди сообщений приложения, и только в том случае, если для окна существует непустая область обновления. Если в окне обновлять нечего, сообщение WM_PAINT не посылается.

При помощи функции InvalidateRect вы можете объявить любую область окна как требующую обновления. Прототип функции:
void InvalidateRect(HWND hwnd, LPRECT lprc, BOOL fErase);
Первый параметр (hwnd) функции является идентификатором окна, для которого выполняется операция. Второй параметр (lprc) - дальний указатель на структуру типа RECT, определяющую прямоугольную область, подлежащую обновлению. Третий параметр (fErase) определяет необходимость стирания фона окна. Если этот параметр задан как TRUE, фон окна подлежит стиранию (см. поле fErase структуры PAINTSTRUCT).
Функция ValidateRect удаляет прямоугольную область из списка областей, подлежащих обновлению. Она имеет следующий прототип:
void ValidateRect(HWND hwnd, LPRECT lprc);
Первый параметр определяет идентификатор окна, второй является дальним указателем на структуру типа RECT, определяющую прямоугольную область, которая должна быть исключена из области обновления. Если в результате вызова функции ValidateRect в окне не остается областей, подлежащих обновлению, сообщение WM_PAINT удаляется из очереди сообщений приложения.
Подводя итог, отметим еще раз несколько моментов, важных для понимания методов, с помощью которых приложение Windows рисует в своих окнах.
Приложение должно выполнять вывод в окно "централизованно" в функции окна при получении сообщения WM_PAINT.
При обработке сообщения WM_PAINT для увеличения скорости работы следует использовать координаты области окна, подлежащей обновлению, хотя можно обновить и все окно. Используя функции API, приложение в любой момент времени может определить любую область окна как подлежащую (или не подлежащую) обновлению и послать самому себе в соответствующую функцию окна сообщение WM_PAINT.

Контекст отображения может использоваться функциями вывода текста или графики, которые будут рассмотрены позже.

Тип контекста отображения устанавливается с помощью задаваемого при регистрации окна стиля:
Общий контекст – wc.style=0; Контекст для класса окна – wc.style=CS_CLASSDC;
Личный контекст - wc.style=CS_OWNDC; Родительский контекст - wc.style=CS_PARENTDC;
Контекст физического устройства не получают, а создают с помощью функции
HDC WINAPI CreateDC(
LPCSTR lpszDriver, //Имя драйвера
LPCSTR lpszDumшcy, //Имя устройства
LPCSTR lpszOutput,   // Имя файла или порта
Const void FAR* lpvInitData//Данные для инициализации
);
Пример:
hdc = CreateDC("epson9", "Epson FX-850", "LPT2:", NULL);

Созданный при помощи функции CreateDC контекст устройства следует удалить (но не освободить), вызвав функцию DeleteDC(HDC hdc).

Контекст для устройства DISPLAY

В некоторых случаях требуется получить контекст отображения, позволяющий приложению рисовать в любом месте экрана дисплея. Такой контекст можно создать при помощи функции CreateDC, указав в качестве имени драйвера строку "DISPLAY ", а в качестве остальных параметров - значение NULL:

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

В данном случае будет создан контекст для видеомонитора, с помощью которого приложение может рисовать в любом месте экрана. Начало системы координат, выбранной в данный контекст, располагается в верхнем левом углу экрана видеомонитора.

Контекст для памяти

В работе с битовыми изображениями bitmap часто используется такое "устройство вывода", как оперативная память. Приложение может полностью подготовить изображение в оперативной памяти, получив контекст для памяти , и затем быстро вывести готовое изображение на экран. Этот способ во многих случаях работает намного быстрее и приятнее для пользователя, чем формирование изображения непосредственно на экране.

Контекст для памяти создается совместимым с тем контекстом отображения, в котором будет выполняться вывод на физическое устройство. Для создания совместимого контекста используется функция CreateCompatibleDC :

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

Созданный таким образом контекст памяти удаляется при помощи функции DeleteDC.

Использование контекста памяти будет подробно описано при рассмотрении битовых изображений bitmap.

Контекст для метафайла

Контекст для метафайла позволяет записывать команды GDI в файл и затем проигрывать такой файл на физическом устройстве вывода. Файл может находиться в памяти или на диске, в последнем случае его можно использовать для переноса графического изображения в другое приложение.

Для создания контекста метефайла используется функция CreateMetaFile :

```
HDC WINAPI CreateMetaFile(LPCSTR lpszFileName);
```

Параметр lpszFileName должен указывать на строку, содержащую путь к имени файла, в который будут записаны команды GDI, или NULL. В последнем случае создается метафайл в оперативной памяти.

После выполнения рисования в контексте метафайла следует закрыть метафайл, вызвав функцию CloseMetaFile :

```
HMETAFILE WINAPI CloseMetaFile(HDC hdc);
```

Эта функция закрывает метафайл для контекста hdc и возвращает идентификатор метафайла. Идентификатор закрытого метафайла использовать нельзя, так как он не содержит никакой полезной информации.

Что можно сделать с полученным идентификатором метафайла?

Можно скопировать метафайл в обычный дисковый файл, вызвав функцию CopyMetaFile :
HMETAFILE WINAPI CopyMetaFile(HMETAFILE hmf, LPCSTR lpszFileName);
Параметр hmf определяет метафайл, параметр lpszFileName содержит путь к имени файла, в который будет записан метафайл .
Можно проиграть метафайл в контексте отображения или контексте устройства, вызвав функцию PlayMetaFile :
BOOL WINAPI PlayMetaFile(HDC hdc, HMETAFILE hmf);
Наконец, при помощи функции DeleteMetaFile можно удалить метафайл:
BOOL WINAPI DeleteMetaFile(HMETAFILE hmf);
Удаление метафайла с помощью функции DeleteMetaFile делает недействительным идентификатор метафайла hmf и освобождает оперативную память, занятую метафайлом. Если метафайл был создан как обычный дисковый файл, функция DeleteMetaFile не удаляет его с диска.
Для того чтобы воспользоваться метафайлом, хранящимся в виде дискового файла, его следует загрузить при помощи функции GetMetaFile , указав ей в качестве единственного параметра путь к соответствующему файлу:
HMETAFILE WINAPI GetMetaFile(LPCSTR lpszFileName);

9.2.Режимы отображения

Режим отображения - это атрибут контекста отображения, влияющий на используемую функциями GDI систему координат. Для обеспечения независимости приложений от аппаратного обеспечения приложения Windows работают с логическими координатами, которые отображаются в физические. Приложения Windows могут не знать номер используемого видеорежима и соответствующее ему разрешение по вертикали и горизонтали в пикселах, определяя размеры элементов формируемого изображения в миллиметрах или дюймах.

В качестве единицы измерения длины в системе физических координат всегда используется пиксел. Если устройством вывода является экран монитора, физические координаты обычно называют экранными координатами.
Логические координаты передаются функциям GDI, выполняющим рисование фигур или вывод текста. Используемые единицы измерения зависят от режима отображения.
При отображении GDI преобразует логические координаты в физические. Способ преобразования зависит от режима отображения - расположения начала системы координат для окна, расположения начала системы физических координат, масштабов осей для окна и масштабов осей физических координат.

Физическая система координат

Начало этой системы координат располагается в левом верхнем углу экрана. Ось X направлена слева направо, ось Y - сверху вниз. В качестве единицы длины в данной системе координат используется пиксел.
Для того чтобы определить физическое разрешение устройства вывода (например, размер экрана в пикселах по вертикали и горизонтали), следует использовать функцию GetDeviceCaps. Если передать в качестве второго параметра этой функции значения VERTRES и HORZRES , она в любом режиме отображения вернет, соответ-

ственно, размер экрана в пикселах по вертикали и по горизонтали. Параметр hdc должен указывать информационный контекст или контекст отображения, связанный с экраном монитора:

```
HDC  hdc;
int iVertRes, iHorzRes;
hdc = CreateDC("DISPLAY", NULL, NULL, NULL);
iVertRes = GetDeviceCaps(hdc, VERTRES);
iHorzRes = GetDeviceCaps(hdc, HORZRES);
DeleteDC(hdc);
```

Физическая система координат "привязана" к физическому устройству вывода, поэтому при ее использовании для вывода изображения следует учитывать особенности видеоконтроллера.

Недостатки физической системы координат:

-Вертикальное (VERTRES) и горизонтальное (HORZRES) разрешение зависит от типа видеоконтроллера.

-Физические размеры пикселов (ASPECTX и ASPECTY), и, что самое главное, отношение высоты и ширины пиксела также зависят от типа видеоконтроллера.

Если приложению требуется нарисовать, например, окружность или квадрат, при использовании физической системы координат придется учитывать форму пикселов, выполняя масштабирование изображения по одной из осей координат. В противном случае вместо окружности и квадрата на экране появятся эллипс и прямоугольник.

Логическая система координат

Приложения Windows могут использовать одну из нескольких логических систем координат , устанавливая соответствующий режим отображения в контексте отображения. При этом можно использовать любое направление координатных осей и любое расположение начала координат. Например, возможна система координат, в которой задаются положительные и отрицательные координаты по любой оси (разумеется только по горизонтали и вертикали).

Для установки режима отображения, непосредственно определяющего направление осей и размер логической единицы системы координат, используется функция SetMapMode :

```
int WINAPI SetMapMode(HDC hdc, int nMapMode);
```

Для контекста отображения hdc эта функция устанавливает новый режим отображения, заданный параметром nMapMode, возвращая номер режима отображения, который был установлен раньше.

Параметр nMapMode может принимать одно из следующих значений.

Режим отображения	Направление оси X	Направление оси Y	Размер одной логической единицы
MM_TEXT	Вправо	Вниз	1 пиксел
MM_LOMETRIC	Вправо	Вверх	0,1 мм
MM_HIMETRIC	Вправо	Вверх	0,01 мм
MM_LOENGLISH	Вправо	Вверх	0,01 дюйм
MM_HIENGLISH	Вправо	Вверх	0,001 дюйм
MM_TWIPS	Вправо	Вверх	1/1440 дюйма
MM_ISOTROPIC	Можно выбирать	Можно выбирать	Произвольный, одинаковый для осей X и Y
MM_ANISOTROPIC	Можно выбирать	Можно выбирать	Произвольный, может быть разный для осей X и Y

Как видно из этой таблицы, в режиме отображения MM_TEXT, выбранном в контекст отображения по умолчанию, используется нестандартное (для геометрии, математики и физики) направление оси Y - вниз от начала координат. Мы уже говорили, что такое направление оси Y удобно для отображения текста, поэтому этот режим отображения иногда называют текстовым.

Нетрудно заметить, что в режиме MM_TEXT логическая единица длины полностью соответствует физической, поэтому при рисовании геометрических фигур возможны искажения формы. Эти искажения связаны с тем, что форма пиксела для некоторых видеоконтроллеров может быть отличной от квадратной. Режим MM_TEXT неудобен для рисования фигур.

В режимах MM_LOMETRIC, MM_HIMETRIC, MM_LOENGLISH, MM_HIENGLISH, MM_TWIPS используется более привычное направление осей координат и единицы длины, не зависящие от аппаратного обеспечения устройства вывода.

В режиме MM_ISOTROPIC вы можете выбирать произвольное направление осей координат и произвольный (но одинаковый) масштаб для осей X и Y.

Режим MM_ANISOTROPIC еще более универсален. Он позволяет устанавливать произвольное направление осей координат, произвольный масштаб для осей координат, причем для каждой оси можно установить свой собственный масштаб.

Во всех режимах отображения, кроме MM_TEXT и MM_ANISOTROPIC с разным масштабом для осей X и Y, приложение может не заботиться о "квадратуре пиксела", так как масштаб по осям координат одинаковый и не зависит от особенностей устройства вывода.

В режиме MM_TWIPS используется единица длины twip (от twentieth of a point - двадцатая часть точки, или, в терминах полиграфии, двадцатая часть пункта). Размер одного пункта приблизительно равен 1/62 дюйма, следовательно размер единицы длины twip равен 1/1440 дюйма.

С помощью функции GetMapMode приложение может в любой момент времени определить номер режима отображения, выбранный в контекст отображения hdc:

```
int WINAPI GetMapMode(HDC hdc);
```

С помощью переменных xViewOrg и yViewOrg можно изменить расположение начала физических координат. По умолчанию в контексте отображения значения атрибутов, соответствующих этим переменным, равны 0. Приложение может сместить начало координат, изменив значения переменных xViewOrg и yViewOrg.

Для перемещения начала логической системы координат приложение может изменить значения переменных xWinOrg и yWinOrg, которые также определены как атрибуты контекста отображения. По умолчанию в этих переменных находятся нулевые значения.

Переменные xViewExt, yViewExt, xWinExt и yWinExt (вернее, их отношения) задают масштаб, который используется в процессе преобразования координат. Этот масштаб зависит от установленного режима отображения. Приложения могут изменить его только в режимах MM_ISOTROPIC и MM_ANISOTROPIC , для остальных режимов отображения используются фиксированные значения.

Есть специальные функции, предназначенные для преобразования логических координат в физические и физических в логические. Это функции LPtoDP и DPtoLP.

Функция LPtoDP выполняет преобразование логических координат в физические , причем одновременно можно преобразовать несколько пар координат, что ускоряет работу приложения за счет сокращения количества вызовов функции:

```
BOOL WINAPI LPtoDP(
  HDC hdc,              // идентификатор контекста отображения
  POINT FAR* lppt, // указатель на массив структур POINT
  int cPoints);          // размер массива структур POINT
```

Параметр hdc указывает контекст отображения, для которого требуется выполнить преобразования. В процессе преобразования используются атрибуты контекста, такие как смещение и масштаб осей координат.

Через параметр lppt передается указатель на массив структур POINT, в котором находятся преобразуемые координаты. Размер массива определяется значением параметра cPoints. Структура POINT определена в файле windows.h следующим образом:

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

Обратное преобразование (физических координат в логические) выполняется функцией DPtoLP :

```
BOOL WINAPI DPtoLP(
  HDC hdc,              // идентификатор контекста отображения
  POINT FAR* lppt, // указатель на массив структур POINT
  int cPoints);          // размер массива структур POINT
```

Назначение параметров функции DPtoLP и возвращаемое ей значение аналогично назначению параметров и возвращаемому значению функции LPtoDP.

Есть еще две функции, предназначенные для преобразования координат - ClientToScreen и ScreenToClient.

Функция ClientToScreen выполняет преобразование координат в системе координат, связанной с внутренней областью окна, в экранные координаты:

```
void WINAPI ClientToScreen(HWND hwnd, POINT FAR* lppt);
```

Параметр hwnd содержит идентификатор окна, для которого выполняется преобразование.
Параметр lppt содержит указатель на структуру типа POINT, в которую перед вызовом функции следует записать преобразуемые координаты.
Функция ScreenToClient имеет аналогичные параметры, но выполняет обратное преобразование:

```
void WINAPI ScreenToClient(HWND hwnd, POINT FAR* lppt);
```

Режим MM_TEXT

Режим отображения MM_TEXT устанавливается в контексте отображения по умолчанию. В этом режиме невозможно изменить масштаб осей координат. Поэтому логическая единица длины в режиме отображения MM_TEXT равна физической, т. е. одному пикселу. Тем не менее, приложение может изменить смещение физической или логической системы координат, изменив, соответственно, значение пар переменных (xViewOrg, yViewOrg) и (xWinOrg,yWinOrg). Для установки смещения можно использовать функции SetViewportOrg и SetWindowOrg.
Функция SetViewportOrg устанавливает смещение физической системы координат:

```
 DWORD WINAPI SetViewportOrg(
 HDC hdc,      // контекст отображения
 int nXOrigin,  // новое значение для xViewOrg
 int nYOrigin); // новое значение для yViewOrg
```

Для контекста отображения hdc эта функция устанавливает новое расположение начала физической системы координат. Младшее и старшее слово возвращаемого значения содержат, соответственно, предыдущие x- и y-координаты начала физической системы координат.
С помощью функции SetWindowOrg вы можете установить начало логической системы координат:

```
 DWORD WINAPI SetWindowOrg(
 HDC hdc,      // контекст отображения
 int nXOrigin,  // новое значение для xWinOrg
 int nYOrigin); // новое значение для yWinOrg
```

По своему смыслу параметры nXOrigin и nYOrigin являются логическими x- и y-координатами верхнего угла окна, используемого для отображения .
Как правило, приложения изменяют либо начало физических координат, вызывая функцию SetViewportOrg, либо начало логических координат, вызывая функцию SetWindowOrg, хотя, в принципе, вы можете изменить и то, и другое.
В программном интерфейсе Windows есть новые варианты описанных выше двух функций, которые называются SetViewportOrgEx и SetWindowOrgEx . Они отличаются более удобным способом передачи старых координат начала соответствующей системы координат:

```
BOOL WINAPI SetViewportOrgEx(
 HDC hdc,        // контекст отображения
 int nXOrigin,    // новое значение для xWinOrg
 int nYOrigin,    // новое значение для yWinOrg
 POINT FAR* lppt); // указатель на структуру POINT
```

```
BOOL WINAPI SetWindowOrgEx(
  HDC hdc,          // контекст отображения
  int nXOrigin,     // новое значение для xWinOrg
  int nYOrigin,     // новое значение для yWinOrg
  POINT FAR* lppt); // указатель на структуру POINT
```

В структуру, адрес которой передается через параметр lppt, записываются старые координаты начала системы координат. Обе функции возвращают TRUE в случае успеха и FALSE при возникновении ошибки.

В любой момент времени вы можете определить расположение начала физических или логических координат, если воспользуетесь функциями GetViewportOrg и GetWindowOrg (или их более новыми аналогами - GetViewportOrgEx и GetWindowOrgEx).

Функция GetViewportOrg возвращает x- и y-координаты начала физической системы координат для контекста отображения hdc:

```
DWORD WINAPI GetViewportOrg(HDC hdc);
```

Младшее и старшее слово возвращаемого значения содержат, соответственно, предыдущие x- и y-координаты начала физической системы координат. Функция GetWindowOrg возвращает x- и y-координаты начала логической системы координат:

```
DWORD WINAPI GetWindowOrg(HDC hdc);
```

Метрические режимы отображения

Режим MM_LOMETRIC , наряду с режимами MM_HIMETRIC , MM_LOENGLISH , MM_HIENGLISH и MM_TWIPS, относится к метрическим режимам. Эти режимы отображения позволяют использовать привычные единицы измерения, такие как миллиметры и дюймы.

Приложение не может изменить значения переменных xViewExt, yViewExt, xWinExt и yWinExt, от которых зависит масштаб по осям координат. Отношения xViewExt/xWinExt и yViewExt/yWinExt имеют фиксированное значение для каждого из метрических режимов отображения.

Для этих режимов отношение yViewExt/yWinExt имеет отрицательный знак, в результате чего ось Y оказывается направленной снизу вверх.

Сразу после переключения в метрический режим отображения ось X окажется направленной слева направо, а ось Y - снизу вверх. Точка с координатами (0,0) будет находиться в верхнем левом углу экрана, поэтому для того чтобы нарисовать что-нибудь в такой системе координат, вам придется для y-координаты графических объектов использовать отрицательные числа. Для того чтобы система координат приняла более удобный вид, можно переместить начало физических координат в нижний левый угол окна или в центр окна.

Прежде, чем выполнять перемещение начала координат, следует определить размеры внутренней области окна. Это можно сделать при обработке сообщения WM_SIZE :

```
static short cxClient, cyClient;
....
case WM_SIZE:
```

```
{  cxClient = LOWORD(lParam);  cyClient = HIWORD(lParam);  ....  return 0;}
```
Для того чтобы расположить начало координат в левом нижнем углу окна, следует вызвать функцию SetViewportOrg, передав ей новые координаты начала физической системы координат (0,cyClient):
```
SetViewportOrg(hdc, 0, cyClient);
```
Аналогичным образом можно расположить начало системы координат в середине окна (рис. 2.6), обеспечив возможность использования положительных и отрицательных координат вдоль оси X и Y:
```
SetViewportOrg(hdc, cxClient/2, cyClient/2);
```
Режимы MM_ISOTROPIC и MM_ANISOTROPIC

Режимы отображения MM_ISOTROPIC (изотропный) и MM_ANISOTROPIC (анизотропный) допускают изменение направления осей X и Y. В изотропном режиме отображения MM_ISOTROPIC масштаб вдоль осей X и Y всегда одинаковый (т. е. для обоих осей используются одинаковые логические единицы длины). Анизотропный режим MM_ANISOTROPIC предполагает использование разных масштабов для разных осей (хотя можно использовать и одинаковые масштабы).
Для изменения ориентации и масштаба осей предназначены функции SetViewportExt, SetViewportExtEx, SetWindowExt и SetWindowExtEx.
Функция SetWindowExt устанавливает для формулы значения переменных xWinExt и yWinExt:
```
 DWORD WINAPI SetWindowExt(
 HDC hdc,       // идентификатор контекста отображения
 int nXExtent,  // значение для xWinExt
 int nYExtent); // значение для yWinExt
```
Функция SetViewportExt должна использоваться после функции SetWindowExt. Она устанавливает для преобразования координат значения переменных xViewExt и yViewExt:
```
 DWORD WINAPI SetViewportExt(
 HDC hdc,       // идентификатор контекста отображения
 int nXExtent,  // значение для xViewExt
 int nYExtent); // значение для yViewExt
```
Обе функции возвращают в младшем и старшем слове предыдущие значения соответствующих переменных для оси X и Y.
Функции SetWindowExt передаются значения, соответствующие логическому размеру логического окна, в которое будет выполняться вывод, а функции SetViewportExt - реальные ширина и высота реального окна.
Например, нам надо создать систему координат, в которой начало отсчета расположено в левом нижнем углу окна, ось X направлена слева направо, а ось Y - снизу вверх. Высота и ширина должны изменяться от 0 до 32666 .
Если требуется получить одинаковый масштаб по осям X и Y, нужно использовать изотропный режим отображения MM_ISOTROPIC.
Приведем фрагмент кода, создающий необходимый режим отображения.
```
SetMapMode(hdc, MM_ISOTROPIC);
SetWindowExt(hdc, 32666, 32666);
SetViewportExt(hdc, cxClient, -cyClient);
```

```
SetViewportOrg(hdc, 0, cyClient);
```
В изотропном режиме отображения при изменении размеров окна Windows настроит систему координат таким образом, чтобы масштаб по осям X и Y был одинаковый.
Если ширина окна больше высоты, масштаб по горизонтальной оси настраивается таким образом, что логическое окно будет расположено в левой части внутренней области окна. Если же высота окна больше его ширины, при использовании изотропного режима отображения логическое окно окажется в нижней части внутренней области окна. При использовании анизотропного режима отображения MM_ANISOTROPIC настройка масштаба не выполняется, поэтому логическое окно будет занимать всю внутреннюю поверхность окна при любом изменении размеров этого окна.
В программном интерфейсе Window есть новые функции, предназначенные для изменения масштабов осей. Это функции SetViewportExtEx и SetWindowExtEx :

```
BOOL WINAPI SetViewportExtEx(
  HDC hdc,       // идентификатор контекста отображения
  int nXExtent,  // значение для xViewExt
  int nYExtent,  // значение для yViewExt
  SIZE FAR* lpSize); // указатель на структуру SIZE

BOOL WINAPI SetWindowExtEx(
  HDC hdc,       // идентификатор контекста отображения
  int nXExtent,  // значение для xWinExt
  int nYExtent,  // значение для yWinExt
  SIZE FAR* lpSize); // указатель на структуру SIZE
```

От функций SetViewportExt и SetWindowExt эти функции отличаются тем, что старые значения переменных, определяющих масштаб преобразования, записываются в структуру SIZE, указатель на которую передается через параметр lpSize.
Изотропный режим отображения удобно использовать в тех случаях, когда надо сохранить установленное отношение масштабов осей X и Y при любом изменении размеров окна, в которое выводится изображение .
Анизотропный режим удобен в тех случаях, когда изображение должно занимать всю внутреннюю поверхность окна при любом изменении размеров окна. Соотношение масштабов при этом не сохраняется .

9.3. Вывод текста в окна Windows.

Для вывода текста в окно вам предоставляется несколько функций:

```
1)  BOOL TextOut(
    HDC hdc,                    // handle of device context
    int nXStart,                // x-coordinate of starting position
    int nYStart,                // y-coordinate of starting position
    LPCTSTR lpString,           // address of string
    int cbString                // number of characters in string
   );

2) BOOL WINAPI ExtTextOut(HDC hdc,
```

int nXStart, int nYStart,
UINT fuOptions, const RECT FAR* lprc,
LPCSTR lpszString, UINT cbString, int FAR* lpDx);

Эта функция может выводить текст внутри ограничивающей области, при этом для области можно задать цвет фона.

Параметр hdc задает идентификатор контекста отображения, который используется для вывода текста. Параметры nXStart и nYStart определяют соответственно X- и Y-координаты начальной позиции вывода текста. Если перед вызовом этой функции вы установили режим обновления текущей позиции (вызвав функцию SetTextAligh с параметром TA_UPDATECP), параметры nXStart и nYStart игнорируются. Текст будет выведен начиная с текущей позиции, которая устанавливается за последним выведенным ранее символом.

Параметр fuOptions позволяет определить тип ограничивающей прямоугольной области, заданной параметром lprc. Этот параметр задается в виде двух флагов, которые можно объединять логической операцией ИЛИ.

Первый флаг имеет имя ETO_CLIPPED. Если указан этот флаг, прямоугольная область, заданная параметром lprc, определяет область ограничения для вывода текста.

Второй флаг имеет имя ETO_OPAQUE. Этот флаг позволяет закрасить прямоугольную область цветом, заданным при помощи функции SetBkColor.

Параметр lprc является дальним указателем на структуру типа RECT. Он определяет прямоугольную область, используемую для ограничения или закрашивания. В качестве этого параметра вы можете указать значение NULL, при этом область ограничения не используется.

Для определения адреса выводимой текстовой строки следует указать параметр lpszString. Этот параметр является указателем на строку символов. Длина строки символов задается в байтах параметром cbString.

Параметр lpDx позволяет задать расстояние между отдельными символами. Если этот параметр указан как NULL, при выводе текста расстояние между символами определяется шрифтом, выбранным в контекст отображения. Если же в качестве этого параметра указать адрес массива значений типа int, вы сможете определять индивидуальное расстояние между отдельными символами. Элемент массива с индексом n определяет расстояние между n-м и n+1-м символами строки. Размер массива должен быть равен значению, указанному в параметре cbString.

Функция ExtTextOut при нормальном завершении возвращает ненулевое значение (TRUE). В противном случае возвращается значение FALSE.

3) LONG WINAPI TabbedTextOut(HDC hdc,
int xPosStart, intyPosStart,
LPCSTR lpszString, int cbString,
int cTabStops, int FAR* lpnTabPositions, int nTabOrigin);

Функция TabbedTextOut предназначена для вывода текстовых строк, содержащих символы табуляции. Эту функцию удобно использовать для вывода текстовых таблиц.

Параметр hdc задает идентификатор контекста отображения, который используется для вывода текста. Параметры xPosStart и yPosStart определяют соответственно X- и Y-координаты начальной позиции вывода текста. Если перед вызовом этой функции вы установили режим обновления текущей позиции (вызвав функцию SetTextAligh с

параметром TA_UPDATECP), параметры xPosStart и yPosStart игнорируются. Текст будет выведен начиная с текущей позиции, которая устанавливается за последним выведенным ранее символом. Для определения адреса выводимой текстовой строки следует указать параметр lpszString. Этот параметр является дальним указателем на строку символов. Длина строки символов задается в байтах параметром cbString. Параметр cTabStops определяет количество значений в массиве позиций символов табуляции. Если значение этого параметра равно 1, расстояние между символами табуляции определяется первым элементом массива, адрес которого передается через параметр lpnTabPositions. Указатель lpnTabPositions определяет адрес массива целых чисел, определяющих расположение символов табуляции. Массив должен быть отсортирован в порядке увеличения значений. Параметр nTabOrigin определяет логическую X-координату начальной позиции, относительно которой происходит расширение символов табуляции. Функция TabbedTextOut возвращает размер (в логических единицах) области, занятой выведенной строкой. Старшее слово возвращаемого значения содержит высоту строки, младшее - ширину строки.

4) int WINAPI DrawText(HDC hdc,
   LPCSTR lpsz, int cb,
   RECT FAR* lprc, UINT fuFormat);

Функция DrawText предназначена для форматированного вывода текста в прямоугольную область окна:

Параметр lprc является дальним указателем на структуру типа RECT, определяющую координаты верхнего левого и правого нижнего углов прямоугольной области, в которую будет выведен текст. Текст может быть выравнен и отформатирован внутри этой области в соответствии со значением, указанным параметром fuFormat. Параметр задается как набор флагов с использованием операции логического ИЛИ:

Значение	Описание
DT_BOTTOM	Выравнивание текста по верхней границе прямоугольника, заданного параметром lprc. Этот флаг должен использоваться в комбинации с флагом DT_SINGLELINE
DT_CALCRECT	Определение высоты и ширины прямоугольника без вывода текста. Если указан этот флаг, функция DrawText возвращает высоту текста. Если выводимый текст состоит из нескольких строк, функция использует ширину прямоугольника, заданную параметром lprc, и расширяет базу этого прямоугольника до тех пор, пока прямоугольник не вместит в себя последнюю строку текста. Если текст состоит из одной строки, функция изменяет правую сторону прямоугольника до тех пор, пока последний символ строки не поместится в прямоугольник. В структуру, заданную параметром lprc, после возврата из функции будут записаны размеры прямоугольной области, использованной для вывода текста
DT_CENTER	Центрирование текста по горизонтали
DT_EXPANDTABS	Расширение символов табуляции. По умолчанию каж-

	дый символ табуляции расширяется в восемь символов
DT_EXTERNALLEADING	Вывод текста выполняется с учетом межстрочного расстояния (external leading), определенного для выбранного шрифта разработчиком шрифтов
DT_LEFT	Выравнивание текста по левой границе прямоугольной области, заданной параметром lprc
DT_NOCLIP	Вывод текста выполняется без ограничения области вывода. Этот режим увеличивает скорость вывода текста
DT_NOPREFIX	Выключение директивы подчеркивания &. По умолчанию символ & используется для того, чтобы вывести следующий символ с выделением подчеркиванием. Для вывода самого символа & его следует повторить дважды. Флаг DT_NOPREFIX выключает этот режим
DT_RIGHT	Выравнивание текста по правой границе прямоугольной области, заданной параметром lprc
DT_SINGLELINE	Текст состоит только из одной строки. Символы возврата каретки и перевода строки не вызывают перехода на следующую строку
DT_TABSTOP	Установить точки останова по символам табуляции
DT_TOP	Выравнивание текста по верхней границе прямоугольной области, заданной параметром lprc. Флаг используется только для текста, состоящего из одной строки
DT_VCENTER	Выравнивание текста по вертикали. Флаг используется только для текста, состоящего из одной строки. Если текст состоит из одной строки, необходимо вместе с этим флагом указывать флаг DT_SINGLELINE
DT_WORDBREAK	Выполнять свертку слов в пределах заданной параметром lprc прямоугольной области. Если слово не помещается в строке, оно может быть перенесено на следующую строку

Если перед вызовом функции вы установили режим обновления текущей позиции (вызвав функцию SetTextAligh с параметром TA_UPDATECP), текст будет выведен начиная с текущей позиции, которая устанавливается за последним выведенным ранее символом. Свертка слов при этом не выполняется.

Функция DrawText в случае успешного завершения возвращает высоту прямоугольной области, использованной для вывода текста.

Функция окна, содержащая простейший вывод текста, может быть такой:

```
LRESULT CALLBACK
WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
  HDC hdc;                  // индекс контекста устройства
  PAINTSTRUCT ps;      // структура для рисования
  switch (msg)
```

```
 {    case WM_PAINT:
   {  hdc = BeginPaint(hwnd, &ps); // Получаем индекс контекста устройства
      TextOut(hdc, 10, 20, "Сообщение WM_PAINT", 18); // Выводим текстовую стро-
ку
       EndPaint(hwnd, &ps); // Отдаем  контекст устройства
        return 0;
   }
    case WM_LBUTTONDOWN:
   { hdc = GetDC(hwnd); // Получаем  контекст
     TextOut(hdc, 10, 40, "Сообщение WM_LBUTTONDOWN", 24);
      ReleaseDC(hwnd, hdc); // Отдаем индекс контекста устройства
     return 0;
   }
  case WM_DESTROY:
   { PostQuitMessage(0);    return 0;  }  }
 return DefWindowProc(hwnd, msg, wParam, lParam);}
```

Выравнивание текста в прямоугольнике вывода.

В контексте отображения можно задать так называемый режим выравнивания текста (text alignment mode). По умолчанию используется выравнивание на левую границу, причем координаты вывода текста указывают верхнюю границу воображаемого прямоугольника, охватывающего текст.

При помощи функции UINT WINAPI SetTextAlign(HDC hdc, UINT fuAlign); можно изменить режим выравнивания.

Функция SetTextAlign возвращает старое значение режима выравнивания.

Первый параметр функции (hdc) указывает идентификатор контекста отображения, для которого надо изменить выравнивание. Это тот самый идентификатор, который возвращается функцией BeginPaint при обработке сообщения WM_PAINT.

Второй параметр (fuAlign) указывает новый режим выравнивания и задается при помощи трех групп флагов. Символические имена флагов определены в файле windows.h и начинаются с префикса TA_.

Первая группа флагов отвечает за выравнивание текстовой строки по горизонтали:

Флаг	Описание
TA_LEFT	Выравнивание по левой границе. Координаты соответствуют левой границе воображаемого прямоугольника, охватывающего текст (используется по умолчанию)
TA_CENTER	Выравнивание по центру. Координаты соответствуют центру воображаемого прямоугольника, охватывающего текст
TA_RIGHT	Выравнивание по правой границе

Вторая группа флагов отвечает за выравнивание текста по вертикали:

Флаг	Описание
TA_TOP	Выравнивание по верхней границе. Координаты соответствуют

	верхней границе воображаемого прямоугольника, охватывающего текст (используется по умолчанию)
TA_BASELINE	Выравнивание по базовой линии выбранного шрифта
TA_BOTTOM	Выравнивание по нижней границе

Третья группа флагов относится к текущей позиции вывода текста:

Флаг	Описание
TA_NOUPDATECP	Не изменять значение текущей позиции вывода текста (используется по умолчанию)
TA_UPDATECP	После использования функций TextOut и ExtTextOut вычислить новое значение текущей позиции вывода текста

Для приложений Windows в контексте отображения вы можете разрешить или запретить использование текущей позиции. Если использование текущей позиции вывода текста разрешено, ее значение будет обновляться при вызове функций вывода текста TextOut и ExtTextOut Из каждой группы флагов можно использовать только один, например:

```
SetTextAlign(hdc, TA_CENTER | TA_BASELINE | TA_UPDATECP);
```

Если задан режим выравнивания TA_UPDATECP, функция TextOut начинает вывод текста с текущей позиции, игнорируя параметры, определяющие расположение текста в окне. В любой момент времени вы можете определить текущий режим выравнивания, вызвав функцию UINT GetTextAlign(hdc);

Эта функция возвращает текущий режим отображения для контекста, указанного функции в качестве параметра.

Установка цвета выводимого текста.

Для этого необходимо вызвать функцию SetTextColor:

```
COLORREF WINAPI SetTextColor(HDC hdc, COLORREF clrref);
```

Параметр hdc определяет контекст отображения, для которого вы собираетесь изменить цвет текста. Структура clrref определяет цвет. Функция SetTextColor возвращает значение цвета, которое использовалось для вывода текста раньше.

Тип данных COLORREF определен в файле windows.h следующим образом:

```
typedef DWORD COLORREF;
```

Для указания цвета удобно использовать макрокоманду RGB, определенную в файле windows.h:

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

В качестве параметров для этой макрокоманды вы можете указывать интенсивность отдельных цветов в диапазоне от 0 до 255:

Параметр	Цвет
r	Красный
g	Зеленый

b	Голубой

Для определения отдельных цветовых компонент из возвращаемого функцией SetTextColor значения удобно использовать макрокоманды GetRValue, GetGValue и GetBValue, определенные в файле windows.h:

```
#define GetRValue(rgb)      ((BYTE)(rgb))
#define GetGValue(rgb)      ((BYTE)(((WORD)(rgb)) >> 8))
#define GetBValue(rgb)      ((BYTE)((rgb)>>16))
```

С помощью этих макрокоманд вы сможете определить значения соответственно для красной, зеленой и голубой компоненты цвета.

В любой момент времени вы можете определить текущий цвет, используемый для вывода текста. Для этого следует вызвать функцию GetTextColor:

```
COLORREF WINAPI GetTextColor(HDC hdc);
```

Для указанного при помощи единственного параметра контекста отображения функция возвращает цвет, используемый для вывода текста.

Текстовые метрики.

Операционная система Windows содержит в себе сложную подсистему для работы со шрифтами. Приложения Windows могут выводить текст с использованием различных шрифтов. Многие программы, такие, как текстовые процессоры, позволяют выбирать произвольную высоту букв. Но характеристики шрифтов в Windows не ограничиваются высотой и шириной букв. Для точного определения внешнего вида шрифта используется больше дюжины различных характеристик.

Все шрифты в Windows можно разделить на две группы. К первой группе относятся шрифты с фиксированной шириной букв (fixed-pitch font). Все буквы (и знаки, такие, как запятая, точка и т. д.) такого шрифта имеют одинаковую ширину. Вторая группа шрифтов - шрифты с переменной шириной букв (variable-pitch font). Для таких шрифтов каждая буква имеет свою ширину. Буква "Ш", например, шире буквы "i".

Кроме того, шрифты Windows можно разделить на **растровые** (raster font), **контурные** (stroke font) и **масштабируемые** (типа TrueType). Растровые шрифты состоят из отдельных пикселов и используются при выводе текста на экран видеомонитора или принтер. Для обеспечения приемлемого качества текста в Windows имеется набор одинаковых растровых шрифтов для нескольких размеров букв. Если попытаться выполнить масштабирование растрового шрифта в сторону увеличения размера букв, наклонные линии и закругления будут изображаться в виде "лестницы" .

Контурные шрифты больше подходят для плоттеров. При масштабировании таких шрифтов можно достигнуть лучших результатов, чем при масштабировании растровых. Однако при большом размере букв результат все равно получается неудовлетворительный.

Масштабируемые шрифты TrueType сохраняют начертание символов при любом изменении размера, поэтому они чаще всего используются, например, при подготовке текстовых документов.

Любой из перечисленных выше шрифтов может быть с фиксированной или переменной шириной букв.

Системный шрифт относится к растровым шрифтам с переменной шириной букв.

Так как ширина букв переменная, вы не сможете выполнять вывод таблицы в несколько столбцов, ориентируясь на ширину букв. Вам придется либо выводить каждый столбец таблицы по отдельности, начиная с некоторой позиции в окне, либо подсчитывать длину каждой ячейки и дополнять ее пробелами до некоторой фиксированной ширины (первый способ нам кажется удобнее).
Переменная ширина букв усложняет задачу вывода текста, так как длина текстовой строки зависит не только от количества букв в строке, но и от того, из каких букв состоит строка. К счастью, в составе программного интерфейса Windows имеется специальная функция GetTextExtent, предназначенная для подсчета длины текстовой строки.
Для получения информации о шрифте, выбранном в контекст устройства, предназначена функция GetTextMetrics. Она имеет следующий прототип:

```
BOOL WINAPI GetTextMetrics(HDC hdc, TEXTMETRIC FAR* lptm);
```

Первый параметр функции (hdc) указывает контекст устройства, для которого требуется получить информацию о метрике шрифта. В качестве этого параметра можно использовать значение, возвращаемое функцией BeginPaint или GetDC.
Второй параметр функции (lptm) является дальним указателем на структуру типа TEXTMETRIC, в которую будет записана информация о метриках шрифта, выбранного в указанный контекст устройства.
В случае успешного завершения функция возвращает значение TRUE, в противном случае - FALSE.
Структура TEXTMETRIC описана в файле windows.h следующим образом:

```
typedef struct tagTEXTMETRIC
{
  int    tmHeight;
  int    tmAscent;
  int    tmDescent;
  int    tmInternalLeading;
  int    tmExternalLeading;
  int    tmAveCharWidth;
  int    tmMaxCharWidth;
  int    tmWeight;
  BYTE   tmItalic;
  BYTE   tmUnderlined;
  BYTE   tmStruckOut;
  BYTE   tmFirstChar;
  BYTE   tmLastChar;
  BYTE   tmDefaultChar;
  BYTE   tmBreakChar;
  BYTE   tmPitchAndFamily;
  BYTE   tmCharSet;
  int    tmOverhang;
  int    tmDigitizedAspectX;
  int    tmDigitizedAspectY;
} TEXTMETRIC;
```

Описание этой структуры мы начнем с полей, определяющих вертикальные размеры шрифта. Программисты MS-DOS пользовались, как правило, одним параметром - высотой букв шрифта. Для приложений Windows используются целых пять параметров, имеющих отношение к вертикальным размерам букв .
Отсчет всех размеров выполняется от так называемой базовой линии шрифта. Общая высота букв находится в поле tmHeight структуры TEXTMETRIC. Эта высота складывается из двух компонент - tmAscent и tmDescent. Компонента tmAscent представляет собой высоту букв от базовой линии с учетом таких элементов, как тильда в букве "Й". Компонента tmDescent определяет пространство, занимаемое буквами ниже базовой линии. Сумма tmAscent и tmDescent в точности равна tmHeight.
Величина tmInternalLeading определяет размер выступающих элементов букв. Эта величина может быть равно нулю. Величина tmExternalLeading определяет минимальный межстрочный интервал, рекомендуемый разработчиком шрифта. Ваше приложение может игнорировать межстрочный интервал, однако в этом случае строки будут соприкасаться друг с другом, что не улучшит внешнего вида окна.
Для определения ширины букв в структуре TEXTMETRIC есть два поля с именами tmAveCharWidth и tmMaxCharWidth. Поле tmAveCharWidth содержит среднее значение ширины строчных букв шрифта. Это значение приблизительно соответствует ширине латинской буквы "x". Поле tmMaxCharWidth определяет ширину самой широкой буквы в шрифте. Для шрифта с фиксированной шириной букв поля tmAveCharWidth и tmMaxCharWidth содержат одинаковые значения, которые зависят от самого шрифта.
Хорошо спроектированные приложения позволяют вам выбирать для отображения текста произвольные шрифты. Поэтому приложение никогда не должно ориентироваться на конкретные размеры шрифта. Вместо этого следует определять эти размеры динамически во время работы приложения с помощью специально предназначенных для этого средств, таких, как функция GetTextMetrics.

9.4.Рисование в окнах.

В программном интерфейсе GDI имеется функция SetPixel , позволяющая нарисовать один пиксел, но рисование линии или окружности не сводится к многократному вызову этой функции - процесс рисования занимал бы очень много времени. На самом деле многие из функций рисования выполняются драйвером или даже аппаратурой видеоконтроллера, что значительно ускоряет вывод. С помощью функции GetDeviceCaps приложение может определить, поддерживает ли драйвер ту или иную функцию рисования. Первый параметр функции hdc задает контекст устройства, для которого необходимо получить информацию о его возможностях. Второй параметр iCapability определяет параметр устройства, значение которого необходимо получить.
Приведем список значений для второго параметра функции GetDeviceCaps, с помощью которых можно определить, какие операции рисования выполняет драйвер устройства вывода.

Имя константы	Описание
LINECAPS	Способности устройства рисовать линии. Возвращаемое значе-

	ние представляет собой набор битовых масок, установленных в 1, если устройство может само рисовать линии различного типа:LC_INTERIORS устройство может закрашивать внутреннюю область;LC_MARKER маркеры;LC_NONE устройство не может рисовать линии;LC_POLYLINE ломаные линии;LC_POLYMARKER линии polymarker;LC_STYLED устройство может рисовать линии с использованием различных стилей (штриховые, пунктирные, штрих пунктирные и т.д.);LC_WIDE широкие линии;LC_WIDESTILED устройство может рисовать широкие линии с использованием различных стилей (штриховые, пунктирные, штрих-пунктирные и т. д.)
CURVECAPS	Способность устройства рисовать различные кривые линии и геометрические фигуры. Возвращаемое значение представляет собой набор битовых масок, установленных в 1, если устройство может само рисовать различные фигуры:CC_CIRCLES окружности;CC_CHORD сегмент эллипса;CC_ELLIPSES эллипсы;CC_INTERIORS устройство может закрашивать внутреннюю область геометрических фигур;CC_NONE устройство не может рисовать кривые линии и геометрические фигуры;CC_PIE секторы эллипса;CC_ROUNDRECT прямоугольники со скругленными углами;CC_STYLED устройство может рисовать рамки с использованием различных стилей (штриховые, пунктирные, штрих-пунктирные и т.д.);CC_WIDE широкие рамки;CC_WIDESTYLED устройство может рисовать широкие рамки с использованием различных стилей (штриховые, пунктирные, штрих-пунктирные и т. д.)
POLYGONALCAPS	Способности устройства рисовать многоугольники. Возвращаемое значение представляет собой набор битовых масок, установленных в 1, если устройство может само рисовать многоугольники различного типа:PC_INTERIORS устройство может закрашивать внутреннюю область;PC_NONE устройство не может рисовать многоугольники;PC_RECTANGLE прямоугольники;PC_SCANLINES устройство может выполнять сканирование линий растра;PC_STYLED устройство может рисовать рамки с использованием различных стилей (штриховые, пунктирные, штрих-пунктирные и т. д.);PC_WIDE широкие рамки;PC_WIDESTILED устройство может рисовать широкие рамки с использованием различных стилей (штриховые, пунктирные, штрих-пунктирные и т. д.)PC_WINDPOLYGON многоугольники с заполнением в режиме WINDING

Для приложения не имеет особого значения, кто именно будет рисовать - видеоконтроллер, драйвер или GDI. Запрос на рисование, например, эллипса, будет выполнен, даже если соответствующая операция не поддерживается драйвером. В последнем случае эллипс будет нарисован самим GDI с использованием более примитивных операций, но процесс рисования займет больше времени.

Современные видеоадаптеры сконструированы таким образом, что большинство основных операций рисования, используемых в операционной системе Windows, выполняются аппаратно. Эти видеоадаптеры иногда называются ускорителями Windows.
Результат рисования геометрических фигур зависит от установки таких атрибутов контекста, как ширина, цвет и стиль линии (определяются выбранным в контекст отображения пером), способ закраски замкнутых фигур (определяется выбранной в контекст отображения кистью), цвета фона, прозрачностью фона (прозрачный режим TRANSPARENT и непрозрачный режим OPAQUE), режимом рисования, режимом закрашивания, областью ограничения, режимом отображения, т. е. практически от всех атрибутов контекста отображения. Поэтому при описании функций мы будем попутно рассматривать способы изменения атрибутов контекста отображения, влияющих на результат их выполнения.

Рисование точки

Функция рисования точки SetPixel устанавливает цвет точки с заданными координатами:

```
COLORREF WINAPI SetPixel(
  HDC hdc,         // контекст отображения
  int nXPos,       // x-координата точки
  int nYPos,       // y-координата точки
  COLORREF clrref); // цвет точки
```

Параметр hdc определяет контекст отображения, для которого необходимо изменить цвет точки. Параметры nXPos и nYPos определяют координаты точки в системе координат, которая зависит от установленного для контекста hdc режима отображения. Последний параметр clrref определяет новый цвет точки.
В файле windows.h есть описание макрокоманды RGB , позволяющей сконструировать цвет в формате COLORREF из отдельных компонент красного (r), зеленого (g) и голубого (b) цвета.
Можно использовать эту макрокоманду совместно с функцией SetPixel для установки, например, красного цвета точки, расположенной в начале системы координат (0,0), следующим образом: SetPixel(hdc, 0, 0, RGB(0xff, 0, 0));
Три параметра макрокоманды RGB позволяют задать любой из примерно 16 млн. цветов и оттенков, однако это не означает, что вы получите на экране точно такой цвет, какой был задан при помощи этой макрокоманды.
Функция SetPixel возвращает цвет, который фактически был использован для рисования точки. Как мы только что заметили, возвращенное значение может отличаться от заданного параметром clrref. В случае ошибки оно будет равно -1.
Функция GetPixel позволяет узнать цвет точки, заданной идентификатором контекста отображения и координатами: COLORREF WINAPI GetPixel(HDC hdc, int nXPos, int nYPos);С помощью следующих трех макрокоманд, определенных в файле windows.h, вы можете определить отдельные цветовые компоненты для значения, возвращаемого функциями SetPixel и GetPixel:

```
#define GetRValue (rgb) ((BYTE)(rgb))
#define GetGValue (rgb) ((BYTE)(((WORD)(rgb)) >> 8))
```

```
#define GetBValue (rgb) ((BYTE)((rgb)>>16))
```
Функции SetPixel и GetPixel используются достаточно редко, так как для построения графических изображений есть более мощные функции.

Рисование линий

Приложения Windows могут рисовать прямые и ломаные линии, а также дуги эллипса (и окружности, как частного случая эллипса). Параметры этих линий определяются несколькими атрибутами контекста отображения. Это режим отображения, влияющий на используемую систему координат, цвет фона, режим фона (прозрачный или непрозрачный), режим рисования, цветовая палитра (в режимах, использующих цветовую палитру), перо (может иметь различный цвет, толщину и стиль).

Текущая позиция пера

Для рисования прямых линий (и только для этого) в контексте отображения хранятся координаты текущей позиции пера . Для изменения текущей позиции пера в Windows есть две функции с именами MoveTo и MoveToEx . Для совместимости с 32-разрядными версиями Windows, такими, как Windows NT, в новых приложениях следует использовать функцию MoveToEx:
```
BOOL WINAPI MoveToEx(
  HDC hdc,        // идентификатор контекста отображения
  int x,          // x-координата
  int y,          // y-координата
  POINT FAR* lppt); // указатель на структуру POINT
```
Для контекста отображения hdc эта функция устанавливает текущую позицию пера, равную (x,y). В структуру типа POINT, на которую указывает параметр lppt, после возврата из функции будут записаны старые координаты пера. Функция MoveToEx возвращает TRUE при нормальном завершении или FALSE при ошибке.
Чтобы узнать текущую позицию пера, приложение может использовать функцию GetCurrentPositionEx : `BOOL WINAPI GetCurrentPositionEx(HDC hdc, POINT FAR* lppt);`После вызова этой функции текущая позиция пера будет записана в структуру типа POINT, на которую указывает параметр lppt. Функция GetCurrentPositionEx возвращает TRUE при нормальном завершении или FALSE при ошибке.

Рисование прямой линии

Для того чтобы нарисовать прямую линию, приложение должно воспользоваться функцией LineTo : `BOOL WINAPI LineTo(HDC hdc, int xEnd, int yEnd);`
Эта функция рисует линию из текущей позиции пера, установленной ранее функцией MoveTo или MoveToEx, в точку с координатами (xEnd,yEnd). После того как линия будет нарисована, текущая позиция пера станет равной (xEnd,yEnd).
Таким образом, для того чтобы нарисовать прямую линию, приложение должно сначала с помощью функции MoveToEx установить текущую позицию пера в точку, которая будет началом линии, а затем вызвать функцию LineTo, передав ей через параметры xEnd и yEnd координаты конца линии.
Особенностью функции LineTo является то, что она немного не дорисовывает линию - эта функция рисует всю линию, не включая ее конец, т. е. точку (xEnd,yEnd).
Преимущества использования отдельных функций для установки текущей позиции и для рисования линии из текущей позиции в заданную точку с последующим изме-

нением текущей позиции проявляются при рисовании ломаных линий. В этом случае вы можете только один раз установить текущую позицию пера на начало ломаной линии и в дальнейшем вызывать только функцию LineTo, передавая ей координаты точек излома линии. Однако для рисования ломаных линий (если известны координаты всех точек излома) больше подходит функция Polyline.

Рисование ломаной линии

Функции Polyline , предназначенной для рисования ломаных линий, следует передать идентификатор контекста отображения hdc, указатель lppt на массив структур POINT, в котором должны находится координаты начала ломаной линии, координаты точек излома и координаты конца ломаной линии, а также размер этого массива cPoints:

```
BOOL WINAPI Polyline(
  HDC hdc,                    // идентификатор контекста отображения
  const POINT FAR* lppt,// указатель на массив структур POINT
  int cPoints);               // размер массива
```

Рисование дуги эллипса

Функция Arc позволяет нарисовать дугу эллипса или окружности:

```
  BOOL WINAPI Arc(
  HDC hdc,                     // идентификатор контекста отображения
  int nxLeft,  int nyTop,     // верхий левый угол
  int nxRight, int nyBottom, // правый нижний угол
  int nxStart, int nyStart,       // начало дуги
  int nxEnd,   int nyEnd);     // конец дуги
```

Параметры (nxLeft,nyTop) и (nxRight,nyBottom) задают координаты, соответственно, верхнего левого и правого нижнего углов воображаемого прямоугольника, в который вписан эллипс.
Начало дуги эллипса определяется пересечением эллипса с воображаемой прямой линией, проведенной из центра эллипса (xC,yC) в точку (xStart,yStart).
Конец дуги определяется аналогично - как пересечение эллипса с воображаемой прямой линией, проведенной из центра эллипса в точку (xEnd,yEnd).
Дуга рисуется в направлении против часовой стрелки.
Координаты центра эллипса (если это потребуется) можно вычислить следующим образом: xC = (nxLeft + nxRight) / 2;yC = (nyTop + nyBottom) / 2;

Настройка атрибутов контекста отображения для рисования линий

Функции, предназначенные для рисования линий, не имеют никаких параметров, определяющих толщину, цвет и стиль линии. Эти, а также другие характеристики (например, режим прозрачности), выбираются при установке соответствующих атрибутов контекста отображения.

Выбор пера

Для рисования линий приложения Windows могут выбрать одно из трех встроенных перьев, либо создать собственное перо.

Для выбора встроенного пера лучше всего воспользоваться макрокомандами GetStockPen и SelectPen , определенными в файле windowsx.h.
Макрокоманда GetStockPen возвращает идентификатор встроенного пера, заданного параметром . Вы можете выбрать для этого параметра одно из следующих значений:

Значение	Описание
BLACK_PEN	Перо, рисующее черную линию толщиной в один пиксел (для любого режима отображения). Это перо выбрано в контекст отображения по умолчанию
WHITE_PEN	Перо белого цвета. Толщина пера также равна одному пикселу и не зависит от режима отображения
NULL_PEN	Невидимое перо толщиной в один пиксел. Используется для рисования замкнутых закрашенных фигур (таких, как прямоугольник или эллипс) в тех случаях, когда контур фигуры должен быть невидимым

После получения идентификатора пера его необходимо выбрать в контекст отображения при помощи макрокоманды SelectPen. Первый параметр этой макрокоманды используется для указания идентификатора контекста отображения, в который нужно выбрать перо, второй - для передачи идентификатора пера. Макрокоманда SelectPen возвращает идентификатор пера, который был выбран в контекст отображения раньше. Вы можете сохранить этот идентификатор и использовать его для восстановления старого пера.
Однако при помощи встроенных перьев вы не можете нарисовать цветные, широкие, штриховые и штрих-пунктирные линии. Если вас не устраивают встроенные перья, вы можете легко создать собственные. Для этого нужно воспользоваться функциями CreatePen или CreatePenIndirect.

Функция CreatePen позволяет определить стиль, ширину и цвет пера:

```
HPEN WINAPI CreatePen(
  int fnPenStyle,        // стиль пера
  int nWidth,            // ширина пера
  COLORREF clrref); // цвет пера
```

Параметр nWidth определяет ширину пера. Используемая при этом единица длины зависит от режима отображения, поэтому вы можете задавать ширину пера не только в пикселах, но и в долях миллиметра или дюйма. Учтите, что для линий PS_DASH, PS_DOT, PS_DASHDOT, PS_DASHDOTDOT можно использовать только единичную или нулевую ширину, в обоих случаях ширина линии будет равна одному пикселу. Поэтому вы не можете создать перо для рисования, например, пунктирной линии шириной 5 миллиметров.
Параметр clrref задает цвет пера.
На первый взгляд линии PS_SOLID и PS_INSIDEFRAME похожи, однако между ними имеются различия, особенно заметные для широких линий. Широкая линия, имеющая стиль PS_SOLID, располагается по обе стороны оси, заданной координатами линии. Линии, имеющие стиль PS_INSIDEFRAME, располагаются внутри контура, определяющего размеры замкнутой фигуры .

Еще одно отличие связано с использованием смешанных цветов (dithered color). Когда Windows не может в точности подобрать цвет, указанный для толстой линии стиля PS_INSIDEFRAME, он раскрашивает эту линию с использованием смешанного цвета, полученного из других цветов. В этом случае изображение линии формируется из отдельных точек разных цветов. Техника смешанных цветов может применяться и при закрашивании замкнутых фигур.
При рисовании тонких линий, а также линий, имеющих другой стиль, используются только чистые цвета.
Небольшое замечание относительно концов толстых линий. Концы толстых линий закруглены . Для изображения толстой линии с прямыми концами следует задать прямоугольную область ограничения . Можно также нарисовать толстую линию как закрашенный прямоугольник с использованием тонкого пера.
Другая возможность создать перо - вызвать функцию CreatePenIndirect :

```
HPEN WINAPI CreatePenIndirect(LOGPEN FAR* lplgpn);
```

Эта функция работает аналогично функции CreatePen, однако в качестве параметра ей необходимо передать указатель на структуру типа LOGPEN, в которой должны находиться характеристики создаваемого пера.

Структура LOGPEN и различные указатели на нее определены в файле windows.h:

```
typedef struct tagLOGPEN
{
  UINT    lopnStyle;      // стиль пера
  POINT   lopnWidth;   // ширина пера
  COLORREF lopnColor;  // цвет пера
} LOGPEN;
typedef LOGPEN*       PLOGPEN;
typedef LOGPEN NEAR* NPLOGPEN;
typedef LOGPEN FAR*  LPLOGPEN;
```

Ширина пера в данном случае находится в поле x структуры POINT. Поле y не используется. Если вы создали перо, его можно выбрать в контекст отображения при помощи макрокоманды SelectPen. После этого можно рисовать линии обычным образом, вызывая функции MoveToEx и LineTo.
Созданные приложением перья принадлежат GDI, соответствующие структуры данных располагаются в его сегменте данных. Поэтому если перо больше не нужно, его нужно удалить для освобождения памяти. Прежде чем удалять созданное вами перо, следует выбрать в контекст отображения одно из встроенных перьев (например то, которое использовалось раньше). После этого для удаления вашего пера нужно вызвать макрокоманду DeleletePen . В качестве параметра этой макрокоманде необходимо передать идентификатор удаляемого пера. Нельзя удалять перо, если оно выбрано в контекст отображения. Нет никакого смысла в удалении встроенных перьев.

Выбор режима фона

Режим фона влияет на заполнение промежутков между штрихами и точками в штрих-пунктирных, штриховых и пунктирных линиях.
По умолчанию в контексте отображения установлен непрозрачный режим фона OPAQUE . В этом режиме промежутки закрашиваются цветом фона, определенным как атрибут контекста отображения. Приложение может установить прозрачный режим фона TRANSPARENT , в этом случае промежутки в линиях не будут закрашиваться .
Для установки режима фона предназначена функция SetBkMode :
int WINAPI SetBkMode(HDC hdc, int fnBkMode);
Эта функция устанавливает новый режим фона fnBkMode для контекста отображения hdc, возвращая в случае успеха код старого режима фона. Для параметра fnBkMode вы можете использовать значения OPAQUE или TRANSPARENT.
Приложение может определить текущий режим фона, вызвав функцию GetBkMode :
int WINAPI GetBkMode(HDC hdc);
С помощью функций SetBkColor и GetBkColor вы можете, соответственно, установить и определить текущий цвет фона, который используется для закраски промежутков между штрихами и точками линий:
COLORREF WINAPI SetBkColor(HDC hdc, COLORREF clrref);
COLORREF WINAPI GetBkColor(HDC hdc);

Выбор режима рисования

Для выбора режима рисования предназначена функция SetROP2 :
int WINAPI SetROP2(HDC hdc, int fnDrawMode);
Параметр hdc предназначен для указания контекста отображения, в котором необходимо установить новый режим рисования, определяемый параметром fnDrawMode.
Функция SetROP2 возвращает код предыдущего режима рисования.
Процесс рисования на экране монитора заключается в выполнении логической операции над цветами точек экрана и цветами изображения. Ниже в таблице мы привели возможные значения для параметра fnDrawMode. Для каждого режима рисования в этой таблице есть формула, с использованием которой вычисляется результат, и краткое описание режима рисования. В формулах цвет пера обозначается буквой P, цвет подложки - D.

Режим рисования	Формула	Цвет пиксела
R2_COPYPEN	P	Соответствует (равен) цвету пера
R2_BLACK	0	Черный
R2_WHITE	1	Белый
R2_NOP	D	Не меняется, т. е. перо ничего не рисует
R2_NOT	~D	Получается инвертированием цвета подложки, т. е. цвета пиксела до рисования
R2_NOTCOPYPEN	~P	Получается инвертированием цвета пера
R2_MASKPEN	P&D	Комбинация компонент цветов, имеющихся

		как в цвете подложки, так и в цвете пера
R2_NOTMASKPEN	~(P&D)	Инверсия предыдущего значения
R2_MERGEPEN	P|D	Комбинация компонент цветов подложки и пера
R2_NOTMERGEPEN	~(P|D)	Инверсия предыдущего значения
R2_XORPEN	P^D	При определении цвета пиксела выполняется операция "ИСКЛЮЧАЮЩЕЕ ИЛИ" между компонентами цвета подложки и пера
R2_NOTXORPEN	~(P^D)	Инверсия предыдущего значения
R2_MASKNOTPEN	~P & D	Комбинация цвета подложки и инверсии цвета пера
R2_MASKPENNOT	P & ~D	Комбинация двух цветов: инверсии цвета подложки и цвета пера
R2_MERGENOTPEN	~P | D	Комбинация компонент цветов подложки и инверсии цвета пера
R2_MERGEPENNOT	P | ~D	Комбинация инверсии цвета подложки и цвета пера

Если изображение и перо черно-белые, результат выполнения описанных выше операций (которые, кстати, называются растровыми операциями) можно легко предсказать.
В режиме R2_COPYPEN, который установлен в контексте отображения по умолчанию, цвет нарисованной линии будет такой же, как и цвет пера. Для режимов R2_BLACK и R2_WHITE цвет линии будет, соответственно, черный и белый. В режиме R2_NOP вы не увидите нарисованную линию, так как цвет вдоль нее вообще не изменится. Более интересен режим R2_NOT, при использовании которого на черном фоне будет нарисована белая линия, а на белом фоне - черная.
Для цветных изображений перечисленные выше формулы применяются по отдельности к каждой компоненте цвета (всего в Windows используется три компоненты цвета - красная, зеленая и голубая), поэтому для некоторых режимов рисования цвет линии предсказать достаточно трудно. Использование цветовых палитр, дополнительно усложняет эту задачу. С помощью функции GetROP2 приложение может определить режим рисования, установленный для контекста отображения hdc:
int WINAPI GetROP2(HDC hdc);

Рисование линий произвольного стиля

Как мы уже говорили, вы не можете создать перо для рисования пунктирных, штрих-пунктирных или штриховых линий толщиной больше одного пиксела. Однако в некоторых случаях у вас может возникнуть необходимость в рисовании таких линий.
В программном интерфейсе GDI есть функция с именем LineDDA, которая позволяет рисовать любые линии (правда, основная работа по рисованию линий при этом будет возложена на программиста).
Функция LineDDA имеет следующий прототип:
void WINAPI LineDDA(

```
  int nxStart, int nyStart,          // начальная точка
  int nxEnd, int nyEnd,              // конечная точка
  LINEDDAPROC lnddaprc,    // адрес функции для рисования
  LPARAM lParam);                    // дополнительные параметры
```

Первые четыре параметра этой функции определяют координаты начальной и конечной точки, между которыми надо нарисовать линию. Через параметр lnddaprc передается указатель на функцию рисования, которая является функцией обратного вызова, определяемой программистом. Эта функция получает управление много раз, она вызывается для каждой точки рисуемой линии.

Для режима STRICT тип LINEDDAPROC определен в файле windows.h следующим образом:

```
typedef void (CALLBACK* LINEDDAPROC)(int, int, LPARAM);
```

Последний параметр предназначен для передачи дополнительных данных в функцию рисования.

Приведем прототип функции рисования (для функции можно использовать любое имя):

```
void CALLBACK LineProc(int xPos, int yPos, LPARAM lParam);
```

Первые два параметра представляют собой координаты точки, для рисования которых вызвана функция. Последний параметр соответствует последнему параметру функции LineDDA и содержит передаваемое этой функции значение.

Рисование замкнутых фигур

Помимо линий, приложения Windows могут использовать функции GDI для рисования замкнутых закрашенных или незакрашенных фигур, таких как прямоугольники, эллипсы, многоугольники с прямыми и скругленными углами и т. д.

Для закрашивания внутренней области замкнутых фигур используется кисть, задаваемая как атрибут контекста отображения. Внешний контур фигуры обводится пером, которое также выбирается в контекст отображения. Учитываются и остальные атрибуты, установку которых мы рассмотрели для функций рисования линий, такие, как режим отображения, режим фона, код растровой операции.

Рисование прямоугольника

Простейшая функция, с помощью которой можно нарисовать прямоугольник, называется Rectangle :

```
BOOL WINAPI Rectangle(
  HDC hdc,   // идентификатор контекста отображения
  int nxTL,  // координата x верхнего левого угла
  int nyTL,  // координата y верхнего левого угла
  int nxBR,  // координата x правого нижнего угла
  int nyBR); // координата y правого нижнего угла
```

Функция Rectangle рисует прямоугольник для контекста отображения hdc, возвращая значение TRUE в случае успеха или FALSE при ошибке. Последние четыре параметра функции задают координаты верхнего левого и нижнего правого угла прямоугольника.

В зависимости от стиля пера граница фигуры может находится полностью внутри прямоугольника, заданного координатами (nxTL, nyTL), (nxBR,nyBR) или выходить

за его пределы . Если выбрать стиль пера PS_NULL, граница фигуры станет невидимой.

В зависимости от кисти, выбранной в контекст отображения, внутренность прямоугольника может быть закрашенной в тот или иной цвет, заштрихована одним из нескольких способов или закрашена с помощью любого битового изображения размером 8x8 пикселов. С помощью функции RoundRect можно нарисовать прямоугольник со

По сравнению с функцией Rectangle функция RoundRect имеет два дополнительных параметра nxEllipse и nyEllipse, определяющих форму и радиус закругления:

```
BOOL WINAPI RoundRect(
  HDC hdc,   // идентификатор контекста отображения
  int nxTL,  // координата x верхнего левого угла
  int nyTL,  // координата y верхнего левого угла
  int nxBR,  // координата x правого нижнего угла
  int nyBR,  // координата y правого нижнего угла
  int nxEllipse,  // ширина эллипса
  int nyEllipse); // высота эллипса
```

Функция FillRect закрашивает прямоугольную область окна заданной кистью:

```
 int WINAPI FillRect(
  HDC hdc,   // идентификатор контекста отображения
  const RECT FAR* lprc, // указатель на структуру RECT
  HBRUSH hbrush); // идентификатор кисти для закрашивания
```

Параметр lprc должен указывать на структуру типа RECT, в которую следует записать координаты закрашиваемой прямоугольной области. Правая и нижняя граница указанной области не закрашивается.

Независимо от того, какая кисть выбрана в контекст отображения, функция FillRect будет использовать для закраски кисть, указанную параметром hbrush.

Правильная работа функции FillRect гарантируется только в том случае, когда значение поля bottom структуры RECT больше значения поля top, а значение поля right больше значения поля left.

Для закрашивания границы прямоугольной области (т. е. для рисования прямоугольной рамки) можно использовать функцию FrameRect :

```
int WINAPI FrameRect(
  HDC hdc,   // идентификатор контекста отображения
  const RECT FAR* lprc, // указатель на структуру RECT
  HBRUSH hbrush); // идентификатор кисти для закрашивания
```

Параметры этой функции аналогичны параметрам функции FillRect.

Ширина пера, используемого для рисования рамки, всегда равна одной логической единице. Структура RECT должна быть подготовлена таким же образом, что и для функции FillRect, т. е. значение поля bottom структуры RECT должно быть больше значения поля top, а значение поля right - больше значения поля left.

Используя функцию InvertRect , вы можете инвертировать содержимое прямоугольной области, заданной параметром lprc:

```
void WINAPI InvertRect(HDC hdc, const RECT FAR* lprc);
```

Есть еще одна интересная функция, предназначенная для рисования прямоугольников. Она имеет имя DrawFocusRect :

```
void WINAPI DrawFocusRect(HDC hdc, const RECT FAR* lprc);
```
Эта функция рисует прямоугольную рамку, предназначенную для выделения окна, имеющего фокус ввода.
Функция DrawFocusRect имеет три интересные особенности.
Во-первых, для рисования используется растровая операция "ИСКЛЮЧАЮЩЕЕ ИЛИ". Это приводит к тому, что для удаления нарисованной таким образом рамки ее достаточно нарисовать еще раз на том же месте.
Вторая особенность заключается в том, что для использования этой функции не нужно выбирать перо, рисующее пунктирную линию. Функция DrawFocusRect рисует пунктирную линию с нестандартным, очень близким расположением точек.
Третья особенность заключается в том, что перед использованием этой функции необходимо установить режим отображения MM_TEXT.
Первые две особенности позволяют использовать ее для рисования рамки выделения произвольных участков изображения на экране монитора (при помощи мыши).
В заключение отметим, что в программном интерфейсе Windows нет функции для рисования квадрата и круга. Эти фигуры являются частными случаями, соответственно, прямоугольника и эллипса, поэтому для рисования, например, квадрата, вы должны использовать одну из только что описанных функций. Для сохранения пропорций проще всего использовать одну из метрических систем координат.

Рисование эллипса

Для рисования эллипса вы можете использовать функцию Ellipse :
```
BOOL WINAPI Ellipse(
  HDC hdc,   // идентификатор контекста отображения
  int nxTL,  // координата x верхнего левого угла
  int nyTL,  // координата y верхнего левого угла
  int nxBR,  // координата x правого нижнего угла
  int nyBR); // координата y правого нижнего угла
```
Первый параметр этой функции указывает идентификатор контекста отображения, остальные - координаты верхнего левого и правого нижнего углов прямоугольника, в который должен быть вписан эллипс .

Рисование сегмента эллипса

Сегмент эллипса можно нарисовать при помощи функции Chord :
```
BOOL WINAPI Chord(
  HDC hdc,                    // идентификатор контекста отображения
  int nxLeft,  int nyTop,     // верхий левый угол
  int nxRight, int nyBottom, // правый нижний угол
  int nxStart, int nyStart,     // начало дуги
  int nxEnd,   int nyEnd);    // конец дуги
```
Параметры этой функции аналогичны параметрам рассмотренной нами ранее функции Arc.

Рисование сектора эллипса

Для рисования сектора эллипса (рис. 2.21) следует использовать функцию Pie , аналогичную по своим параметрам функциям Arc и Chord:
```
BOOL WINAPI Pie(
```

```
  HDC hdc,                    // идентификатор контекста отображения
  int nxLeft,  int nyTop,     // верхний левый угол
  int nxRight, int nyBottom, // правый нижний угол
  int nxStart, int nyStart,   // начало дуги
  int nxEnd,   int nyEnd);    // конец дуги
```

Рисование многоугольников

Рисование многоугольников выполняется функцией Polygon , аналогичной по своим параметрам функции Polyline, с помощью которой рисуются ломаные линии:

```
BOOL WINAPI Polygon(
  HDC hdc,                    // идентификатор контекста отображения
  const POINT FAR* lppt,// указатель на массив структур POINT
  int cPoints);               // размер массива
```

Через параметр hdc передается идентификатор контекста отображения.
Параметр lppt указывает на массив структур POINT, в котором должны находится координаты вершин многоугольника. Параметр cPoints определяет размер этого массива.
Функция Polygon возвращает TRUE при нормальном завершении или FALSE при ошибке. Она не использует текущую позицию пера и не изменяет ее.
В массиве структур POINT, определяющих вершины многоугольника, каждая вершина должна быть указана один раз. Функция Polygon автоматически замыкает ломаную линию, образующую многоугольник.
С помощью функции PolyPolygon можно нарисовать одновременно несколько многоугольников:

```
BOOL WINAPI PolyPolygon(
  HDC hdc,                    // идентификатор контекста отображения
  const POINT FAR*lppt, // указатель на массив структур POINT
  int FAR* lpnPolyCounts, // адрес массива количества точек в многоугольниках
  int cPolygons);             // количество многоугольников
```

Параметр cPolygons определяет количество многоугольников, которые нужно нарисовать. Параметр lppt должен содержать указатель на массив структур типа POINT, содержащий координаты вершин всех многоугольников. Через параметр lpnPolyCounts передается указатель на массив целых чисел. Каждое число в этом массиве определяет количество точек в соответствующем многоугольнике.
В отличие от функции Polygon, функция PolyPolygon не замыкает автоматически ломаную линию, образующую многоугольник.
В контексте отображения имеется атрибут, влияющий на способ закрашивания для самопересекающихся многоугольников. По умолчанию выбран режим ALTERNATE , в котором эти области не закрашиваются (закрашиваются только те области, которые расположены между нечетными и четными сторонами многоугольника).
С помощью функции SetPolyFillMode вы можете изменить значение этого атрибута на WINDING . В этом режиме для того чтобы определить, надо ли закрашивать область многоугольника, учитывается направление, в котором был нарисован этот многоугольник. Каждая сторона многоугольника может быть нарисована в направлении либо по часовой стрелке, либо против часовой стрелки. Если воображаемая линия, нарисованная в направлении из внутренней области многоугольника в

наружную, пересекает сегмент, нарисованный в направлении по часовой стрелке, содержимое некоторого внутреннего счетчика увеличивается на единицу. Если же эта линия пересекает сегмент, нарисованный против часовой стрелки, содержимое счетчика уменьшается на единицу. Область закрашивается только в том случае, если содержимое счетчика не равно нулю.
Прототип функции SetPolyFillMode: int WINAPI SetPolyFillMode(HDC hdc, int fnMode);
Параметр fnMode, определяющий режим закрашивания многоугольников, может принимать значения ALTERNATE или WINDING. Функция возвращает код старого режима закрашивания.
Вы можете определить используемый в данный момент режим закрашивания многоугольников с помощью функции GetPolyFillMode :
int WINAPI GetPolyFillMode(HDC hdc);

Выбор кисти

Для закрашивания внутренней области замкнутых фигур вы можете использовать встроенные кисти, или кисти, созданные вашим приложением. Последние необходимо удалять после использования.

Использование встроенной кисти

Для выбора одной из встроенной кисти вы можете воспользоваться макрокомандой GetStockBrush .В качестве параметра для этой макрокоманды можно использовать следующие значения:

Значение	Описание
BLACK_BRUSH	Кисть черного цвета
WHITE_BRUSH	Кисть белого цвета
GRAY_BRUSH	Серая кисть
LTGRAY_BRUSH	Светло-серая кисть
DKGRAY_BRUSH	Темно-серая кисть
NULL_BRUSH	Бесцветная кисть, которая ничего не закрашивает
HOLLOW_BRUSH	Синоним для NULL_BRUSH

Как видно из только что приведенной таблицы, в Windows есть только монохромные встроенные кисти. Макрокоманда GetStockBrush возвращает идентификатор встроенной кисти. Прежде чем использовать полученную таким образом кисть, ее надо выбрать в контекст отображения (так же, как и перо). Для этого проще всего воспользоваться макрокомандой SelectBrush .

Создание кисти

Если вам нужна цветная кисть, ее следует создать с помощью функции CreateSolidBrush :HBRUSH WINAPI CreateSolidBrush(COLORREF clrref);
В качестве параметра для этой функции необходимо указать цвет кисти. Для выбора цвета вы можете воспользоваться, например, макрокомандой RGB, позволяющей указать содержание отдельных цветовых компонент.
Windows может выбрать для кисти чистые или смешанные цвета, что зависит от текущего цветового разрешения.

После использования созданной вами кисти ее следует удалить, не забыв перед этим выбрать в контекст отображения старую кисть. Для удаления кисти следует использовать макрокоманду DeleteBrush .
Можно заштриховать внутреннюю область замкнутой фигуры, создав одну из шести кистей штриховки функцией CreateHatchBrush :
HBRUSH WINAPI CreateHatchBrush(int fnStyle, COLORREF clrref);
С помощью параметра clrref вы можете определить цвет линий штриховки.
Параметр fnStyle задает стиль штриховки:
Вы можете использовать свой собственный стиль штриховки, создав кисть из битового изображения размером 8x8 пикселов (можно использовать только такой размер).
Если битовое изображение кисти определено в ресурсах приложения, его следует загрузить при помощи функции LoadBitmap . Эта функция возвратит идентификатор битового изображения. Затем для создания кисти этот идентификатор следует передать в качестве параметра функции CreatePatternBrush :
HBRUSH WINAPI CreatePatternBrush(HBITMAP hBitmap);
Битовые изображения делятся на те, которые хранятся в формате, зависящем от аппаратных особенностей устройства отображения, и на те, которые хранятся в аппаратно-независимом формате. Последние более универсальны, однако труднее в использовании. С помощью функции CreateDIBPatternBrush вы можете использовать для кисти битовое изображение в аппаратно-независимом формате:
HBRUSH WINAPI CreateDIBPatternBrush(HGLOBAL hglbDibPacked, UINT fnColorSpec);
Первый параметр указывает на область глобальной памяти, в которой содержится аппаратно-независимое битовое изображение в упакованном формате. Второй параметр определяет содержимое таблицы цветов, используемое этим битовым изображением, и может принимать два значения: DIB_PAL_COLORS (таблица цветов содержит ссылки на цветовую палитру) DIB_RGB_COLORS (таблица цветов содержит отдельные компоненты цвета).

Закрашивание внутренней области окна

Напомним, что кисть можно использовать еще и для закрашивания внутренней области окна . Для этого идентификатор кисти следует записать в поле hbrBackground структуры типа WNDCLASS перед регистрацией класса окна:
wc.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);

Установка начальных координат кисти

Начальные координаты кисти (brush origin) - это атрибут контекста отображения. Он используются для определения координат точки внутри кисти, которая будет служить начальной при закраске внутренней области фигуры или окна. По умолчанию используются координаты (0,0), соответствующие верхнему левому углу кисти (в системе координат, выбранной в контекст отображения по умолчанию).
Если кисть используется для закраски внутренней области окна, верхний левый угол изображения кисти совмещается с верхним левым углом этой области. Затем изображение кисти многократно повторяется с шагом 8 пикселов.

При закраске фигур начальное расположение кисти привязывается не к фигуре, а по-прежнему к верхнему левому углу внутренней области окна. Поэтому при закраске, например, прямоугольника, верхний левый угол кисти может не совпадать с верхним левым углом прямоугольника. Приложение может изменить начальные координаты кисти (сдвинуть кисть) при помощи функций SetBrushOrg и UnrealizeObject. Прежде всего нужно вызвать функцию UnrealizeObject , передав ей в качестве параметра идентификатор сдвигаемой кисти (только если это не встроенная кисть):
BOOL WINAPI UnrealizeObject(HGDIOBJ hbrush);
В этом случае система сбросит координаты кисти после выбора ее в контекст отображения. После сброса надо установить новые значения координат кисти, вызвав функцию SetBrushOrg : DWORD WINAPI SetBrushOrg(HDC hdc, int nx, int ny);
Параметры nx и ny определяют новые значения для начальных координат кисти пикселах (от 0 до 6). В завершении следует снова выбрать кисть в контекст отображения при помощи макрокоманды SelectBrush.

Цвет и цветовые палитры

Цветовые возможности приложений ограничиваются в основном цветовым разрешением аппаратуры. Хорошие видеоконтроллеры SVGA, способные работать в режиме True Color , могут отображать 16666216 цветов.
Для использования режимов с высоким разрешением нужны драйверы, которые обычно поставляются вместе с видеоконтроллером.
В операционной системе Windows приложения определяют цвет , задавая интенсивности трех его RGB- компонент : красный (R), зеленый (G), голубой (B).Интенсивность каждой компоненты задается числом в диапазоне от 0 (минимальная интенсивность) до 255 (максимальная интенсивность). Такая система позволяет приложению указать любо из 16666216 цветов (256x256x256=16666216).

Раскрашивая изображения , приложения Windows может использовать любые цвета. Однако это не означает , что цвет изображения, полученного на экране (или принтере) будет в точности такой, какой был указан при выводе. Windows учитывает цветовое разрешение устройств вывода, ограничивая соответствующим образом цветовую гамму изображения или работая со смешенными цветами (смешенный цвет образуется из чистых цветов, при этом изображение состоит из точек, имеющих чистые цвета). Соответствующий механизм достаточно сложен и зависит от текущего цветового разрешения. В режиме с низким цветовым разрешением используется 16 различных цветов (режим совместимости с VGA). Определяя цвет пера для рисования линий, вы можете указать любой цвет, однако в результате будет выбран один из 16 цветов, максимально соответствующий заказанному. Цвет кисти может быть либо чистым (в этом случае используется 1 из 16 цветов), либо смешанным .

В режиме низкого цветового разрешения может использоваться механизм цветовых палитр. Приложению доступны сотни тысяч цветов , составляющих палитру. Механизм палитры не является прозрачным для приложений и сложен в использовании .

В режиме True Color палитры не используются, а полученный на экране цвет полностью соответствует заказанному. Несмотря на то что стоимость видеоконтроллеров True Color постоянно снижается они все же дороже обычных

видеоконтроллеров SVGA (есть еще одна проблемма , связанная с увеличением объема памяти, необходимого для хранения битовых изображений с высоким цветовым разрешением).

Поэтому когда пользователю не нужна высокая скорость работы в Windows но нужна возможность работы с разрешением 256 цветов, он может остановить свой выбор на более дешевом видео контроллере. Кроме того, даже если в компъютере установлен видеоадаптер True Color , он может использоваться в режимах со средним или низким разрешением. Для программиста это означает необходимость использования цветовых палитр , так как только в этом случае будут реализованы все цветовые возможности такого видеоконтроллера.

Системная цветовая палитра

Цветовая палитра- это не более чем набор цветов.

В GDI встроенны средства для работы с 256 - цветными палитрами. Если видеоконтроллер способен работать с палитрами , создается одна системная палитра, которая содержит отображаемые на экране цвета. Вы можете думать об этой палитре как о таблице цветов , хранящейся в памяти видеоконтроллера.

Часть системной палитры (20 элементов) зарезервированны для использования операционной системы. В зарезервированных элементах хранятся статические цвета, которые нужны для рисования таких объектов, как рамки окон , полосы просмотра и т.п., а также изображений , рисуемых в приложении. Если видеоконтроллер рабо-
тает в режиме низкого цветового разрешения или приложение не использует цветовые палитры (несмотря на наличие соответствующих возможностей аппаратуры), цветовая гамма приложения ограничена статическими цветами.

Приложения никогда не изменяют статические цвета , записанные в зарезервированных ячейках системной палитры. Содержимое остальных 236 ячеек системной палитры может изменяться в процессе реализации приложениями своих собственных цветовых палитр.

Расположение статических цветов в системной палитре -10 цветов вначале, 10-вконце выбранны для обеспечения правильной работы часто используемой растровой операции ИСКЛЮЧАЮЩЕЕ ИЛИ .

Выбор цвета без использования палитры

Приложения, которые не хотят ничего знать про палитры , могут указывать логический цвет изображений , составляя его из RGB - компонент , указывая их количественный состав. Однако если видеоконтроллер не работает в режиме True Color, для вывода на экран будут использованы только статические цвета или смешанные цвета , состоящие из статических цветов. В результате полученный на экране физический цвет может не соответствовать запрошенному логическому цвету.

Как мы уже говорили , в зависимости от текущего цветового разрешения Windows может предоставить приложению приближенный цвет, который максимально соответствует запрошенному логическому цвету. Функция GetNearestColor возвращает для запрошенного логического цвета clrref физический цвет , составленный только из компонент чистого цвета:

COLORREF WINAPI GetNearestColor (HDC hdc, COLORREF clrref);

Системные цвета

Относитедьно удачный способ задания цветов заключается в использовании так назы-
ваемых системных цветов.Системные цвета-это цвета, с помощью которых перационная система Windows рисует отдельные элементы окон и органов управления.

Приложение Control Panel, которое входит в состав Windows, позволяет вам изменять системные цвета, обеспечивая приемлемую цветовую палитру практически для любого типа видеомонитора.

Приложение может выбрать для использования некоторые из системных цветов, при этом пользователь сможет влиять на внешний вид вашего приложения с помощью Control Panel, настраивая цвета на свой вкус.

Сообщение WM_SYSCOLORCHANGE

посылается всем активным окнам верхнего уровня при изменении системных цветов. В ответ на это сообщение приложения , которое создают свои перья и кисти на базе системных цветов, должны удалить эти перья и кисти, а затем создать их заново.

Так как после изменения системных цветов все активные окна получают сообщение WM_PAINT, обработчик сообщения WM_SYSCOLORCHANGE не должен ничего перерисовывать в окне .

Функция ChooseColor

В составе DLL - библиотеки commdlg.dll есть функция ChooseColor, которая предназначена для выбора цвета. Эта функция выводит на экран диалоговую панель, с помощью которой пользователь может выбрать цвет из основного набора цветов "Basic Color" и дополнительного "Custom Colors". Возможность использования цветовых палитр при выборе не обеспечивается, так что с помощью этой функции можно выбирать только чистые статические или смешанные цвета.

Если нажать на кнопку "Define Custom Colors...", внешний вид диалоговой панели изменится. Пользуясь правой половиной диалоговой панели, пользователь сможет добавить новый цвет в набор "Custom Colors"и затем выбрать его из этого набора. Функция ChooseColor описана в файле commdlg.h:

BOOL WINAPI ChooseColor (CHOOSECOLOR FAR* lpcc);

Перед вызовом функции следует заполнить структуру CHOOSECOLOR, передав функции ее адрес через параметр lpcc. В случае успешного выбора цвета функция возвращает TRUE. Если пользователь отказался от выбора или если произошла ошибка, возвращается значение FALSE.

Структура CHOOSECOLOR и указатель на неее описанный в файле commdlg.h :

```
typedef struct tagCHOOSECOLOR
{
  DWORD lStructSize;
  HWND hwndOwner;
  HWND hInstance;
  COLORREF rgbResult;
  COLORREF FAR*lpCustColors;
  DWORD Flagsp;
```

```
  LPARAM lCustData;
  UINT (CALLBACK* lpfnHook)(HWND, UINT, WPARAM, LPARAM);
  LPCSTR lpTemplateName;
} CHOOSECOLOR;
typedef CHOOSECOLOR FAR*LPCCHOOSECOLOR;
```

Назначение и правила использования отдельных полей этой структуры.

Поле lStructSize заполняется перед вызовом функции. Оно должно содержать размер структуры в байтах. Поле Flags также заполняется до вызова функции.В него следует записать флаги инициализации влияющие на использование других полей этой структуры.

Флаг	Описание
CC_RGBINIT	Для цвета,выбранного по умолчанию, используетсяцвет, указанный в поле rgbResult
CC_FULLOPEN	Если указан этот флаг, на экране появляется полный вариант диалоговой панели, обеспечивающий возможность определения произвольного цвета.Если этот флаг не указан, для определения произвольного цвета пользователь должен нажать на кнопку "Define Custom Color"
CC_PREVENTFULLOPEN	Кнопка "Define Custom Color" блокируется, таким образом, при использовании этого флага пользователь не может определить произвольный цвет
CC_SHOWHELP	Флаг разрешает отображение кнопки "Help". Если указан этот флаг, в поле нельзя указывать значение hwndOwner
CC_ENABLEHOOK	Если указан этот флаг, используется функция фильтра, адрес которой указан в поле lpfmHook.С помощью этой функции можно организовать дополнительнуюобработку сообщений от диалоговой панели
CC_ENABLETEMPLATE	Используется шаблон диалоговой панели, пределяемый содержимым поля hInstance. Адрес строки, содержащей имя шаблона, должен быть указан в поле lpTemplateName
CC_ENABLETEMPLATEHANDLE	Используется шаблон диалоговой панели, идентификатор которого записан в поле hInstance. Содержимое поля lpTemplat-eName игнорируется

В поле hwndOwner перед вызовом функции следует записать идентификатор окна, создавшего диалоговую панель или NULL. В последнем случае нельзя использовать флаг CC_SHOWHELP. Поле hInstance заполняется в тех случаях, когда приложение
использует свой собственный шаблон диалоговой панели (вместо стандартного шаблона, расположенного в ресурсах DLL-библиотеки).
В этом случае перед вызовом функции в это поле следует записать идентификатор модуля, содержащего шаблон диалоговой панели. В поле Flegs необходимо указать флаг CC_ENABLETEMPLATE или CC_ENABLETEMPLATEHANDLE .

В поле lpTemplateName следует записать адрес текстовой строки идентификатора ресурсов шаблона диалоговой панели (если этот шаблон используется). Поле rgbResult предназначено для передачи приложению цвета, выбранного пользователю.Если записать в это поле значение NUUL, сразу после вывода диалоговой панели выбора цвета по умолчанию будет выбран черый цвет. Вы, однако, можете использовать для начального выбора любой другой цвет, записав его в это поле и указав флаг CC_RGBINIT.

Перед вызовом функции вы должны подготовить массив из 16-ти двойных слов, содержащих цвета для использования в меню "Custom Colors".Адрес этого массива следует передавать через параметр lpCustColors.

Поле lCustData используется для передачи данных функции фильтра (если она определена). Адрес функции фильтра передается через параметр lpfnHook. Для использования функции фильтра следует указать флаг CC_ENABLEHOOK.

Использование цветовых палитр

В большинстве случаев вам не потребуется использовать палитры Windows так как не все приложения работают с большим количеством цветов.Приложения для обычной презинтационной графики , обработки текстовыхдокументов и другие аналогичные приложения выглядят вполне удовлетворительно при использовании 16 цветов , обеспечиваемых стандартным драйвером VGA. Более того , увеличение цветового разрешения , как правило , сказывается отрицательно на производительности Windows. И это понятно-чем больше используется цветов , тем больше обьем данных передаваемых из процессора в видеоконтроллер.Несмотря на то что акселераторы Windows значительно ускоряют работу , очень много пользователей имеют дешевые видеоконтроллеры , работающие в режиме незкого или ,в крайнем случае , среднего цветового разрешения.

Если же ваше приложение должно работать с многокрасочными реалистическими изображениями или выводить на экран битовые изображения , полученные с помощью цветного сканера , задача использования цветовых палитр выдвигается на первый план .

Приложение должно использовать все цветовые возможности обычных видеоадаптеров SVGA, которые реализуются с помощью механизма цветовых палитр.

Механизм реализации логической палитры

В системной палитре , состоящей из 256 ячеек , 20 ячеек зарезервированы для статических цветов. Остальные 236 ячеек доступны для приложений. Что же приложения могут с ними сделать ?

Любое приложение может создать свою собственную палитру цветов в виде массива размером до 256 елементов, содержащие данные типа PALETTEENTRY , которые могут хранить RGB-цвета или номера цветов в системной палитре.

Подготовив массив, содержащий цвета, приложение может создать логическую палитру, вызвав функцию **CreatePalette**. Затем палитру нужно ввыбрать в контекст отображения функцией **SelectPalette** так как на экране могут отображаться только цвета, находящиеся в системной палитре. В процессе реализации палитры приложение должно перенести (или отобразить)цвета из логической палитры в системную палитру, вызвав функцию RealisePalette. Последний шаг называется реали-

зацией палитры. В зависимости от того, как подготовленна логическая палитра, а так же от того является приложение активным или фоновым, механизм реализации может выполняться по-разному.
Рассмотрим обычный способ реализации.
Первоначально в системной палитре 20 ячеек отмеченны как занятые для статических цветов и 236- как свободные. Когда первое приложение реализует свою логическую палитру, цвета из этой палитры переписываются в свободные ячейки системной палитры, после чего они становятся доступными для отображения. Если свободных
ячеек не хватает для размещения всей логической палитры, оставшиеся цвета будут отображены на ближайшие имеющиеся в системной палитре. При этом неизбежны цветовые искажения.

Однако в Windows одновременно может работать несколько приложений, одно из которых является активным , а остальные - фоновые. Для активных и фоновых приложений используется разный механизм реализации логической палитры.

Активное приложение имеет большой приоритет использования свободных ячеек системной палитры.Как правило, запрос активного приложения (точнее активного окна)на реализацию логической палитры выполняется полностью, так как перед реализацией все ячейки системной палитры (кроме 20 зарезервированных) отмечаются как
свободные .

Фоновые приложения довольствуются теми свободными ячейками, что остались после риализации логической палитрой активного приложения. Поэтому для фоновых приложений качества "цветопередачи"может быть не очень высоким.

Часть цветов логической палитры фонового окна уже есть в системной палитре, поэтому они просто отображаются на соответствующие ячейки системной палитры. Три свободные ячейки используются для тех цветов, которых нет в системной палитре.

Однако в нашей логической палитре есть цвета , которым нет точного соответствия в системной палитре, причем свободных ячеек тоже не осталось. В этом случае GDI отображает эти цвета на близкие из системной палитры.После такого упращенного описания процесса реализации логической палитры перейдем к конкретным шагам,
необходимым для использования палитры.

Проверка возможности использования палитры

Далеко не все устройства отображения способны работать с палитрой.Например, драйвер видеоадаптера VGA палитры не поддерживает, несмотря на то что аппаратура VGA может работать с палитрой. Последнее обстоятельство связано с тем, что размер палитры VGA составляет всего 64 ячейки,что явно не достаточно.

Самый лучший способ убедиться в том, что драйвер видео адаптера способен работать с палитрами, заключается в вызове функции GetDeviceCaps с параметром RASTERCAPS.Если в возвращаемом слове установлен бит RC_PALETTE, приложение может использовать цветовые палитры .

Стандартный размер системной палитры равен 256 ячейкам, однако можно уточнить это значение, вызвав функцию GetDeviceCaps с параметром SIZEPALETTE. Если драйвер видеоконтроллера не работает с палитрами, размер

палитры, определенный с помощью функции GetDeviceCaps , может быть равным нулю.
Для определения количества системных цветов используйте эту же функцию с параметром NUMRESERVED. С помощью функции GetDeviceCaps можно также определить цветовое разрешение устройства вывода.При этом следует учитывать , что некоторые устройства работают с цветовыми плоскостями. Количество этих плоскостей можно определить , пользуясь значением PLANES. Отметим , что количество цветов , которые могут быть представлены устройством с цветовыми плоскостями, равно 2^n, где n-количество цветовых плоскостей.

Если устройство работает с цветовыми плоскостями и использует несколько бит для представления цвета одного пиксела, количество одновременно отображаемых цветов можно определить по формуле : nColors=2*(nPixel*nPlanes)
Где nPixel-количество бит , используемых для представления цвета пиксела (значение BITSPIXEL), nPlanes-количество цветовых плоскостей (значение PLANES).

Значение NUMCOLORS равно количеству статических цветов.Так как палитра может быть изменена (перезагружена) фактически вы можете использовать больше цветов чем указано в NUMCOLORS. Но в этом случае вы должны сами перезагружать палитру. Для устройств , работающих с палитрами , правильное количество используемых цветов возвращается при использовании значения COLORRES - количество бит цветового разрешения. Для режима с использованием 256 цветов это значение равно 18, что соответствует 6 битам для представления каждого из трех базовых цветов.

Создание логической палитры

Для того чтобы создать палитру , ваше приложение должно заполнить структуру LOGPALETTE, описывающую палитру,и массив структур PALETTEENTRY, определяющий содержимое палитры.

```
typedef struct tagLOGPALETTE
{
  WORD palVersion;
  WORD palNumEntries;
  PALETTEENTRY palPalEntry[1];
} LOGPALETTE;
typedef LOGPALETTE*PLOGPALETTE;
typedef LOGPALETTE NEAR*NPLOGPALETTE;
typedef LOGPALETTE FAR*LPLOGPALETTE;
```

Поле palVersion должно содержать значение 0x300.

В поле palNumEntries нужно записать размер палитры (количество элементов в массиве структур PALETTEENTRY). Сразу после структуры LOGPALETTE в памяти должен следовать массив структур PALETTEENTRY , описывающий содержимое палитры:

```
typedef struct tagPALETTEENTRY
{
  BYTE peRed;
  BYTE peGreen;
  BYTE peBlue;
  BYTE peFlags;
```

} PALETTEENTRY;
typedef PALETTEENTRY FAR* LPPALETTEENTRY;

Поле peFlags определяет тип елемента палитры и может иметь значения NULL,PC_EXPLICIT,PC_NOCOLLAPSE,PC_RESERVED. Если поле peFlags содержит значение NULL в полях peRed, peGreen, peBlue находятся RGB- компоненты цвета. В процессе реализации логической палитры для этого элемента используется описанный нами ранее алгоритм.

Если поле peFlags содержит значение PC_EXPLICIT , младшее слово элемента палитры содержит индекс цвета в системной палитре.

Если поле peFlags содержит значение PC_NOCOLLAPSE, в процессе реализации логической палитры данный элемент будет отображаться только на свободную ячейку системной палитры. Если же свободных ячеек нет, используется обычный алгоритм реализации.

Если поле peFlags содержит значение PC_RESERVED оно используется для анимации палитры с помощью функции AnimatePalette.
Анимация палитры позволяет динамически вноситm изменения в палитру.Такой элемент палитры после реализации не подвергается изменениям при реализации других палитр, он становится зарезервированным.

После подготовки структуры LOGPALETTE и массива структур PELETTEENTRY приложение может создать логическую палитру , вызвав функцию CreatePalette:
HPALETTE WINAPI CreatePalette(const LOGPALETTE FAR* lplgpl) ;

В качестве параметра следует передать функции указатель на заполненную структуру LOGPALETTE. Функция возвращает идентификатор созданной палитры или NULL при ошибке.

Выбор палитры в контекст отображения

Созданная палитра перед использованием должна быть выбрана в контекст отображения. Выбор палитры выполняется функцией SelectPalette:
HPALATTE WINAPI SelectPalette(HDC hdc, HPALETTE hpal, BOOL fPalBack);

Функция выбирает палитру hpal в контекст отображения hdc возвращая в случае успеха идентификатор палитры , которая была выбрана в контекст отображения раньше. При ошибке возвращается значение NULL.

Указав для параметра fPalBack значение TRUE вы можете заставить GDI в процессе реализации палитры использовать алгоритм соответствующий фоновому окну. Если же этот параметр равен FALSE , используется алгоритм для активного окна (то есть все ячейки системной палитры , кроме зарезервиролванных для статических цветов , отмечаются как свободные и используются для реализации палитры).

Реализация палитры

Процедура реализации палитры заключается в вызове функции RealizePalette:

UINT WINAPI RealizePalette(HDC hdc);

Возвращаемое значение равно количеству цветов логической палитры , которое удалось отобразить в системную палитру.

Рисование с использованием палитры

Если приложению нухно создать перо или кисть , определить цвет текста функцией SetTextColor или закрасить его область функцией FloofFill (то есть вызвать одну из функций , которой в качестве параметра передается переменная типа COLORREF, содержащая цвет), вы можете вместо макрокоманды RGB воспользоватся одной из
следующих макрокоманд :

```
#define PALETTEINDEX(i) \
   ((COLORREF)(0x01000000L | (DWORD)(WORD)(i)))
#define PALETTERGB (r,g,b) (0x02000000L | RGB(r,g,b))
```

Макрокоманда PALETTEINDEX позволяет указать вместо отдельных компонент цвета индекс в логической палитре , соответствующий нужному цвету.

Макрокоманда PALETTERGB имеет параметры, аналогичные знакомой вам макрокоманде RGB, однако работает по-другому. Если цвет определен с помощью макрокоманды RGB , в режиме низкого и среднего цветового разрешения для рисования будет использован ближайший к указанному статический цвет. В режиме высокого цветового разрешения полученный цвет будет полностью соответствовать запрошенному.

Если же для определения цвета использована макрокоманда PALETTERGB , GDI просмотрит системные палитры и подберет из нее цвет, наиболее лучшим образом соответствующий указанному в параметрах макрокоманды.

В любом случае при работе с палитрой GDI не использует для удовлетворения запроса из логической палитры смешанные цвета.

Какой из двух макрокоманд лучше пользоваться?

Макрокоманда PALETTEINDEX работает быстрее, однако с ее помощью можно использовать только те цвета, которые есть в системной палитре. Если ваше приложение будет работать в режиме True Color , лучшего эффекта можно добиться при использовании макрокоманды PALETTERGB, так как для режимов высокого цветового разрешения эта макрокоманда обеспечит более точное цветовое соответствие.

Удаление палитр

Так как палитра является объектом , принадлежащим GDI, а не создавшему ее приложению, после использования палитры приложение должно обязательно ее удалить. Для удаления логической палитры лучше всего воспользоваться макрокомандой DeletePalette ,
В качестве параметра этой макрокоманде следует передать идентификатор удаляемой палитры. Rак и любой другой объект GDI , нельзя удалять палитру, выбранную в контекст отображения. Перед удалением следует выбрать старую палитру, вызвав функцию SelectPalette.

Сообщение об изменении палитры

Если ваше приложение активно работает с палитрами оно должно иметь в виду, что одновременно вместе с ним могут работать и другие приложения, претендующие на изменение системной палитры. Для того чтобы при передаче фокуса ввода другому приложению изображение, построенное с использованием палитры, не оказалось

испорчено в результате изменения системной палитры другим приложением, ваше приложение должно обрабатывать сообщение WM_QUERYNEWPALETTE и WM_PALETTECHANGED .

Сообщение WM_QUERYNEWPALETTE

посылается окну, которое становится активным.Это сообщение не имеет параметров. Главное окно приложения может менять свой статус с активного на фоновое несколько раз.Каждый раз, когда оно становится активным ему посылается сообщение WM_QUERYNEWPALETTE. В ответ на это сообщение приложение должно заново реализовать свою логическую палитру , так как , пока его главное окно было не активно другое приложение могло изменить системную палитру. Если палитра изменилась обработчик сообщения WM_QUERYNEWPALETTE должен перерисовать окно.

Если обработчик сообщения WM_QUERYNEWPALETTE изменил системную палитру, он должен вернуть ненулевое значение, а если нет - нулевое.

Сообщение WM_PALETTECHANGED

Когда любое приложение изменяет системную палитру, все перекрывающиеся (overlapped) и временные (pop up) окна получают сообщение WM_PALETTECHANGED. Это сообщение посылается так же в окно приложения, которое выполнило изменение системной палитры.

Параметр wParam сообщения WM_PALETTECHANGED содержит идентификатор окна, изменившего системную палитру. Если приложение обрабатывает это сообщение, оно должно вернуть нулевое значение.

В ответ на сообщение WM_PALETTECHANGED приложение должно заново реализовать палитру и если палитра изменилась перерисовать окно. Вместо полной перерисовки окна можно обновить цвета в окне, вызывая функцию UpdateColors:

```
int WINAPI UpdateColors (HDC hdc);
```

Следует , однако, иметь ввиду, что обновление цветов может привести к деградации качества изображения , поэтому при изменении палитры лучше перерисовать окно заново.

Битовые изображения

Работа с битовыми изображениями, особенно в аппаратном независимом формате , не самое простое занятие.Программист должен учитывать многочисленные нюансы, уметь работать с цветовыми палитрами и хорошо ориентироваться в контексте отображения.

В операционной системе Windows используются два формата битовых изображений - аппаратно зависимый DDB (device-dependent bitmap)и аппаратно независимый DIB(device-independent bitmap).

Согласно определению, данному в документации к SDK, битовое изображение DDB есть набор бит в оперативной памяти, который может быть отображен на устройстве вывода(например , распечатан на принтере). Внутренняя структура изображения DDB жестко привязанна аппаратным особенностям устройства вывода.Поэтому представление изображения DDB в оперативной памяти полностью зависит от устройства вывода.

Иногда битовые изображения называют растровыми , подчеркивая тот факт , что его можно раматривать как совокупность строк растра (горизонтальных линий развертки).

Если бы в Windows можно было работать только с изображениями DDB было бы необходимо иметь отдельные наборы изображений для каждого типа видеоконтроллера и каждого видеорежима, что крайне неудобно.

Аппаратно-независимая битовое изображение DIB содержит описание цвета пикселов изображения, которое не зависит от особенностей устройства отображения.Операционная система Windows после соответствующего преобразования может отобразить такое изображение на любом устройстве вывода.Несмотря на некоторое замедление процесса вывода по сравнению с выводом изображений DDB , универсальность изображений DIB делает их весьма привлекательными для хранения изображений.

Вместе с Windows версии поставляется несколько файлов, имеющих расширение имени bmp. Эти файлы содержат битовые изображения в формате DIB. Создавая приложение, которое должно работать с bmp - файлами следует учитывать, что существуют несколько форматов таких файлов.Файлы битовых изображений могут содержать таблицу цветов или цветовую палитру, могут быть сжаты с использованием алгоритмов RLE4 или RLE8. Кроме того коммерческая версия приложения должна уметь работать с битовыми изображениями в формате оболочки Presentation Manager операционной системы SO/2. Былобы также не плохо если бы приложение могло анализировать структуру bmp-файла с целью обнаружения возможных ошибок , так как отображение содержимого неправильного файла может создать полный хаос на экране.

Кроме битовых изображений используются так называемые векторные изображения. Если битовые изображения содержат описание цвета каждого пиксела , векторные изображения состоят из описаний отдельных графических обьектов , из которых состоит изображение. Это могут быть линии , окружности и т. п. Некоторые графические редакторы , например Corel Draw, Microsoft Draw, Micrografx Designer , для вашего представления изображения используют векторный формат .

Сравнивая эти форматы , отметим что каждый из них имеет свои преимущества и свои недостатки. Битовые изображения , как правило , выводятся на экран быстрее , так как их внутренняя структура аналогична (до некоторой степени) структуре видеопамяти.Изображения , получаемые при помощи сканеров и цифровых видеокамер получаются как битовые изображения.

К нелостаткам битовых изображений можно отнести большой обьем памяти , требующейся для их хранения (около 1Мбайт в режиме True Color), невозможность масштабирования без потери качества изображения , а также сложность выделения и изменения отдельных обьектов изображения .

Векторные изображения состоят из описаний отдельных элементов , поэтому они легко масштабируются.С помощью такого графического редактора , как, например, Micrografx Designer , вы без особого труда сможете выделить отдельный элемент изображения и изменить его внешний вид. Однако вывод векторных изображений выполняется,как правило,медленее,чем битовых.

Следует отметить , что некоторые устройства вывода , такие, как плоттер (графопостроитель), способны работать только с векторными, так как с помощью пера можно рисовать только линии.

Существует множество форматов файлов, предназначенных для хранения битовых и векторных изображений.В этой главе мы будем подробно рассматривать только формат файлов с расширением bmp (мы будем называть их bmp-файлами).Эти файлы хранят битовые изображения и используются в различных версиях операционной системы Microsoft Windows, Microsoft Windows NT и в графической оболочке Presentation Manager операционной системы OS/2.Векторное изображение можно хранить в виде метафайлов .

Битовое изображение в формате DDB

Как мы уже говорили , битовые изображения в формате DDB являются аппаратно зависимыми. Поэтому структура изображения в оперативной памяти зависит от особенностей аппаратуры . Как правило, изображение DDB либо загружается из ресурсов, либо создаются непосредственно в оперативной памяти. Для вывода изображений DDB на экран используются такие функции , как BitBlt и StretchBlt.

Изображения DIB, в отличие от изображений DDB являются аппаратно независимыми, поэтому без дополнительного преобразования их нельзя отображать на экране с помощью функций BitBlt и StretchBlt. В операционной системе Windows 3.x битовые изображения хранятся в файлах с расширением имени bmp, при этом используется аппаратно независимый формат DIB .

Загрузка изображений из ресурсов приложений

Самый простой способ использования битовых изображений в приложениях Windows, заключаются в том, что изображение создается графическим редактором в виде bmp-файла и описывается в файле определения ресурсов приложения при помощи оператора BITMAP:

```
LOGO BITMAP mylogo.bmp
```

Созданное таким образом битовое изображение можно загрузить в память при помощи функции LoadBitmap:

```
HBITMAP WINAPI LoadBitmap (HINSTANSE hinst, LPCSTR lpszBitmap);
```

Параметр hinst определяет идентификатор копии приложения, из ресурсов которого нужно загрузить изображение. Идентификатор ресурсов изображения задан параметром lpszBitmap. Функция LoadBitmap возвращает идентификатор загруженного изображения или NULL при ошибке.

После использования приложение должно удалить битовое изображение. Для этого лучше всего воспользоваться макрокомандой DeleteBitmap В качестве параметра этой макрокоманде можно передать идентификатор удаляемого изображения.

Приложение может определить параметры загруженного изображения вызвав функцию GetObject:

```
int WINAPI GetObject (HGDIOBJ hgdiobj, //идентификатор объекта
                      int cbBuffer,    //размер буфера
                      void FAR* lpvObject);//адрес буфера
```

С помощью этой функции можно получить разнообразную информацию об объектах GDI, таких, как логические перья, кисти, шрифты или битовые изображения . Для нас интересно использование этой функцией с целью получения параметров изображения. Идентификатор изображения должен передаваться через параметр hgdiobj. Параметр lpvObject должен указывать на структуру типа BITMAP, в которую будут записаны сведения об изображении. Через параметр cbBuffer следует передать размер структуры BITMAP:

```
BITMAP bm;
HBITMAP hBitmap;
GetObject(hBitmap, sizeof(BITMAP),(LPSTR) &bm);
```

Структура BITMAP и указатели на нее описаны в файле windows.h:

```
typedef struct tagBITMAP
{  int     bmType;
   int     bmWidth;
   int     bmHeight;
   int     bmWidthBytes;
   BYTE    bmPlanes;
   BYTE    bmBitsPixel;
   void FAR* bmBits;
} BITMAP;
typedef BITMAP*       PBITMAP;
typedef BITMAP NEAR* NPBITMAP;
typedef BITMAP FAR*  LPBITMAP;
```

Назначение отдельных полей этой структуры.

Поле	Описание
bmType	Тип битового изображения, должен быть равен нулю
bmWidth	Ширина битового изображения в пикселах, должна быть больше нуля
bmHeight	Высота битового изображения в пикселах, должна быть больше нуля
bmWidthBytes	Размер памяти , занимаемый одной строкой , растра битового изображе ния. Это значение должно быть четным, так как массив изображения состоит из целых чисел размером 16 бит.Таким образом, произведение bmWidthBytes*8 должно быть кратно 16. Кроме того, это произведение должно быть больше или равно произведению bmWidth*bmBitsPixel
bmPlanes	Количество плоскостей в битовом изображении. В зависимости от типа видеоадаптера и его режима работы для представления цвета одного пиксела может использоваться несколько бит, расположенных в одной или нескольких областях видеопамяти
bmBitsPixel	Количество бит, используемых для представления цвета пиксела. Если используется несколько плоскостей, то это поле содержит количество бит одной плоскости, используемых для представления цвета пиксела
bmBits;	Указатель на массив, содержащий биты изображения

Загрузив битовое изображение из ресурсов приложения, вы можете определить его размеры, узнав количество цветовых плоскостей и количество бит в одной плоскости, определяющих цвет пиксела. Кроме этого, вы можете получить указатель на область памяти, содержащую биты изображения.

Для монохромных битовых изображений используется одна плоскость. Для определения цвета пиксела (черный или белый) используется один бит памяти. Раз-

мер памяти, занимаемый одной строкой растра битового изображения, кратен величине 16 бит.
Для изображений DDB используется система координат, соответствующая режиму отображения MM_TEXT, то есть система координат6 принятая для устройства отображения.

Рисование изображения DDB

В програмном интерфейсе Windows (а точнее , в програмном интерфейсе GDI) нет функции предназначенной для рисования битовых изображений. Используется следующая последовательность действий.

* Прежде всего надо создать контекст памяти , совместимый с контекстом отображения реального устройства вывода. Для этого следует воспользоваться функцией CreateCompatibleDC.

* Далее необходимо выбрать предварительно загруженное битовое изображение в контекст памяти с помощью функции SelectObject, указав ей в качестве параметров идентификатор контекста памяти и идентификатор загруженного отображения.

* Затем нужно скопировать битовое изображение из контекста памяти в контекст отбражения, вызвав функцию BitBlt или StretchBlt. При этом изображение будет нарисовано на устройстве вывода, которое соответствует контексту отображения.

Рассмотрим реализацию этой последовательности действий на примере функции DrawBitmap:

```
#define STRICT
#include <windows.h>
void DrawBitmapRop(HDC hDC, int x, int y, HBITMAP hBitmap);
{
HBITMAP hbm, hOldbm;
HDC hMemDC;
BITMAP bm;
POINT ptSize, ptOrg;
//созданием контекст памяти, совместимый с контекстом отображения
hMemDC=CreateCompatibleDC(hDC);
//выбираем изображение bitmap в контекст памяти
hOldbm=(HBITMAP)SelectObject(hMemDC, hBitmap);
// если небыло ошибок , продолжаем работу
 if(hOldbm){
//для контекста памяти - тот же режим отображения, в контексте отображения
SetMapMode(hMemDC, GetMapMode(hDC));
//определяем размеры изображения
GetObject(hBitmap, sizeof(BITMAP),(LPSTR) &bm);
ptSize.x=bm.bmWidth; //ширина
ptSize.y=bm.bmHeigth; //высота
//преобразуем координаты устройства в логические для устройства вывода
DPtoLP(hDC, &ptSize, 1); ptOrd.x=0; ptOrd.y=0;
  //преобразуем координаты устройства в логические для контекста памяти
  DPtoLP(hMemDC, &ptOrd, 1);
```

```
  //рисуем изображение bitmap
  BitBlt(hDC, x, y, ptSize.x, ptSize.y,  hMemDC, ptOrd.x, ptOrd.y, SRCCOPY);
    //восстанавливаем контекст памяти
  SelectObject (HmemDC, hOldbm);
}
//удаляем контекст памяти
DeleteDC(hMemDC);
}
```

Для копирования битов изображения из контекста памяти в контекст отображения функция DrawBitmap использует функцию BitBlt :

```
BOOL  WINAPI BitBlt(
HDC hdcDest,  //контекст для рисования
int nXDest, //x-координата верхнего угла области рисования
int nYDest, //y-координата верхнего угла области рисования
int nWidth,//  ширина изображения
int nHeight, //  высота изображения
HDC hdcSrc, //идентификатор исходного контекста
int nXSrc, //x-координата верхнего левого угла исходной облласти
int nYSrc, //y-координата верхнего левого угла исходной облласти
DWORD dwRop);//код растровой операции
```

Функция копирует битовое изображение из исходного контекста hdcSrc в контекст отображения hdcDest. Возвращаемое значение равно TRUE при успешном завершении или FALSE при ошибке.

Размеры копируемого изображения задаются параметрами nWidth и nHeight. Координаты верхнего левого угла изображения в исходном контексте определяются параметрами nXSrc и nYSrc, а в контексте, куда копируется изображение,- параметрами nXDest и nYDest. Последний параметр dwRop определяет растровуб операцию используемую для копирования.

Размеры и координаты необходимо задавать в логических единицах, соответствующих выбранному режиму отображения. В нашем случае изображение копируется из точки (0, 0) в физической системе координат в трочку (x,y) в логической системе координат.

Поэтому перед копированием изображения необходимо выполнить преобразование физических координат (0,0) в логические вызвав функцию DptoLP. После этого можно вызывать ф-цию BitBlt. Эта функция копирует битовое изображение , имеющее размеры ptSize, из контекста памяти hMemDC в контекст отображения hDC. При этом логические координаты верхнего левого угла изображения в контексте памяти находятся в структуре ptOrg. Координаты верхнего левого угла прямоугольной области в контексте отображения, куда будет копироваться изображение, передаются через параметры x и y.

После рисования битового изображения функция DrawBitmap выбирает в контекст памяти первоначально выбранное при его создании изображение, состоящее из одного монохромного пиксела, и затем удаляет контекст памяти.

В качестве кода растровой операции используется константа SRCCOPY. При этом цвет пикселов копируемого изображения полностью замешает цвет соответствующих пикселов контекста отображения.

При рисовании битовых изображений вы можете использовать растровые операции , причем цвет полученного изображения в зависимости от выбранной растровой операции может определяться цветом исходного изображения, цветом поверхности, на которой выполняется рисование , и цветом кисти, выбранной в контекст отображения.

Чаще всего используется код растровой операции SRCCOPY. В этом случае цвет кисти, выбранной в контекст отображения, не имеет значения, так как не цвет кисти, не цвет фона не влияют на цвет нарисованного изображения.

Однако вы можете использовать и другие команды растровых операций (всего их 256). В этом случае для вычисления цвета полученного после рисования пиксела можно выбрать практически любое логическое выражение,учитывающее цвет фона, цвет кисти и цвет пиксела изображения.

В файле windows.h описаны константы для наиболее полезных кодов растровых операций. Мы опишем эти константы вместе с соответствующими логическими выражениями. При этом символом S ьы будем обозначать цвет исходного изображения, символом D - цвет фона на котором выполняется рисование , и P - цвет кисти, выбранной в контекст отображения.

Код растровой операции	Логическое выражение	Описание
SRCCOPY	S	Исходное изображение копируется в кон текст отображения
SRCPAINT	S|D	Цвет полученного изображения определяется при помощи логической операции ИЛИ над цветом изображения и цветом фона SRCAND
S&D		Цвет полученного изображения определяется при помощи логической операции И над цветом изображения и цветом фона
SRCINVERT	S^D	Цвет полученного изображения определяется при помощи логич операции ИСКЛЮЧАЮЩЕЕ ИЛИ над цветом изображения и цветом фона
SRCERASE	S&~D	Цвет фона инвертируется , затем выполняется операция И над результатом и цветом исходного изображением
NOTSRCCOPY	~S	После рисования цвет изображения полуается инвертированием цвета исходного изображения
NOTSRCERASE	~(S|D)	Цвет полученного изображения получается инвертированием результата логической операции ИЛИ над цветом изображения и цветом фона
MERGECOPY	P&S	Выполняется логическая операция И над цветом исходного изображения и цветом кисти
MERGEPAINT	~S|D	Выполняется логическая операция И над нвертированным цветом исходного изображения и цветом фона
PATCOPY	P	Выполняется копирование цвета кисти

PATPAINT P|~S|D Цвет кисти комбинируется с комбинируемым цветом исходного изображения, при этом используется логическая операция И.Полученный результат комбинируется с цветом фона, так же с помощью логической операции ИЛИ

PATINVERT P^D Цвет полученного изображения определяется при помощи логической операции ИСКЛЮЧАЮЩЕЕ ИЛИ над цветом кисти и цветом фона

DSTINVERT ~D Инвертируется цвет фона

BLACKNESS 0 Область закрашивается черным цветом

WHITENESS 1 Область закрашивается белым цветом

Остальные коды приведены в документации, которая поставляется вместе с SDK.

Для рисования битовых изображений можно использовать вместо функции BitBlt функцию StretchBlt, с помощью которой можно выполнить масштабирование (сжатие или растяжение) ьитовых изображений:

```
BOOL  WINAPI  StretcBlt(
HDC hdcDest,  //контекст для рисования
int nXDest,       //x-координата верхнего угла области рисования
int nYDest,       //y-координата верхнего угла области рисования
int nWidthDest,// новая ширина изображения
int nHeightDest, // новая высота изображения
HDC hdcSrc,    //идентификатор исходного контекста
int nXSrc,         //x-координата верхнего левого угла исходной облласти
int nYSrc,        //y-координата верхнего левого угла исходной облласти
int nWidthSrc,  //ширина исходного изображения
int nHeightSrc,// высота исходного изображения
DWORD dwRop);//код растровой операции
```

Параметры этой функции аналогичны параметрам функции BitBlt, за исключением того, что ширина и высота исходного и полученного изображения должна определяться отдельно.Размеры исходного изображения (логические) задаются параметрами nWidthSrc и nHeightSrc , размеры нарисованного изображения задаются параметрами nWidthDest и nHeightDest. Возвращаемое значение равно TRUE при успешном завершении или FALSE при ошибке.

Есть еще одна функция, которая сама по себе не может рисовать битовые изображения, но часто используется для закраски прямоугольных областей экрана. Эта функция имеет имя PatBlt:

```
BOOL  WINAPI  PatBlt(
HDC hdc,  //контекст для рисования
int nX,       //x-координата верхнего угла закрашиваемой области
int nY,       //y-координата верхнего угла закрашиваемой области
int nWidth,//  ширина области
int nHeight, //  высота области
DWORD dwRop);//код растровой операции
```

При использовании этой функции вы можете закрашивать области экрана с использованием следующих кодов растровых операций : PATCOPY, PATINVERT, PATPAINT, DSTINVERT, BLACKNESS, WHITENESS.

Создание изображения в памяти

Другой способ работы с изображениями в формате DDB заключается в создании их непосредственно в оперативной памяти. Вы должны подготовить массив , содержащий биты изображения , заполнить структуру типа BITMAP, которая описывает изображение , и затем вызвать функцию CreateBitmapIndirect, указав ей в качестве единственного параметра указатель lpbm на заполненную структуру типа BITMAP:

```
BITMAP CreateBitmapIndirect(BITMAP FAR* lpbm);
```

Функция вернет идентификатор битового изображения , который вы можете использовать обычным способом. Как правило , в памяти создаются монохромные изображения небольших размеров. В этом случае структура битов изображения является достаточно простой.

Например, пусть нам нужно нарисовать битовое изображение. Подготовим в памяти массив ,описывающий это изображение. Каждая строка массива соответствует одной строке сканирования битового изображения :

```
BYTE bBytes[]=
{0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,
0xff,0x00,0x00,0xff,0xff,0x00,0x00,0xff,
0xff,0x00,0x00,0xff,0xff,0x00,0x00,0xff,
0x00,0xff,0xff,0x00,0x00,0xff,0xff,0x00,
0x00,0xff,0xff,0x00,0x00,0xff,0xff,0x00,
0x00,0xff,0xff,0x00,0x00,0xff,0xff,0x00,
0xff,0x00,0x00,0xff,0xff,0x00,0x00,0xff,
0xff,0x00,0x00,0xff,0xff,0x00,0x00,0xff,
0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff};
```

При этом нам необходимо принимать во внимание, что размер одной строки сканирования изображения должен быть кратен 16 битам, то есть двум байтам. Однако сам по себе приведенный выше массив бит не содержит не содержит информации о размере изображения или о количестве бит , обозначающих цвет одного пиксела. Для формирования битового изображения необходимо подготовить структуру типа BITMAP, содержищую все необходимые сведения:

```
BITMAP bmp={ 0,64,9,8,1,1,NULL};
```

В этом массиве указаны размеры изображения (ширина - 64 пиксела, высота- 9 пикселов), размер памяти для одной строки сканирования в байтах (равен 8), количество цветовых плоскостей (одна) и количество бит, используемых для представления цвета одного пиксела (1 бит). Указатель на массив бит будет проинициализирован непосредственно перед созданием изображения, так как сегмент данных может переместиться. После того как массив данных и структура подготовлены , можно вызывать функцию CreateBitmapIndirect, передав ей в качестве параметра указатель на структуру :

```
bmp.bmBits=(LPSTR)bBytes; bmLogo2=CreateBitmapIndirect(&bmp);
```

Непосредственно перед вызовом функции CreateBitmapIndirect следует установить в структуре типа BITMAP указатель на массив бит изображения. Есть еще одна возможность. Вы можете создать битовое изображение вызвав функцию CreateBitmap:

```
BOOL  WINAPI  CreateBitmap(
int nWidth,      //  ширина изображения
int nHeight,     //  высота изображения
UINT cbPlanes,//количество цветовых плоскостей
UINT cbBits,    //количество бит на один пиксел
const void FAR* lpvBits);//указатель на массив бит
```

Через параметры этой функции передаются значения, которые необходимо записать в структуру типа BITMAP перед вызовом функции CreateBitmapIndirect.

Функции CreateBitmap и CreateBitmapIndirect возвращают идентификатор созданного в памяти изображения, который можно использовать для выбора изображения в контекст памяти, или NULL при ошибке.

Другие функции для работы с изображениями DDB

Если вы создали изображение DDB, то пользуясь его идентификатором, нетрудно скопировать в него новый массив бит, соответствующий новому изображению. Для этой цели можно воспользоваться функциейSetBitmapBits :

```
LONG    WINAPI SetBitmapBits(HBITMAP hbmp,DWORD cBits,const void FAR*
lpvBits);
```

Первый параметр этой функции предназначен для передачи идентификатора битового изображения. Параметр lpvBits является указателем на массив бит , размер этого массива задается при помощи параметра cBits. При успешном выполнении функция возвращает количество байт, использованных для изменения изображения, если произошла ошибка, возвращается нулевое значение.

Обратную операцию (чтение массива памяти изображения) можно выполнить при помощи функции GetBitmapBits:

```
LONG    WINAPI GetBitmapBits(HBITMAP hbmp,LONG cbBuffer,void FAR* lpvBits);
```

Эта функция копирует байты изображения hbmp в массив lpvBits, причем размер массива указвается через параметр cbBuffer. Функция возвращает количество скопированных в буфер байт в случае нормального завершения если произошла ошибка, возвращается нулевое значение.

Если вам нужно создать цветное битовое изображение DDB, можно воспользоваться функцией

```
HBITMAP  WINAPI  CreateCompatibleBitmap(
HDC hdc,    //контекст отображения
int nWidth,  //  ширина изображения
int nHeight) //  высота изображения
```

Эта функция создает неинициализированное изображение шириной nWidth пикселов и высотой nHeight пикселов, причем формат изображения соответствует контексту отображения hdc. Ваше приложение может выбрать неинициализированное изображение, созданное при помощи функции CreateCompatibleBitmap, в контекст памяти (совместимый с тем же контекстом отображения). Затем оно может нарисовать в контексте памяти все что угодно, используя обычные функции GDI , такие как LineTo (передавая им в качестве первого параметра идентификатор контекста памяти). После этого приложение может вызвать функцию BitBlt для отображения результата в окне приложения. Если желательно создать удаляемое (discardable)

битовое изображение, вместо предыдущей функции вы можете вызвать функцию
HBITMAP WINAPI CreateDiscardableBitmap(
HDC hdc, //контекст отображения
int nWidth, // ширина изображения
int nHeight) // высота изображения

Она имеет параметры, аналогичные параметрам CreateCompatibleBitmap .

Если изображение, созданное с использованием этой функции , не выбрано в контекст памяти, оно может быть удалено. При попытке выбрать в контекст удаленное изображение функция SelectBitmap вернет нулевое значение. В этом случае необходимо удалить изображение макрокомандой DeleteBitmap, после чего создать его заново.

Битовые изображения в формате dib

Битовые изображения DDB имеют один существенный недостаток-они "привязаны" к контному типу устройства вывода . В графической оболочке Presentation Manager операционной системы OS/2 впервые был использован аппаратно-независимый формат для хранения изображений , который называется DIB.

В операционной системе Windows этот формат получил свое дальнейшее развитие. В частности, была добавлена возможность хранения изображения в компрессованном виде. К сожалению, использованный алгоритм компрессии дает хорошие результаты только для таких изображений , которые содержат болдьшие одинаково закрашенные области. Операционная система Windows NT позволяет использовать новые форматы изображений DIB и произвольные методы компрессии , такие , как, например, JPEG (очень эффектный метод компрессии графических изображений, при котором можно ценой потери качества получить практически любую степень
сжатия).

В программном интерфейсе Windows нет функции специально предназначенной для рисования битовых изображений DIB. Поэтому если вы создаете приложение которое может отображать bmp-файлы на экране или печатать их на принтере , вам прийдется иметь дело с внутренней структурой этих файлов. Мы познакомим вас с форматом всех необходимых структур данных , имеющих отношение к bmp-файлам.

Форматы файлов и структур данных

Среди структур bmp-файлов представляют интерес 2 формата –графической оболочки Presentation Manager операционной системы OS/2 и Windows.
Ваше приложение может преобразовать формат bmp-файлов оболочки Presentation Manager , к формату Windows , как это делают стандартные приложения Windows.

Формат bmp-файлов Windows

Файл содержащий битовое изображение , начинается со структуры BITMAPFILEHEADER. Эта структура описывает тип файла и его размер, а также смещение области битов изображения. Сразу после структуры BITMAPFILEHEADER в файле следует структура BITMAPINFO, которая содержит описание изображения и таблицу цветов. Описание изображения (размеры изображения, метод компрессиии и т.д.) находится в структуре BITMAPINFOHEADER. Внекоторых случаях (не всегда)

в файле может присутствовать таблтца цветов (как массив структур RGBQUSD) , присутствующих в изображении. Биты изображения обычно располагаются сразу после таблици цветов. Точное значение смещения битов изображения находится в структуре BITMAPFILEHEADER .

Структура BITMAPFILEHEADER

```
typedef struct tagBITMAPFILEHEADER
{
   UINT    bfType;
   DWORD   bfSize;
   UINT    bfReserved1;
   UINT    bfReserved2;
   DWORD   bfOffBits;
} BITMAPFILEHEADER;
typedef BITMAPFILEHEADER*      PBITMAPFILEHEADER;
typedef BITMAPFILEHEADER FAR* LPBITMAPFILEHEADER;
```

Структура BITMAPFILEHEADER одинакова как для bmp-файлов Windows, так и для bmp-файлов оболочки Presentation Manager. И втом и в другом случае она расположена в начале файла и обычно используется для идентификации типа файла. Поля этой структуры.

Поле	Описание
bfType	Тип файла. Поле содержит значение 0x4D42 (текстовая строка "BM". Анализируя содержимое этого поля приложение может идентифицировать файл как содержащий битовое изображение
bfSize	Размер файла в байтах.
fReserved1	Зарезервировано, должно быть равным нулю
bfReserved2	Зарезервировано, должно быть равным нулю
bfOffBits	Смещение битов изображения от начала файла в байтах. Область изображения не обязательно должна быть расположена сразу вслед за заголовками файла или таблицей цветов(если она есть)

В структуре BITMAPFILEHEADER для нас важны два поля-поле bfType, определяющее тип файла и поле bfOffBits определяющее смещение битов, из которых формируется изображение. Остальные поля можно проигнорировать.

Сразу после структуры BITMAPFILEHEADER в bmp-расположена структура BITMAPINFO (для изображений Windows) или BITMAPCOREINFO (для изображений Presentation Manager). Структура BITMAPINFO и указатели на нее описаны в файле windows.h следующим образом:

```
typedef struct tagBITMAPINFO
{
BITMAPINFOHEADER bmiHeader;
RGBQUAD          bmiColors[1];
}
BITMAPINFO
```

```
typedef BITMAPINFO*  PBITMAPINFO;
typedef BITMAPINFO FAR* LBITMAPINFO;
```
Структура BITMAPINFOHEADER описывает размеры и способ представления цвета в битовом изображении:
```
typedef struct tagBITMAPINFOHEADER
{ DWORD   biSize;
   LONG    biWidth;
   LONG    biHeight;
   WORD    biPlanes;
   WORD    biBitCount;
   DWORD   biCompression;
   DWORD   biSizeImage;
   LONG    biXPelsPerMeter;
   LONG    biYPelsPerMeter;
   DWORD   biClrUsed;
   DWORD   biClrImportant;
} BITMAPINFOHEADER;
typedef BITMAPINFOHEADER*      PBITMAPINFOHEADER;
typedef BITMAPINFOHEADER FAR* LPBITMAPINFOHEADER;
```

Опишем назначение отдельных полей этой структуры:

поле	описание
biSize	Размер структуры BITMAPINFOHEADER в байтах
biWidth	Ширина битового изображения в пикселах
biHeight	Высота битового изображения в пикселах
biPlanes	Количество плоскостей в битовом изображении. Содержимое этого поля должно быть равно единице
biBitCount	Количество бит на один пиксел. Может быть равно 1, 4, 8, 24.Для новых 16- и 32- битовых форматов файлов DIB, используемых в Windows NT , в этом поле могут находиться так же значения 16 и 32.
biCompression	Метод компрессии. Может принимать одно из следующих значений : BI_RGB- компрессия не используется. BI_RLE4- компрессия изображений, в которых для представления пикселов используется 4 бита.При использовании этого метода компрессии содержимое поля biBitCount должно быть равно 4; BI_RLE8- компрессия изображений, в которых для представления пикселов используется 8 бита.При использовании этого метода компрессии содержимое поля biBitCount должно быть равно 8; BI_BITFIELDS- другой формат компрессии. Это значение используется для Windows NT. Соответствующая констан-

та описана в файле windows.h, который представляется вместе со средствами разработки приложений Windows NT

biSizeImage Размер изображения в байтах.Это поле содержит размер, необходимый для хранения разжатого изображения.Если компрессия не используется (в поле biCompression находится значение BI_RGB) содержимое поля biSizeImage может быть равно нулю

biXPelsPerMeter Разрешение устройства вывода по горизонтале в пикселах на метр, необходимое для вывода битовое изображение без искажения. Это поле используется невсегда. Если оно не используется , в нем следует установить нулевое значение.

biYPelsPerMeter Разрешение устройства вывода по вертикали в пикселах на метр, необходимое для вывода битовое изображение без искажения. Это поле используется невсегда. Если оно не используется , в нем следует установить нулевое значение.

biClrUsed Размер таблицы цветов. Это поле определяет размер массива структур RGBQUAD, расположенного в файле сразу после структуры BITMAPINFOHEADER.Если в этом поле находится нулевое значение, размер таблицы цветов зависит от количества бит используемых для представления цвета одного пиксела (поле biBitCount)

biClrImportant Количество цветов, необходимое для отображения файла без искажения. Обычно в этом поле находится нулевое значение , в этом случае важны все цвета

Сразу после структуры BITMAPINFOHEADER в файле находится таблица цветов . Эта таблица содержит массив структур RGBQUAD:

```
typedef struct tagRGBQUAD
{
   BYTE    rgbBlue;
   BYTE    rgbGreen;
   BYTE    rgbRed;
   BYTE    rgbReserved;
} RGBQUAD;
typedef RGBQUAD FAR* LPRGBQUAD;
```

Поля rgbBlue, rgbGreen, rgbRed, содержат RGB - компоненты цветов поле rgbReserved зарезервированно и должно содержать нулевое значение.

Файл битового изображения может содержать таблицу цветов, а может и не содержать ее. Покажем зависимость размера таблицы цветов от значения поля biBitCount (количество бит, используемых для представления одного пиксела):

значение поля biBitCoun	размер таблицы цветов
1	2
4	16

8	256
24	не используется

Если содержимое поля biClrUsed отлично от нуля, используется таблица цветов уменьшенного размера. В ней описанны только те цвета, которые содержатся в изображении.

Биты изображения

Каждая строка битового изображения (scan line) хранится в буфере, длина которого кратна двойному слову DWORD.Для черно белых изображений каждый бит буфера используется для представления цвета одного пиксела (если не используется компрессия). Формат области битов изображения для bmp-файлов не зависит от аппаратных особенностей какого-либо устройства отображения.Всегда используется одна цветовая плоскость, при этом цвет пиксела представляется разным количеством бит памяти , в зависимости от цветового разрешения изображения.

Последнее обстоятельство может оказаться полезным , например, при печати изображения на принтере, способном работать с оттенками серого цвета. В этом случае драйвер принтера избыточную цветовую информацию, содержащуюся в bmp-файле, для формирования полутонового изображения.

В памяти изображение хранится в перевернутом виде.

В отличие от изображений DDB при работе с изображениями DIB используется система координат , начало которой расположено в левом нижнем углу, а координатные оси направлены вверх и вправо.

Если ваше приложение загружает изображение в оперативную память , необходимо предварительно определить размер буфера. Если изображение не компрессовано (то есть в поле biCompression структуры BITMAPINFOHEADER находится значение BI_RGB), для вычисления размера буфера можно воспользоваться следующей формулой:dwSizeImage=((nBits+31)/32x4) x biHeight, где nBits=biBitCount x biWidth

В этой формуле переменные biHeight ,biBitCount и biWidth являются полями структуры BITMAPINFOHEADER.

Если же изображение хранится в компрессованом формате(в поле biCompression структуры BITMAPINFOHEADER находится значение BI_RLE4 или BI_RLE8), нужный размер буфера можно получить из поля biSizeImage структуры BITMAPINFOHEADER.

Формат области битов изображения для цветных файлов сложнее, однако размер области определяется аналогично. Как правило, приложения не формируют сложные изображения непосредственно в оперативной памяти, а загружают их из bmp-файлов.Однако иногда требуется создать простое черно-белое изображение непосредственно в памяти. В этом случае требуется знать точный формат области битов изображения.

Но в большинстве случаев для вывода готового изображения требуется только умение определить размер области битов изображения для получения соответствующего буфера, куда эта область загружается из bmp-файла перед выводом изображения на экран.Поэтому мы не будем рассматривать формат области битов изображения для цветных bmp-файлов.

Для bmp-файлов в формате DDB для представления цвета одного пиксела используются несколько ьит памяти.Например, в 16-цветных файлах цвет пиксела пред-

ставляется при помощи 4 бит памяти, в 256-цветных файлах для этой цели используется 8 бит , файлы True Color используют 24 бита.

Рисование изображений dib

Процесс рисования изображений DIB включает в себя несколько этапов.

Сначала необходимо загрузить bmp-файл в оперативную память и убедиться в том , что этот файл действительно содержит изображение DIB.Ваше приложение может полностью проигнорировать bmp-файлы в формате Presentation Manager (Paintbrush в Windows версии 3.1) или выполнить их преобразование в формат Windows, что намного лучше. Следует также проверить формат заголовка BITMAPINFOHEADER.

Затем нужно определить размер таблицы цветов(если она есть). Если в DIB есть таблица цветов, ее следует преобразовать в палитру. Непосредственно перед рисованием изображения DIB созданная палитра должна быть выбрана в контекст отображения и реализована.Если bmp- файл содержит изображение с высоким цветовым разрешением , в файле нет таблицы цветов. В этом случае нет необходимости создавать политру.

После создания палитры следует определить адрес битов изображения. Напомним, что смещение битов изображения находится в поле bfOffBits структуры BITMAPINFOHEADER.Если содержимое этого поля равно нулю , можно вычислить адрес битов изображения, исходя из размера заголовка и размера таблицы цветов.

В заключени считанное и проверенное изображение DIB можно нарисовать, использовав один из двух способов.

Первый способ заключается в преобразовании изображения DIB в изображение DDB с помощью функции SetDIBits.Полученное таким образом изображение DDB может быть выбранно в контекст памяти и нарисована обычным способом при помощи функции BitBlt или StretchBlt.

Второй способ основан на использовании функции StretchDIBits, которая сама выполняет необходимые преобразования , однако в некоторых случаях работает медленее функции BitBlt.

Если изображение DIB содержит таблицу цветов и устройство выводов способно работать с цветовыми палитрами, ваше приложение должно обрабатывать сообщение WM_PALETTECHANGED и WM_QUERYNEWPALETTE.

Загрузка bmp-файла и проверка заголовков

Вы можете загрузить в операративную память весь bmp-файл сразу или вначале только заголовки, а затем таблицу цветов и биты изображений.

Прочитав файл в память , следует убедиться , что его первые два байта содержит значение 0x4d42("BM"). Если это так , нужно определить формат bmp-файла для этого следует проанализировать содержимое поля biSize , расположенные сразу после заголовка BITMAPINFOHEADER. Для файла в формате Windows в этом поле должно быть значение 40, что соответствует размеру структуры BITMAPINFOHEADER. Для файлов в формате Presentation Manager в этом поле должно находится значение 12 (размер структуры BITMAPCOREHEADER) .

Приложение может отвергнуть файл в формате Presentation Manager , и это не будет большим недостатком для приложения Windows. В случае необходимости

bmp-файлы Presentation Manager могут быть преобразованы в формат Windows , например с помощью приложения Paintbrush.

Убедившися в том , что вы загрузили bmp-файл в формате Windows , следует проверить содержимое полей структуры BITMAPINFOHEADER .

Следует проверить поля biPlanes , biBitCount и biCompression. Вы можете использовать для проверки следующие критерии :

поле	Критерии проверки
biPlanes	Должно быть равно единице
biBitCount	Может быть равно 1, 4,8, 24.Для новых 16- и 32-битовых форматов файлов DIB, используемых в Windows NT , в этом поле могут находиться так же значения 16 и 32.Если ваше приложение не умеет обрабатывать такие файлы, данную ситуацию следует рассматривать как ошибочную
biCompression	Может принимать одно из следующих значений : BI_RGB, BI_RLE4, BI_RLE8 при использовании метода компрессии BI_RLE4 содержимое поля biBitCount должно быть равно 4. при использовании метода компрессии BI_RLE8 содержимое поля biBitCount должно быть равно 8. Ваше приложение может ограничиться обработкой bmp- файлов в формате BI_RGB, как это делает , например, приложение Paintbrush

Создание цветовой палитры

Процесс создания цветовой палитры несложен. Вначале нужно убедиться в том что bmp-файл содержит таблицу цветов. Если размер таблицы цветов не равен нулю , следует заказать память для структуры LOGPALETTE, заполнить соответствующим образом заголовок и переписать в палитру цвета из таблици цветов:

```
lpPal-.palVersion=0x300;
lpPal-.palNumEntries=wNumColors;
for(i=0;i,wNumColors;i++)
{lpPal-.palPalEntry[i].peRed=lpbmi-.bmiColors[i].rgbRed;
 lpPal-.palPalEntry[i].peGreen=lpbmi-.bmiColors[i].rgbGreen;
 lpPal-.palPalEntry[i].peBlue=lpbmi-.bmiColors[i].rgbBlue;
 lpPal-.palPalEntry[i].peFlags=0;
}
```

Палитра создается с помощью функции CreatePalette: hPal= CreatePalette(lpPal);

Рисование DIB

Если отображаемый bmp-файл содержит таблицу цветов и на предыдущем эжтапе была создана палитра , ее следует выбрать в контекст отображения и реализовать:

```
hOldPal=SelectPalette(hdc, hPal, FALSE);   RealizePalette(hdc);
```

После этого вы можете нарисовать DIB одним из двух способов.

Первый способ заключается в преобразовании изображения DIB в изображение DDB с последующим рисованием изображения DDB. Вы уже умеете рисовать изображение DDB, для этого следует выбрать в специально созданный контекст памяти и затем отобразить функцией BitBlt или StretchBlt.

Для преобразования DIB в DDB вы должны использовать функцию SetDIBits:

```
int    WINAPI  SetDIBits(
HDC hdc,              //контекст отображения
HBITMAP hbmp,   //изображение DDB
UINT  uStartScan,    // номер первой строки
UINT uScanLines,     // количество строк
const void FAR* lpvBits,  //биты изображения
BITMAPINFO FAR* lpbmi,  //заголовок изображения
UINT fuColorUse);          //содержимое таблици цветов
```

Параметр hdc должен содержать идентификатор контекста отбражения, в котором будет отображаться полученное изображение DDB. Через параметр hbmp следует передать идентификатор битового изображения DDB, совместимого с контекстом hdc. Его можно создать при помощи функции CreateCompatibleBitmap. После преобразования это изображение можно будет использовать для рисования функцией BitBlt или
StretchBlt.

Параметр uScanLines задает номер строки сканирования битового изображения начиная с которого будет выполняться преобразование. Если вам нужно нарисовать все изображение целиком , для этого параметра следует задать значение 0.

Параметр uScanLines определяет количество строк сканирования участвующих в преобразовании. Если нужно преобразовать все изображение , для этого параметра следует задать высоту изображения, взятую из заголовка BITMAPINFOHEADER .

Через параметр lpvBits следует передать указатель на область памяти, содержащую биты изображения в формате DIB.

В процессе преобразования функция SetDIBits использует заголовок bmp-файла BITMAPINFO, указатель на который следует передать через параметр lpbmi.

Последний параметр fuColorUse указывает функции на содержимое таблицы ыветов , которая расположена сразу после структуры BITMAPINFOHEADER. Возможны два значения - DIB_RGB_COLORS и DIB_PAL_COLORS.

Если указанно значение DIB_RGB_COLORS , таблица цветов содержит RGB-цвета , которые можно использовать для создания палитры.Если же указанно значение DIB_PAL_COLORS,таблица цветов содержит 16-битовые ссылки на элементы системной палитры.

Если вы загрузили bmp-файл в память, таблица цветов обычно содержит именно RGB -цвета , поэтому для преобразования и последующего рисования изображения вы должны указать значение DIB_RGB_COLORS .

Возвращаемая функции SetDIBits значение равно количеству преобразованных строк сканирования или нулю при ошибке.

Поясним процесс рисования на простом примере.

Пусть мы загрузили изображение DIB с шириной wWidth и высотой wHeight. Создаем изображение DDB, совместимое с контекстом отображения hdc и имеющее теже размеры.Воспользуемся функцией CreateCompatibleBitmap :
hbmp=CreateCompatibleBitmap (hdc ,wWidth, wHeight);

Создадим так же контекст памяти , совместимый с контекстом отображения :
hMemDC=CreateCompatibleDC(hdc);

Далее вызываем функцию SetDIBits,которая преобразует биты изображения DIB и запишет их в созданные нами изображения DDB с идентификатором hbmp:
SetDIBits(hdc, hbmp, 0, wHeight,
lpDibBits, (LPBITMAPINFO)lpih, DIB_RGB_COLORS);

Теперь нам нужно нарисовать полученное изображение DDB. Для этого выбираем его в контекст памяти и переносим в контекст отображения, например функцией BitBlt:
hbmp=(HBITMAP)SelectObject (hMemDC, hbmp);
BitBlt(hdc, x, y,,wWidth, wHeight ,hMemDC, 0,0,SRCCOPY);

Все!Изображение нарисовано.Теперь можно удалить контекст памяти, незабыв перед этим выбрать в него старое битовое изображение (размером 1x1 пиксел):
DeleteObject(SelectObject (hMemDC, hbmp));
DeleteDC(hMemDC);

Второй способ нарисовать DIB немного проще:
StretchDIBits (hdc,
x, y,,wWidth, wHeight ,
0, 0,,wWidth, wHeight ,
lpDibBits, (LPBITMAPINFO)lpih,
DIB_RGB_COLORS),SRCCOPY);

Прототип функции StretchDIBits выглядит несколько громоздко, однако эта функция дополнительно позволяет масштабировать рисуемое изображение. Функция имеет параметры аналогичные параметрам функций StretchBLT и SetDIBits :
I

nt WINAPI StretchDIBits(
HDC hdc, //контекст для рисования
int nXDest, //x-координата верхнего угла области рисования
int nYDest, //y-координата верхнего угла области рисования
int nWidthDest,// новая ширина изображения
int nHeightDest, // новая высота изображения
int nXSrc, //x-координата верхнего левого угла исходной облласти
int nYSrc, //y-координата верхнего левого угла исходной облласти
int nWidthSrc,//ширина исходного изображения
int nHeightSrc, //высота исходного изображения
const void FAR* lpvBits,//биты изображения
LPBITMAPINFO lpbmi, //заголовок изображения
UINT fuColorUse, // содержимое таблицы цветов
DWORD dwRop); //код растровой операции

Возвращаемое значение равно количеству преобразованых строк сканирования или нулю при ошибке.

Преобразование DDB в DIB

Если перед вами встанет задача преобразования DDB в DIB (например,для последующей записи изображения в bmp- файл) , вам не обойтись без функции GetDIBits:

```
int WINAPI GetDIBits(
HDC hdc ,//контекст отображения
HBITMAP hbmp,// изображения DDB
UINT uStartScan, //номер первой строки
UINT uScanLines, // количество строк
void FAR* lpvBits, //биты изображения
BITMAPINFO FAR* lpbmi, //заголовок изображения
UINT fuColorUse);// содержимое таблицы цветов
```

Параметры этой функции полностью аналогичны параметрам функции SetDIBits, однако действия прямопротивоположные. Функция преобразует биты изображения в формат DIB и записывает их по адресу заданному параметром lpvBits.Дополнительно заполняется заголовок lpbmi(если параметр lpvBits указан как NULL, функция ограничивается заполнением заголовка изображения).

Если вы собираетесь сохранить изображение DIB в bmp-файле, вы должны самостоятельно сформировать заголовок файла BITMAPFILEHEADER.

Шрифты

Для приложений доступны растровые, векторные и масштабируемые шрифты.Кроме этого, приложения могут использовать шрифты встроенные или отображаемые в устройство выводов, например в принтер.

Растровые шрифты удобны для вывода текста на экран (особенно при малой высоте букв)однако для каждого видео режима и типа контроллера используется свой набор растровых шрифтов.

Векторные шрифты используются в тех случаях, когда в качестве устройства вывода применяется плоттер.Более того , для вывода на плоттер можно использовать только векторные шрифты.

Однако подавляющее большинство компьютеров предназначено для подготовки текстовой и табличной информации, причем для печати используются матричные струйные или лазерные принтеры.В этом случае приемлемое качество документа может быть получена только с использованием масштабируемых шрифтов(в некоторых случаях принтерных шрифтов).

Масштабируемые шрифты True Type доступные приложениям Windows, нетолько сохраняют свое начертание при произвольном изменении высоты букв , но и обладают другими достоинствами.

Существует и возможность вывода строк текста, расположенных под любым углом относительно горизонтальной оси. Растровые шрифты позволяют располагать строки текста только в горизонтальном направлении, что создает трудности.

Еще одно приимущество масштабируемых шрифтов True Type связано с тем что вы можете встроить такой шрифт непосредственно в документ .

Стандартный набор шрифтов True Type поставляемых в составе Windows не всегда удовлетворяет пользователей. Поэтому они приобратают дополнительные шрифты у независимых разработчиков.Использование нестандартных шрифтов может привести к проблемам при необходимости переноса документов из одного компьюте-

ра в другие, так как там нужного шрифта может не оказаться. Вы конечно , можете скопировать нужный шрифт и перенести вместе с документами, однако эта процедура может быть запрещена разработчикам шрифта.

Проблему переноса документа на другой компьютер можно решить используя шрифты, встроенные в документ.Пользователь,например, готовит документ в текстовом процессоре Microsoft Word for Windows 6.0 и встроить в него все использованые шрифты.При переносе такого документа на другой компьютер эти шрифты можно будет использовать для просмотра и , возможно6 редактирования (только этого) документа.Возможность редактирования с использованием встроенного шрифта определяется разработчиком шрифта.

Классификация шрифтов

Можно сказать, что шрифт состоит из изображений (рисунков) отдельных символов - глифов (glyph). Для внутреннего представления глифа в файле шрифта True Type
используется описание контуров, причем один глиф может содержать несколько контуров. Глифы могут иметь различный внешний вид (typeface). Операционная система Windows класифицирует шрифты на несколько типов или семейств (font family): Modern, Roman, Swiss, Script, Decorative .

Шрифты семейства Modern имеют одинаковую ширину букв. Таким шрифтом оформлены все листинги программ. Шрифты семейства Roman содержат буквы различной ширины, имеющие засечки.Семейство Swiss отличается тем, что при переменной ширине букв они не имеют засечек. Буквы в шрифтах семейства Script как бы написаны от руки. Семейство Decorative содержит глифы в виде небольших картинок
(пиктограмм).

Приложения Windows могут заказывать штриф , ссылаясь на название соответствующего семейства, однако в зависимости от состава имеющихся шрифтов Windows может представить в ваше распоряжение не тот шрифт, какой бы вам хотелось.

Другая важная характеристика шрифта - это размер функв. Для описания вертикального размера букв шрифта используется несколько параметров. Не останавливаясь на тонкостях, отметим, что шрифты содержащие буквы разного размера являются разными шрифтами.

Растровые шрифты, относящиеся к одному семейству, но имеющие разные размеры букв, хранятся в отдельных файлах. Благодаря аозможность масштабирования шрифтов True Type для них нет необходимости в отдельном хранении глифов различных размеров.

GDI может выполнять масштабирование растровых шрифтов, увеличивая (но не уменьшая)размер букв.Результат такого масштабирования при большом размера букв обычно неудовлетворительный, так как на наклонных линиях контура букв образуются зазубрины.

Векторные шрифты легко поддаются масштабированию , поэтому для хранения шрифта одного семейства , но разного размера, можно использовать отдельный файл.

Вы знаете , что шрифты могут иметь нормальное (normal),жирное (bold) или накланенное (italic) :

Еще один часто используемый атрибут оформления строк текста - подчеркивания : Иногда нужен шрифт с перечеркнутыми буквами.
GDI выполняет подчеркивание и перечеркивание самостоятельно файлы шрифтов не содержат глифы с подчеркиванием. Рстровые и векторные шрифты хранятся в системном каталоге Windows в файлах с расширением имени fon .

Глифы масштабируемых шрифтов находятся в файлах с расширением имени ttf , при чем сами эти файлы могут располагаться в самом каталоге. В процессе регистрации масштабируемых шрифтов Windows создает в своем системном каталоге файлы с расширением имени fot , которые содержат ссылки на соответсвующие ttf-файлы.

С помощью приложения Control Panel вы можете добавлять или удалять любые шрифты. Следует однако , учитывать ограничения : в системе можно одновременно использовать не более 253 шрифтов. Для представления жирного и наклонного начертания используются отдельные масштабируемые шрифты. Чрезмерное количество установленных шрифтов может привести к снижению производительности системы.

Выбор шрифтов в контексте отображения

Для того чтобы написать строку текста заданным шрифтом, этот шрифт следует , подобно остальным объектам GDI, выбрать в контекст отображения. После этого функции TextOut, DrawText и аналогичные будут использовать для вывода текста нужный вам шрифт.

Приложения Windows могут использовать либо один из встроенных шрифтов, либо создать свой, описав требуемые характеристики шрифта.В любом случае в распоряжении пользователя будет предоставлен один из шрифтов, зарегистрированный при установке Windows или позже (с помощью Control Panel).Для выбора шрифта, соответствующего описания, используеься достаточно сложный алгоритм, учитывающий степень важности обеспечения соответствия параметров предоставленного шрифта запрошенным параметрам.

Обратите ваше внимание на одно важное обстоятельство.

Приложение заказывает шрифт, описывая его параметры.GDI анализирует запрошенные параметры и подбирает наиболее подходящий шрифт.При этом приложение не может "заставить " GDI выделить ему какой-то конкретный шрифт, указав его название или путь к файлу.Однако приложение может определить параметры шрифта, выбранного пользователем из состава установленных шрифтов, и запросить у GDI шрифт с этими параметрами.
В последнем случае будет выделен шрифт , выбранный пользователем.

Выбор встроенного шрифта

По умолчанию в контекст отображения при его создании выбирается системный шрифт, основным(и почти единственным) преимуществом которого является то, что он всегда доступен.Системный шрифт не является масштабируемым, содержит буквы переменной ширины, не имеющие засечек, для него используется кодировка ANSI.

Однако в некоторых случаях вам может понадобится шрифт с фиксированной шириной букв, или шрифт в кодировке OEM. Вы можете получить идентификатор одного из встроенных шрифтов при помощи макрокоманды GetStockFon

В качестве единственного параметра этой макрокоманде следует передать идентификатор одного из встроенных шрифтов:

Идентификатор	Описание
SYSTEM_FONT	Системный шрифт в кодировке ANSI с переменной шириной букв, используются операционной системой Windows для отображения текста в меню, заголовка окон диалоговых панелях
SYSTEM_FIXED_FONT	Шрифт в кодировке ANSI с фиксированной шириной букв. Используются в старых версиях операционной системы Windows (до версии 3.0)Как системный шрифт
ANSI_VAR_FONT	Шрифт в кодировке ANSI с переменной шириной букв
ANSI_FIXED_FONT	Шрифт в кодировке ANSI с фиксированной шириной букв
OEM_FIXED_FONT	Шрифт в кодировке OEM с фиксированной шириной букв
DEVICE_DEFAULT_FONT	Шрифт, который используется для данного устройства по умолчанию.Если устройство не имеет своих шрифтов, используется системный шрифт SYSTEM_FONT

После того как вы получили идентификатор шрифта, этот шрифт можно выбрать в контекст отображения макрокомандой SelectFont.

Первый параметр этой макрокоманды определяет идентификатор контекста отображения, в который выбирается шрифт с идентификатором hfont. Она возвращает идентификатор шрифта, который был выбран в контекст отображения раньше , до вызова SelectFont. Вам не нужно удалять встроенные шрифты, так же как не нужно удалять встроенные кисти и перья.

Определение логического шрифта

Если приложение ограничивает себя только встроенными шрифтами, оно не сможет воспользоваться всем много образием созданных масштабируемых шрифтов и даже просто не сможет изменить размер букв или сделать текст жирным или наклонным.Для полного использования шрифтовых возможностей операционной системы Windows вы должны познакомиться с процедурой определения логических шрифтов.

Приложение может получить идентификатор шрифта, указав его параметры (ьакие как размеры символов, семейство шрифта и тому подобное) функции CreateFont. Эта функция имеет 14 параметров, поэтому не слишком удобна в использовании. Вместо нее лучше использовать функции CreateFontIndirect:

```
HFONT WINAPI CreateFontIndirect(const LOGFONT FAR* lplf);
```

Функция возвращает идентификатор созданного логического шрифта, который можно выбрать в контекст отображения макрокомандой SelectFont, при этом для вывода будет подобран наиболее подходящий физический шрифт.

В качестве параметра функции CreateFontIndirect определяется указатель на структуру типа LOGFONT определенную в файле windows.h:

```
typedef struct tagLOGFONT
{
   int    lfHeight;
   int    lfWidth;
   int    lfEscapement;
   int    lfOrientation;
   int    lfWeight;
   BYTE   lfItalic;
   BYTE   lfUnderline;
   BYTE   lfStrikeOut;
   BYTE   lfCharSet;
   BYTE   lfOutPrecision;
   BYTE   lfClipPrecision;
   BYTE   lfQuality;
   BYTE   lfPitchAndFamily;
   char   lfFaceName[LF_FACESIZE];
} LOGFONT;
typedef LOGFONT*       PLOGFONT;
typedef LOGFONT NEAR* NPLOGFONT;
typedef LOGFONT FAR*  LPLOGFONT;
```

Перед вызовом функции CreateFontIndirect вы должны заполнить структуру LOGFONT нужными значениями, определяющими параметры шрифта. В неиспользованные поля следует записать нулевые значения. Можно записать нулевые значения во все поля, однако это едва ли имеет смысл. Назначение отдельных полей структуры LOGFONT .

lfHeight

Высота шрифта в логических единицах, которые зависят от установленного режима отображения.

Можно указывать положительные и отрицательные значения, а так же нуль.Если указано нулевое значение, выбираетсяы шрифт размером в 12 пунктов (значение по умолчанию).

Положительные значения определяют высоту ячеек, в которых располагается буква, что соответствует содержимому поля tmHaight структуры TEXTMETRICS. Абсолютная величина отрицательного значения определяет высоту символа, то есть tmHeight - tmInternalLeading

lfWidTh

Ширина символов в логических единицах.

Если указанно нулевое значение, используется значение по умолчанию, которое зависит от высоты шрифта и отношения масштабов по осям координат(aspect ratio) для шрифта и устройства вывода.

lfEscapement

Угол между базовой линией шрифта и координатной осью X в десятых долях градуса (угол отсчитывается в направлении против часовой стрелки).

Если в процессе отображения логического шрифта на физический будет выбран растровый или векторный шрифт текст будет выведен в горизонтальном направлении, так как вращать можно только шрифты True Type и векторные шрифты.

lfOrientation

Это поле определяет ориентацию символов шрифта.

lfWeight

Вес шрифта.

Определяет насыщенность символов шрифта и может находиться в пределах от 0 до 1000. В файле windows.h вы найдете список символических констант для этого поля.

Вы можете использовать любое из указанных там значений, однако следует иметь в виду, что многие шрифты содержат описания символов только для веса FW_NORMAL, FW_REGULAR и FW_BOLD.

flItalic

Если содержимое этого поля не равно нулю, запрашивается шрифт с наклонными буквами

lfUnderline

Если содержимое этого поля не равно нулю, запрашивается шрифт с подчеркиванием букв.

lfStrikeOut

Если содержимое этого поля не равно нулю, запрашивается шрифт с перечеркнутыми буквами.

lfCharSet

Набор символов.

Можно использовать одну из констант, определенных в файле windows.h:

lfOutPrecision

Требуемая степень соответствия параметров шрифта.

Это поле используется для того, чтобы указать GDI способ выбора между двумя шрифтами, имеющими одинаковые значение, но разный тип. Например для удовлетворения запроса можно использовать растровый или масштабируемый шрифт с названием OddType.Если в поле lfOutPrecision указать константу OUT_TT_PRECIS, будет выбран масштабируемый шрифт.

Можно указать одну из констант 0,1,2, 3.

lfClipPrecision

Поле используется для определения способа , при помощи которого обрезается отображение символа, частично попавшего за пределы области отраничения вывода (clipping region), выбранной в контекст отображения.

Можно использовать следующие константы: CLIP_DEFSULT_PRECIS, CLIP_CHARACTER_PRECIS,CLIP_STOKE_PRECIS , CLIP_MASK, CLIP_LH_ANGLES, CLIP_TT_ALWAYS, CLIP_EMBEDDED.

Если указанна константа CLIP_LH_ANGLES, направление вращения текста зависит от установленного режима отображения.

lfQuality

Качество шрифта , полученного при отображении.

Можно указывать одну из следующих констант:

Константа	Описание
DEFAULT_QUALITY	Качество не имеет значения
DRAFT_QUALITY	Низкое качество. Допустимо масштабирование шрифтов синтезирование наклонных, жирных, перечеркнутых, символов
PROOF_QUALITY	Высокое качество. Масштабирование шрифтов не допускается. При этом могут быть получены символы имеющие размер немного меньше запрошенного

lfPitchAndFamily

С помощью этого поля можно определить , нужна фиксированная или переменная ширина символов. Кроме этого , можно определить семейство, к которому должен принадлежать полученный шрифт. Фиксированная или переменная ширина символов задается при помощи констант, значения которых вы найдете в справочной системе.

lfFaceName

Строка, закрытая двоичным нулем, которая служит названием внешнего вида шрифта. Размер строки (включая закрывающий строку нуль) не должен превышать LF_FACESIZE байт. Вы можете указать , что вам нужен , например , шрифт "Arial", однако это вовсе не гарантирует , что именно этот шрифт будет предоставлен в распоряжение приложения .

Выбор созданного шрифта в контекст отображения

Если вы заполнили все нужные поля в структуре LOGFONT и затем передали адрес структуры функции CreateFontIndirect , эта функция вернет идентификатор шрифта. Вы должны выбрать шрифт с этим идентификатором в контекст отображения с помощью макрокоманды SelectFont (точно также , как для встроенных шрифтов):

hfontOldFont = SelectFont(hdc,hfont) ;

Как только в строенном шрифте отпадет необходимость его следует удалить при помощи макрокоманды DeleteFont, предварительно выбрав в контекст отображения тот шрифт , который был выбран в него раньше

Процесс отображения логического шрифта достаточно сложно. GDI сравнивает заданные в структуре LOGFONT параметры с параметрами различных шрифтов , которые можно использовать для данного устройства отображения , выбирая наиболее подходящий шрифт. Для сравнения используются пенальти (штрафные очки) , которые имеют разные весовые коефициенты. Выбирая тот шрифт для которого сумма пенальти наименьшая.

Наиболее важное поле в структуре LOGFONT - поле lfCharSet. Если в этом поле будет установлено нулевое значение, будет выбран шрифт ANSI_CHARACTER , так как значение соответствующей ему константы равно нулю.Понятно, почему это поле самое важное: если приложение запрашивает шрифт OEM_CHARSET, оно предлагает использовать для вывода кодировки OEM. Если бы GDI предоставил приложению шрифт в кодировке ANSI скорее всего строку было бы невозможно прочесть. Если же в Windows нет не одного шрифта с кодировкой OEM, приложение всеравно получит какой-нибудь шрифт, однако результат вызова текста может оказаться неудовлетворительным.

Растровые шрифты семейства Modern, Roman и Script, которые пришли из Windows 3.0, отмечен как имеющие кодировку OEM, хотя в действительности для этих шрифтов используется кодировка ANSI. Это сделанно для того , чтобы в процессе выбора GDI вначале использолвал масштабируемые шрифты перечисленных семейств, и только в крайнем случае остановил свой выбор на растровых шрифтах.

Следующее по важности поле в структуре LOGFONT - это поле lfPitchAndFamily. Оно имеет большое значение по тому, что приложение, запрашивающее шрифт с фиксированной шириной букв , может работать неправильно, если ему будет выделен шрифт с переменной шириной букв.

Далее следует поле lfFaceName, а после него - поле lfFamily.

После сравнения всех описанных полей GDI сравнивает высоту букв шрифта (поле lfHeight) , затем в сравнении принимают участие поля lfWidth, lfItalic, lfUnderline, lfStrikeOut.

Функция ChooseFont

Только что мы приближенно описали алгоритм , который используется GDI для отображения логического шрифта на физический.Нетрудно заметить, что приложение может получить от GDI совсем не тот шрифт, который ему нужен. Поэтому лучше всего предоставить пользователю возможность выбрать шрифт самостоятельно.

DLL- библиотека commdlg.dll содержит функцию ChooseFont, специально предназначенную для выбора одного из зарегистрированных в системе шрифтов.Эта функция выводит на экран диалоговую панель "Font", с помощью которой пользователь может выбрать шрифт, стиль шрифта, размер шрифта , цвет букв, может выбрать подчеркнутый или перечеркнутый шрифт.

Из списка "Font", который расположен в верхней левой части этой диалоговой панели пользователь может выбрать название шрифта. Список "Font Style"позволяет выбрать один из доступных стилей, например наклонный или жирный. Список "Size" предназначен для выбора размера шрифта. С помощью переключателей "Strikeout" и "Underline", расположены в поле "Effects"можно создать соответственно перечеркнутый и подчеркнутый шрифт.И наконец,из меню "Color" можно выбрать цвет букв.

Образец выбранного шрифта отображается в поле "Sample".

Обратите внимание на то, что в списке "Font" некоторые шрифты отмечены двойной буквой "T".Это масштабируемые шрифты True Type. Приведем прототип функции ChooseFont:

```
BOOL WINAPI ChooseColor(CHOOSEFONT FAR* lpcf);
```

Единственный параметр функции является указателем на структуру типа CHOOSEFONT.Эта структура, а так же сама функция ChooseFont определены в файле commdlg.h.Структура определена следующим образом:

```
typedef struct tagCHOOSEFONT
{
  DWORD        lStructSize;
  HWND         hwndOwner;
  HDC          hDC;
  LOGFONT FAR*   lpLogFont;
  int          iPointSize;
```

```
    DWORD          Flags;
    COLORREF       rgbColors;
    LPARAM         lCustData;
    UINT (CALLBACK* lpfnHook)(HWND, UINT, WPARAM, LPARAM);
    LPCSTR         lpTemplateName;
    HINSTANCE      hInstance;
    LPSTR          lpszStyle;
    UINT           nFontType;
    int            nSizeMin;
    int            nSizeMax;
} CHOOSEFONT;
typedef CHOOSEFONT FAR *LPCHOOSEFONT;
```

Перед вызовом функции ChooseFont вы должны проинициализировать нужные поля структуры CHOOSEFONT , записав в остальные поля нулевые значения.

Опишем название отдельных полей структуры CHOOSEFONT:

Поле	Описание
lStructSize	Размер структуры в байтах. Это поле необходимо заполнить перед вызовом функции ChooseFont
hwndOwner	Идентификатор окна, которому будет пренадлежать диалоговая панель, если в поле Flags не указан флаг CF_SHOWHELP, в это поле можно записать значение NULL.Поле заполняется до вызова функции ChooseFont
hDC	Идентификатор контекста отображения или информационного контекста для принтера.Если установлен флаг CF_PRINTERFONTS в списках появятся шрифты, доступные в данном контексте
lpLogFont	Указатель на структуру LOGFONT.Приложение может заполнить нужные поля в этой структуре перед вызовом фугкции ChooseFont. Если при этом будет установлен флаг CF_INITTOLOGFONTSTRUCT, выбранное значение будет использоваться в качестве начальной
iPointSize	Размер букв выбранного шрифта в десятых долях пункта. Содержимое этого поля устанавливается после возврата из функции ChooseFont
Flags	Флаги инициализации диалоговой панели. Можно использовать следующее значение : CF_APPLY - разрешается использование кнопки "Apply"; CF_ANSIONLY - в списке выбора появляются только шрифты в кодировке ANSI; CF_BOTH - в списке шрифтов появляются экранные и принтерные шрифты; CF_TTONLY - можно выбирать только масштабируемые шрифты True Type;

CF_EFFECTS - если указан этот флаг, с помощью диалоговой панели можно определять цвет букв, создавать подчеркнутые и перечеркнутые шрифты, в этом случае необходимо перед вызовом функции проинициализировать содержимое полей lfStrikeout, lfUnderline, rgbColors;
CF_ENABLEHOOK - разрешается использовать функцию фильтра, адрес которой указан в поле lpfnShok;
CF_ENABLETEMPLATE - разрешается использование шаблонга диалоговой панели, определяемого содержимым полей hInstance, lpTemplateName;
CF_ENABLETEMPLATEHANDLE - флаг указывает, что поле hInstance содержит идентификатор загружаемого шаблона диалоговой панели. Содержимое поля lpTemplateName игнорируется;
CF_FIXEDTITCHONLY-можно выбрать только шрифты с фиксированной шириной символов;
CF_FORCEFONTEXIST - выдается сообщение об ошибке, если пользователь пытается выбрать не существующий шрифт;
CF_INITTOLOGFONTSTRUCT - для инициализации диалоговой панели используется содержимое панели LOGFONT, адрес который передается через поле lpLogFont;
CF_LIMITSIZE - при выборе шрифта учитывается содержимое полей nSizeMin и nSizeMax;
CF_NOFACESEL- отменяется выбор в списке "Font";
CF_NOOEMFONTS- -нельзя выбирать векторные шрифты, этот флаг аналогичен флагу CF_NOVECTORFONTS;
CF_NOSIMULATIONS-запрещается эмуляция шрифтов;
CF_NOSIZESEL-отменяется выбор размера шрифта
CF_NOSTYLESEL-отменяется выбор стиля шрифта
CF_NOVECTORFONTS-нельзя выбирать векторные шрифты, этот флаг аналогичен флагу CF_NOOEMFONT;
CF_PRINTERFONTS-в списке появляются только такие шрифты , которые поддерживаются принтером ,контекст отображения для которого задан в поле hDC;
CF_SCALABLEONLY-можно выбирать только масштабируемые и векторные шрифты ;
CF_SCREENFONTS-можно выбирать только экранные шрифты;
CF_SHOWHELP-в диалоговой панели отображается кнопка "Help";
CF_USESTYLE-строка lpszStyle содержит указатель на буфер , который содержит строку описания стиля. Эта строка используется для инициализации списка

"Font Style" диалоговой панели "Font";
CF_WYSIWYG- можно выбирать только такие шрифты , которые доступны и для отображения на экране , и для печати на принтере. Если установлен этот флаг, следует также установить флаги CF_BOTH и CF_SCALABLEONLY

rgbColors Цвет символов шрифта , который будет выбран в меню "Colors" диалоговой панели "Font" сразу после отображения диалоговой панели. Должен использоваться флаг CF_EFFECTS. Поле заполняется до вызова функйии ChooseFont, после возврата из функции поле содержит значение выбранного цвета

lCustData Произвольные данные , передаваемые функции фильтром, определенным содержимым поля lpfnHook

lpfnHook Указатель на функцию фильтра , обрабатывающую сообщения , поступающие в диалоговую панель. Для работы с фильтром необходимо в поле Flags указать флаг CF_ENABLHOOK

lpTemplateName Строка , закрытая двоичным нулем , которая содержит идентификатор шаблона диалоговой панели. Для использования этого поля необходимо указать флаг CF_ENABLETEMPLATE

hInstance Идентификатор модуля , который содержит шаблон диалоговой панели в качестве ресурса. Поле используется только в тех случаях , когда в поле Flags указано значение CF_ENABLETEMPLATE и CF_ENABLETEMPLATEHANDLE
Поле заполняется до вызова функции ChoseFont

lpszStyle Указатель на буфер , содержащийй строку описывающую шрифт. Если указан флаг CF_USESTYLE, эта строка используется для инициализации списка "Font Style" .
Размер буфера должен быть меньше LF_FACESIZE байт

nFontType Тип выбираемого шрифта. Можно использовать одно из следующих значений:
SIMULATED_FONTTYPE - GDI может эмулировать этот шрифт;
PRINTER_FONTTYPE- принторный шрифт;
SCREEN_FONTTYPE-экранный шрифт;
BOLD_FONTTYPE-жирный шрифт, используется только для шрифтов True Type;
ITALIC_FONTTYPE-наклонный шрифт, используется только для шрифтов True Type;
REGULAR_FONTTYPE-не наклонный и не жирный шрифт, используется только для шрифтов True Type;

nSizeMin Минимальный размер шрифта, который можно выбрать.

Для использования этого поля необходимо установить флаг CF_LIMITSIZE

nSizeMax Максимальный размер шрифта, который можно выбрать. Для использования этого поля необходимо установить флаг CF_LIMITSIZE

Если пользователь выбрал шрифт , функция ChooseFont возвращает значение TRUE. Если пользователь отказался от выбора,нажав кнопку "Cancel" или клавишу <Esc> , возвращается щначение FALSE.

Часть вторая. Практикум по программированию.

ТЕМА:СОБЫТИЙНОЕ ПРОГРАММИРОВАНИЕ В МНОГОЗАДАЧНОЙ СИСТЕМЕ С ГРАФИЧЕСКИМ ИНТЕРФЕЙСОМ ПОЛЬЗОВАТЕЛЯ (C, C++ MS для Windows).

ПРОГРАММА № 1. СОЗДАНИЕ ШАБЛОНА ПРИКЛАДНОЙ ПРОГРАММЫ С ГЛАВНЫМ ОКНОМ,МЕНЮ, ПАНЕЛЬЮ ИНСТРУМЕНТОВ (TOOLBAR), СТРОКОЙ СТАТУСА (STATUSBAR) (BORLAND C++ 5.02, Windows API).

ЗАДАНИЕ:

Составить программу с главной фуекцией WinMain, оконной функцией, обрабатывающей основные системные сообщения WM_CREATE, WM_SIZE, WM_MOVE, WM_DESTROY, WM_LBUTTONDOWN(и другие сообщения от мыши), WM_CHAR, WM_KEYDOWN, WM_KEYUP, WM_PAINT, WM_TIMER, WM_COMMAND (и команды меню).

В функции окна обработчики сообщения выполнить с помощью оператора switch(код сообщения) - case WM_ :

В этой первой программе не будет прикладной задачи, а каждый обработчик сообщения должен будет вызвать диалоговое окно MessageBox с информацией, которую несет в себе обрабатываемое сообщение. Например, обработчик сообщения от мыши должен вывести, какая клавиша нажата (или отпущена) и координаты мышиного курсора, сообщение WM_SIZE - размеры новой клиентской области окна и т.д.

Для начала Вам предоставляется пример обработчиков, а остальные вы должны сделать самостоятельно.

МЕТОДИЧЕСКОЕ ПОСОБИЕ ПО СОСТАВЛЕНИЮ ПРОГРАММЫ.

Наша задача - создать шаблон программы для Windows, которая ничего не делает, но имеет многие необходимые компонеты для того, чтобы можно было наполнить их
конкретным содержанием в зависимости от того, чем должна заниматься прикладная
программа. В следующих программах мы будем вначале копировать этот шаблон,

постепенно заполняя пустые секции. Единственное, чем мы обеспечим наш шаблон с самого начала - это возможностью завершения программы.

1.Вызовите на выполнение программу IDE (bcw в командной строке или мышкой по иконке на рабочем столе).
2.Меню File - New - Proect - Browse в диалоговом окне и переход в свой рабочий каталог(пусть он называется группа\TWINCASE). Остальные пункты дилога - Application.exe, Win32, GUI, и убрать птичку OWL.
Введите придуманное Вами имя проекта в строке Target Name (пусть оно тоже twincase) и нажмите кнопку ОК.
3.В списке файлов проекта удалите файл определения модуля имя.def (будет исполь зован стандартный файл по умолчанию).
4.Правой кнопкой мыши в свернутом окне проекта вызовите диалог TargetExpert и выберите режим GUI(Graphics User Interface – графический интерфейс пользователя)
5.Откройте файл имя.cpp - получите пустое окно, которое Вам предстоит заполнить текстом Вашей прикладной программы.
6.Введите комплект директив препроцессора, приведенный ниже с поясняющими комментариями.
*/

Файл twincase.cpp

```
#define STRICT       //Просим компилятор осуществлять строгую проверку типов
#include <windows.h> //Включить файл интерфейса с библиотекой Windows API
#include <windowsx.h>//То же для связи с расширенной версией Windows API
#include <commctrl.h>//Интерфейс со стандартными органами управления
#include <afxres.h>  //Интерфейс с ресурсами MFC - пока он нам не нужен
                     //но в дальнейшем пригодится
/* В этом месте текста программы обычно размещают объявления кон-
стант,глобальных
переменных, которые должны быть доступны всем подпрограммам файла, и прото-
типы
функций. Стандартный набор констант состоит из имени класса окна и заголовка
окна, минимальный коплект глобальных переменных - структура WNDCLASS,
описывающая свойства класса окна (используется при регистрации в операционной
системе) и идентификатор (handle) типа HINSTANCE составляемой прикладной
программы (приложения)*/

// Имя класса окна
char const szClassName[]   = "TWindowClass";
// Заголовок окна
char const szWindowTitle[] =
"Шаблон Widows-программы, которая ничего не делает";
WNDCLASS wc;    // структура для регистрации класса окна
HINSTANCE hInst; //Идентификатор нашей программы
/* Так как наша программа - ничего не делающий шаблон, список прототипов
функций будет содержать только функцию начальной инициализации (в нее мы
```

вынесем заполнение полей структуры WNDCLASS, регистрацию класса окна и создание окна) и функцию главного окна, предназначенную для обработки сообщений Windows;
по крайней мере одно сообщение - о необходимости завершить ничего не делющую программу эта функция должна будет обработать. Ее имя произвольно, возвращаемый
тип - целое LRESULT, модификатор WINAPI или APIENTRY или CALLBACK (обратный вызов), а аргументы описаны ниже в комметариях.
*/
```
//Прототипы функций
//Начальной инициализации
BOOL InitApp(HINSTANCE);
//Главного окна
LRESULT CALLBACK  WndProc(HWND,//Идентификатор окна, для которого эта ф-
//ция  будет получать и обрабатывать сообщения
                                    UINT,//Код сообщения
                                    WPARAM,//Специфич для сообщения информа-
ция
                                    LPARAM);


/* К текстам инициализирующей и оконной функций мы вернемся ниже, а пока как обычно начнем составление программы с главной функции, которая получает управление при запуске программы, носит имя WinMain и имеет 4 получаемых из ОС
аргумента, типы которых прокомментированы в ее заголовке.
Совет: не набирайте ее заголовок, а выкопируйте из Help для Win32.
В теле главной функции мы вызовем на выполнение функцию начальной инициализации, библиотечную функцию создания окна CreateWindow, которая вернет нам handle окна, пару библиотечных функций вывода окна  ShowWindow и UpdateWindow и создадим для нашей программы цикл обработки сообщений, получаемых от Windows.
*/

////////////////////////////////////////////////////// Функция WinMain//////////////////////////////////////
int WINAPI WinMain(
   HINSTANCE hInstance,      //handle запускаемой программы
   HINSTANCE hPrevInstance,//handle ранее запущенной копии этой же программы-
                             //в Win32 всегда NULL и не используется (остался
                             //от Win16)
   LPSTR lpCmdLine,          //Указатель на командную строку
   int nCmdShow              // Код состояния окна при запуске программы;
//возможные значения - SW_HIDE, SW_MINIMIZE, SW_RESTORE,
//SW_SHOWNORMAL, SW_SHOW,  SW_SHOWMAXIMIZED,
//SW_SHOWMINIMIZED, SW_SHOWMINNOACTIVE, SW_SHOWNA,
//SW_SHOWNOACTIVATE, SW_SHOWNORMAL.
```

```
  )
{
 MSG  msg;  // структура для работы с сообщениями
 HWND hwnd; // идентификатор главного окна приложения
 //Сохраним в глобальной переменной идентификатор приложения
 hInst=hInstance;
 // Инициализируем приложение
 if(!InitApp(hInstance)) return FALSE; //Если не удалось - выходим
 // Инициализируем библиотеку стандартных органов управления
 InitCommonControls();
 // После успешной инициализации приложения создаем главное окно
 hwnd = CreateWindow(
   szClassName,              // имя класса окна
   szWindowTitle,            // заголовок окна
   WS_OVERLAPPEDWINDOW, // стиль окна перекрываемое (другие возможные
значения

                                        //посмотрите в Help-системе)
   CW_USEDEFAULT,      //горизонтальная позиция окна (у нас - по умолчанию)
   CW_USEDEFAULT,      //вертикальная позиция окна
   CW_USEDEFAULT,      //ширина окна
   CW_USEDEFAULT,      //высота окна
   0,                  //идентификатор родительского окна
   0,                  // идентификатор меню или дочернего окна
   hInst,              //идентификатор программы
   NULL);              //указатель на дополнительные данные
 // Если создать окно не удалось, завершаем приложение
 if(!hwnd)    return FALSE;
 ShowWindow(hwnd, nCmdShow); // Рисуем главное окно
 UpdateWindow(hwnd);
//Запускаем цикл обработки сообщений - его простейший вариант
 while(GetMessage(      //Функция выборки сообщения из очереди
            &msg,//Адрес структуры с сообщением
            0,      //Идентификатор окна, котрому надо переадресовать сообщ
            0,      //Начало диапазона отбираемых (фильтруемых) сообщений
            0))     //Конец диапазона отбираемых (фильтруемых) сообщений
 {
  TranslateMessage(&msg);//Транслирует виртуальные коды клавиш в символьные
  DispatchMessage(&msg);//Направляет сообщение нужной оконной функции
 }
 //Более сложный вариант использует ф-цию PeekMessage, определяющую наличие
 //сообщений из очереди без извлечения. Это дает возможность выполнять
 //несрочную работу при отсутствии в очереди сообщений. Этот фрагмент мы
 //приводим в комментарии.
 /*
  for(;;)//Бесконечный цикл до получения WM_QUIT
```

```
{
if(PeekMessage(&msg, NULL, 0, 0,PM_REMOVE))//Если в очереди есть сообщения
{
if(msg.message==WM_QUIT) break;//Если это конец работы
TranslateMessage(&msg);
DispatchMessage(&msg);
}
else //Если сообщений нет
{
//Тут можем делать что-то  несрочное
}
}
  */
  return msg.wParam;
}
/*
```

Мы собираемся оснастить наше окно строкой главного меню с выпадающими подменю. Для начала сделаем это простейшим способом - щелкнем в файле проекта по имени
файла ресурсов имя.кс - в гланом меню IDE появится пункт Resource, а в нем - позиция New, использовав которую вы получите список стандартных ресурсов, включающий и меню. Выберите MENU - получите стандартное меню с позициями File, Edit, Window, Help. Удалите лишнее, добавьте недостающее.
При выходе вам предложат сохранить ресурс - согласитесь.
Из созданного Resource Workshop-ом файла twincase.rc выкопируйте строку
#define IDM_MENU11 и поместите ее в файл resource.h, сам этот файл включите директивой #include "resource.h" в файл twindow.cpp, как показано ниже.
В файле имя.rc вы найдете числовые коды строк меню - работать с ними неудобно, поэтому заголовочный файл twindow.h пополним символьными значениями этих констант - их мы будем использовать в оконной функции как команды меню.
Подошла очередь составить функцию инициализации - в ней может быть и специфичная
для приложения часть, но пока мы ограничимся стандартной задачей заполнения полей структуры WNDCLASS и ее регистрацией в ОС.
СОВЕТ: не набирайте список полей , а скопируйте из Help, сотрите типы и допишите
через точку имя переменной - в нашем случае wc., выполните присвоения.
Возможные варианты значений полей WNDCLASS смотрите в Help

```
*/
// Функция InitApp  Выполняет регистрацию класса окна
#include "resource.h"
#include "twincase.h"
BOOL InitApp(HINSTANCE hInstance)
{
  WNDCLASS wc;    // структура для регистрации
  // Преверяем, не было ли это приложение запущено ранее
```

```
 HWND hWnd = FindWindow(szClassName, NULL);//Ф-ция поиска копии
 if(hWnd)
 {
  if(IsIconic(hWnd))//Если копия найдена и свернута в иконку
        ShowWindow(hWnd, SW_RESTORE);//Восстанавливаем
     SetForegroundWindow(hWnd);   //И выводим на передний план
  return FALSE;              //И выходим из процедуры инициализации
 }
 //Если же программа запускается впервые-заполняем поля оконной стуктуры
 wc.lpszMenuName  = MAKEINTRESOURCE(IDM_MENU1);//Имя меню
 wc.style         = CS_HREDRAW | CS_VREDRAW;//Стиль класса окна
 wc.lpfnWndProc   = (WNDPROC) WndProc;      //Указатель на оконную функцию
 wc.cbClsExtra    = 0;                     //Дополнит данные для класса окна
 wc.cbWndExtra    = 0;                      //Дополнит данные для окна
 wc.hInstance     = hInstance;              //Идентификатор программы
 wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);//Иконка для программы
 wc.hCursor       = LoadCursor(NULL, IDC_ARROW);    //Курсор мышки - стрелка
 wc.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);//Кисть для фона
 wc.lpszClassName = (LPSTR)szClassName;     //Имя класса окна
 return RegisterClass(&wc);           //Библиот ф-ция регистрации в ОС
}
```

/*Теперь - текст оконной функции - главного оконного работяги, обработчика всех направляемых окну сообщений. Эта функция состоит в основном из операторов switch - case, в теле которых осуществляется вызов необходимых функций обработки соответствующих сообщений. Пока большинство секций case будет содержать лишь оператор return 0; - этим мы берем обработку сообщения на себя, но ничего не делаем сами и не даем сделать что-либо ОС по умолчанию*/

```
//Пусть 2 пары переменных хранят размеры клиентской области окна и его позицию
int cwClient,chClient,xPos,yPos;
LRESULT CALLBACK WndProc( HWND hWnd,//Идентификатор окна, для которого
//эта ф-ция будет получать и обрабатывать сообщения
                  UINT msg, //Код сообщения
                  WPARAM wParam,//Специфическая для сообщения информация
                  LPARAM lParam)
{
//Оконная ф-ция всегда содержит переключатель switch - case по коду
//вызвавшего ее сообщения. Секции case можно непосредственно заполнить
//операторами выполнения требуемой работы
//Внимание - обработчик WM_CREATE должен вернуть TRUE, иначе окно не будет
//создано.
switch (msg)
  {
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
//В самом начале обеспечим нормальный выход из программы по нажатию кнопоч-
ки
//в правом верхнем углу окна или Alt-F4.В очередь приложения пойдет сообщение
// WM_QUIT. Это приведет к завершению цикла обработки сообщений в WinMain.
  case WM_DESTROY:
{
MessageBox(NULL, "Пришло сообщение WM_DESTROY",
"Шаблон Windows - программы",MB_OK);
PostQuitMessage(0); //Посылает сообщение WM_QUIT в очередь
return 0;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//Если понадобится что-либо сделать при создании окна
  case WM_CREATE:
   { MessageBox(NULL, "Пришло сообщение WM_CREATE",
    "Шаблон Windows - программы",MB_OK);
    //Создадим 4-секундный таймер
    SetTimer(hWnd,1,8000,NULL);
     return 0;
   }
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//При изменении размеров окна надо обычно корректировать  размеры изображения
// и перерисовывать  - поэтому берем на себя обработку WM_SIZE, WM_MOVE
case WM_SIZE:
{
//Из младшего и старшего слов lParam возьмем новые размеры окна
cwClient=LOWORD(lParam);
chClient=HIWORD(lParam);
char s[128];
wsprintf(s,"Пришло сообщение WM_SIZE - cwClient= %d, chClient=%d",
cwClient,chClient);
MessageBox(NULL,s,"Шаблон Windows - программы",MB_OK);
return 0;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//При изменении позиции окна его надо перерисовывать
case WM_MOVE:
{
//Из младшего и старшего слов lParam возьмем новые координаты окна
xPos=LOWORD(lParam);
yPos=HIWORD(lParam);
//Сформируем строку сообщения и выведем ее
char s[128];
wsprintf(s,"Пришло сообщение WM_MOVE - xPos= %d, yPos=%d",
xPos,yPos);
MessageBox(NULL,s,"Шаблон Windows - программы",MB_OK);
```

```
return 0;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//Если понадобится рисовать
PAINTSTRUCT ps;
case WM_PAINT:
{
HDC hdc=BeginPaint(hWnd,&ps);
//Тут можно рисовать или выводить текст
MessageBox(NULL,"Пришло сообщение WM_PAINT","Шаблон Windows - програм-
мы",MB_OK);
EndPaint(hWnd,&ps);
return 0;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//Обработка  клавиш
case WM_KEYDOWN:
{
char s[256];
wsprintf(s,"Пришло сообщение WM_KEYDOWN: нажата клавиша  %c",wParam);
MessageBox(NULL,s,"Шаблон Windows - программыа",MB_OK);
return 0;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//Если надо будет обработать "тики" таймера
case WM_TIMER:
{MessageBox(NULL,"Пришло сообщение таймера","Шаблон Windows - програм-
мы",MB_OK);
return 0;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//Обработка команд
case WM_COMMAND:
{switch(wParam)
{
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
case CM_NEW:
{     MessageBox(NULL, "Поступила команда CM_NEW на выход с вещами",
     "Сообщение WM_COMMAND", MB_OK);
     return 0;     }
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
  case CM_EXIT:
{    KillTimer(hWnd,1);
     DestroyWindow(hWnd);//Разрушаем оконные структуры,очищаем очередь
                                 //сообщений
     PostQuitMessage(0);
```

```
    return 0;    }
}//switch wparam
}//case WM_COMMAND
//Обработку всех других сообщений, не упомянутых у нас в case, поручим Windows
default:return(DefWindowProc(hWnd, msg, wParam, lParam));
}//switch msg
}
```

ПРОГРАММА № 2. СОЗДАНИЕ ШАБЛОНА ПРИКЛАДНОЙ ПРОГРАММЫ С ГЛАВНЫМ ОКНОМ,МЕНЮ (BORLAND C++ 5, Windows API).

ЗАДАНИЕ:

Эта программа будет полным дублем предыдущей, но обработка сообщений будет осуществляться с помощью специальных макрокоманд разборки сообщений - Message Cracers, оформленных по следующему прототипу:

HANDLE_MSG (hWnd,WM_сообщение,имя_подпрограммы_обработчика).

МЕТОДИЧЕСКОЕ ПОСОБИЕ ПО СОСТАВЛЕНИЮ ПРОГРАММЫ.

В качестве пособия для вас оформлены обработчики некоторых сообщений и команд
через макрокоманды разборки сообщений. Остальные вам предлагается выполнить самостоятельно по приведенным образцам.

```
*/
#define STRICT       //Просим коппилятор осуществлять строгую проверку типов
#include <windows.h> //Включить файл интерфейса с библиотекой Windows API
#include <windowsx.h>//То же для связи с расширенной версией Windows API
#include <commctrl.h>//Интерфейс со стандартными органами управления
#include <\mfc\afxres.h>//Интерфейс с ресурсами MFC - пока он нам не нужен
                  //но в дальнейшем пригодится
#include "resource.h"
#include "twincrac.h"
// Имя класса окна
char const szClassName[]   = "TWindowClass";
// Заголовок окна
char const szWindowTitle[] =
"Шаблон Widows-программы, которая ничего не делает";
WNDCLASS wc;    // структура для регистрации класса окна
HINSTANCE hInst; //Идентификатор нашей программы
//Прототипы функций
//Начальной инициализации
BOOL InitApp(HINSTANCE);
//Главного окна
LRESULT CALLBACK  WndProc(HWND,//Идентификатор окна, для которого эта ф-
//ция будет получать и обрабатывать сообщения
                   UINT,//Код сообщения
                   WPARAM,//Специфическая для этого сообщения информация
                   LPARAM);
```

```
//////////////////////////////////////// Функция WinMain//////////////////////////////////////
int WINAPI WinMain(
   HINSTANCE hInstance,   //handle запускаемой программы
   HINSTANCE hPrevInstance,//handle ранее запущенной копии этой же программы-
//в Win32 всегда NULL и не используется (остался от Win16)
   LPSTR lpCmdLine,        //Указатель на командную строку
   int nCmdShow            // Код состояния окна при запуске программы;
//возможные значения - SW_HIDE, SW_MINIMIZE, SW_RESTORE,
//SW_SHOWNORMAL, SW_SHOW, SW_SHOWMAXIMIZED,
SW_SHOWMINIMIZED, //SW_SHOWMINNOACTIVE,SW_SHOWNA,
SW_SHOWNOACTIVATE, //SW_SHOWNORMAL.
  )
{
 MSG  msg;      // структура для работы с сообщениями
 HWND hwnd; // идентификатор главного окна приложения
 //Сохраним в глобальной переменной идентификатор приложения
 hInst=hInstance;
 // Инициализируем приложение
 if(!InitApp(hInstance)) return FALSE; //Если не удалось - выходим
 // Инициализируем библиотеку стандартных органов управления
 InitCommonControls();
 // После успешной инициализации приложения создаем главное окно
 hwnd = CreateWindow(
   szClassName,         // имя класса окна
   szWindowTitle,       // заголовок окна
   WS_OVERLAPPEDWINDOW, // стиль окна перекрываемое (другие возможные
зна            //чения посмотрите в Help-системе)
   CW_USEDEFAULT,       //горизонтальная позиция окна (у нас - по умолчанию)
   CW_USEDEFAULT,       //вертикальная позиция окна
   CW_USEDEFAULT,       //ширина окна
   CW_USEDEFAULT,       //высота окна
   0,                   //идентификатор родительского окна
   0,                   // идентификатор меню или дочернего окна
   hInst,               //идентификатор программы
   NULL);               //указатель на дополнительные данные
 // Если создать окно не удалось, завершаем приложение
 if(!hwnd)    return FALSE;
 // Рисуем главное окно
 ShowWindow(hwnd, nCmdShow);
 UpdateWindow(hwnd);
//Запускаем цикл обработки сообщений - его простейший вариант
 while(GetMessage(      //Функция выборки сообщения из очереди
            &msg,        //Адрес структуры с сообщением
            0,   //Идентификатор окна, котрому надо переадресовать сообщ
            0,   //Начало диапазона отбираемых (фильтруемых) сообщений
            0))  //Конец диапазона отбираемых (фильтруемых) сообщений
```

```
  {
   TranslateMessage(&msg);//Транслирует виртуальные коды клавиш в символьные
   DispatchMessage(&msg);//Направляет сообщение нужной оконной функции
  }
  //Более сложный вариант использует ф-цию PeekMessage, определяющую наличие
  //сообщений из очереди без извлечения. Это дает возможность выполнять
  //несрочную работу при отсутствии в очереди сообщений. Этот фрагмент мы
  //приводим в комментарии.
  /*
   for(;;)//Бесконечный цикл до получения WM_QUIT
{
if(PeekMessage(&msg, NULL, 0, 0,PM_REMOVE))//Если в очереди есть сообщения
{
if(msg.message==WM_QUIT) break;//Если это конец работы
TranslateMessage(&msg);
DispatchMessage(&msg);
}
else //Если сообщений нет
{
//Тут можем делать что-то  несрочное
}
}
  */
  return msg.wParam;
}
////////////////////////// Функция InitApp - выполняет регистрацию класса окна//////////////////
BOOL   InitApp(HINSTANCE hInstance)
{  WNDCLASS wc;    // структура для регистрации
  // Преверяем, не было ли это приложение запущено ранее
  HWND hWnd = FindWindow(szClassName, NULL);//Ф-ция поиска копии
  if(hWnd)
  {
   if(IsIconic(hWnd))//Если копия найдена и свернута в иконку
         ShowWindow(hWnd, SW_RESTORE);//Восстанавливаем
SetForegroundWindow(hWnd);                    //И выводим на передний план
   return FALSE;                                //И выходим из процедуры инициализа-
ции
  }
  //Если же программа запускается впервые-заполняем поля оконной стуктуры
  wc.lpszMenuName  = MAKEINTRESOURCE(IDM_MENU1);//Имя меню
  wc.style         = CS_HREDRAW | CS_VREDRAW;//Стиль класса окна
  wc.lpfnWndProc   = (WNDPROC) WndProc;         //Указатель на оконную функцию
  wc.cbClsExtra    = 0;                  //Дополнит данные для класса окна
  wc.cbWndExtra    = 0;                  //Дополнит данные для окна
  wc.hInstance     = hInstance;          //Идентификатор программы
  wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);//Иконка для программы
```

```
  wc.hCursor       = LoadCursor(NULL, IDC_ARROW);    //Курсор мышки - стрелка
  wc.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);//Кисть для фона
  wc.lpszClassName = (LPSTR)szClassName;     //Имя класса окна
  return RegisterClass(&wc);           //Библиот ф-ция регистрации в ОС
 }
//////////////////////////////////////////////Оконная функция//////////////////////////////////////////////
//Пусть 2 пары переменных хранят размеры клиентской области окна и его позицию
int cwClient,chClient,xPos,yPos;
LRESULT CALLBACK WndProc( HWND hWnd,//Идентификатор окна, для которого
//эта ф-ция  будет получать и обрабатывать сообщения
                  UINT msg,             //Код сообщения
                  WPARAM wParam,//Специфическая для сообщения информация
                  LPARAM lParam)
{
//Оконная ф-ция всегда содержит переключатель switch - case по коду
//вызвавшего ее сообщения. Секции case можно непосредственно заполнить
//операторами выполнения требуемой работы, а можно использовать имеющиеся
//в windowsx.h макрокоманды разборки сообщений (message-cracers)
//          HANDLE_MSG(hWnd,WM_сообщение,имя_обработчика),
//Использование макрокоманд сделает короче операторы case
switch (msg)
  {
//Использование макрокоманд разборки некоторых сообщений
HANDLE_MSG(hWnd, WM_CREATE,WndProc_OnCreate);
HANDLE_MSG(hWnd, WM_DESTROY,WndProc_OnDestroy);
HANDLE_MSG(hWnd, WM_SIZE,WndProc_OnSize);
HANDLE_MSG(hWnd, WM_MOVE,WndProc_OnMove);
HANDLE_MSG(hWnd, WM_CHAR,WndProc_OnChar);
HANDLE_MSG(hWnd, WM_LBUTTONDOWN,WndProc_OnLbuttonDown);
HANDLE_MSG(hWnd, WM_COMMAND,WndProc_OnCommand);//Разбрки команд
//Обработку всех других сообщений, не упомянутых у нас в case, поручим Windows
default:return(DefWindowProc(hWnd, msg, wParam, lParam));
   }
}
//////////////////////////////////////Обработчик WM_CHAR//////////////////////////////////////////////
BOOL WndProc_OnChar(HWND hwnd, TCHAR ch, int rpt)
{
MessageBox(NULL, "Пришло сообщение WM_CHAR",
"Шаблон Windows - программы", MB_OK);
}
//////////////////////////////////////Обработчик WM_LBUTTONDOWN//////////////////////////////
BOOL WndProc_OnLbuttonDown(HWND hwnd, BOOL fDblClk, int x, int y, UINT
keyFlags)
{
MessageBox(NULL, "Пришло сообщение WM_LBUTTONDOWN",
"Шаблон Windows - программы", MB_OK);
```

```
}
////////////////////////////////////Обработчик WM_CREATE//////////////////////////////////////
//Действия выполняемые при создании окна- обработка WM_CREATE
BOOL WndProc_OnCreate(HWND hWnd, LPCREATESTRUCT lpCreateStruct)
{return TRUE;}
////////////////////////////////////Обработчик WM_DESTROY/////////////////////////////////////
void WndProc_OnDestroy(HWND hWnd)
{
MessageBox(NULL, "Пришло сообщение WM_DESTROY", "Шаблон Windows - про-
граммы",MB_OK);
PostQuitMessage(0);
}
//////////////////////////////////////Обработчик команд/////////////////////////////////////////
void WndProc_OnCommand(HWND hWnd, int id,
  HWND hwndCtl, UINT codeNotify)
{  switch (id)
  {
case CM_EXIT: PostQuitMessage(0); break;
case CM_NEW: MessageBox(NULL, "Команда CM_NEW создания нового файла",
"Шаблон Windows - программы", MB_OK); break;
case CM_OPEN:MessageBox(NULL, "Команда CM_OPEN открытияия  файла",
"Шаблон Windows - программы", MB_OK); break;
case CM_SAVE: MessageBox(NULL, "Команда CM_SAVE сохранения  файла",
"Шаблон Windows - программы", MB_OK); break;
case CM_SAVEAS: MessageBox(NULL, "Команда CM_SAVEAS сохранения  файла",
"Шаблон Windows - программы", MB_OK); break;
case CM_SELECTFONT:{
MessageBox(NULL, "Команда CM_SELECTFONNT выбора шрифта", "Шаблон
Windows - программы", MB_OK);
break;
}
default:break;
}}
////////////////////////////////////Обработчик WM_SIZE//////////////////////////////////////////
//Обработка сообщения об изменении размеров окна w,h-новые размеры окна
void WndProc_OnSize(HWND hwnd, UINT state, int w, int h)
{
MessageBox(NULL, "Пришло сообщение WM_SIZE", "Шаблон Windows - програм-
мы",
 MB_OK);
}
//Обработка сообщения об изменении позиции окна  cx,cy-новые координаты окна
void WndProc_OnMove(HWND hwnd, int cx, int cy)
{
MessageBox(NULL, "Пришло сообщение WM_MOVE", "Шаблон Windows - про-
граммы", MB_OK);}
```

ПРОГРАММА № 3. СОЗДАНИЕ ШАБЛОНА ПРИКЛАДНОЙ ПРОГРАММЫ С ГЛАВНЫМ ОКНОМ, МЕНЮ, ПАНЕЛЬЮ ИНСТРУМЕНТОВ (TOOLBAR), СТРОКОЙ СТАТУСА (STATUSBAR) (BORLAND C++ 5, Windows API).
ЗАДАНИЕ:
1)Дополнить предыдущую программу строкой статуса и многокнопочной панелью инструментов, которая будет дублировать позиции меню.
Как и в предыдущих заданиях - максимально расширить диапазон орабатываемых сообщений. Увеличьте количество кнопок в Toolbar по сравнению с приведенным в образце.
2)В приведенном образце Statusbar не разбит на области и в нее не заносятся графические изображения. Вам предлагается дополнить программу этими элементами,
используя в качестве учебного материала 22-й том БСП Фроловых (он есть на диске вашего компьютера).
МЕТОДИЧЕСКОЕ ПОСОБИЕ ПО СОСТАВЛЕНИЮ ПРОГРАММЫ.
1)По созданию органа управления Toolbr
Вначале надо подготовить графическое изображение кнопок - для этого проще всего взять из MS Visual C++ файл toolbar.bmp и включить его в ресурсы записью в *.rc файл оператора

```
IDB_TBBITMAP  BITMAP  DISCARDABLE    "toolbar.bmp"
```

Далее придется создать массив описателей кнопок TBBUTTON - этот массив подготовлен для вас в файле twinbar.h - разберитесь в его содержании.
Следующий этап - создание окна Toolbar - проще всего с помощью функции CreateToolbarEx - ее использование с комментариями для каждого аргумента вы найдете в обработчике сообщения WM_CREATE WndProc_OnCreate.
Перед вызовом этой функции следует вызвать функцию инициализации библиотеки стандартных органов управления InitCommonControls без аргументов - это можно сделать еще в подпрограмме инициализации приложения или в WinMain.
Орган Toolbar посылает в родительское окно сообщения WM_COMMAND и WM_NOTIFY
(wParam - идентификатор Toolbar, lParam - указатель на структуру TBNOTIFY, содержащую первым полем структуру NMHDR с 3-мя полями - идентификатор окна,
органа управления и код сообщения. Другие поля - номер кнопки, структура описания кнопки, длина тектовой строки, ассоциированной с кнопкой и адрес этой строки).
Пример обработки сообщения WM_NOTIFY приведен ниже м комментариями.
2)По созданию органа Statusbar
Создать строку статуса проще всего с помощью универсальной функции создания окон CreateWindowEx или с помощью специальной функции CreateStatusWindow при обработке сообщения WM_CREATE.
Для записи текста или графики в строку статуса используют посылаемое ей сообщение SB_TEXT с помощью например функции SendMessage.
*/

Файл twinbar.cpp

```
#define STRICT       //Просим коппилятор осуществлять строгую проверку типов
#include <windows.h> //Включить файл интерфейса с библиотекой Windows API
#include <windowsx.h>//То же для связи с расширенной версией Windows API
#include <commctrl.h>//Интерфейс со стандартными органами управления
#include <\mfc\afxres.h>//Интерфейс с ресурсами MFC
#include "resource.h"
#include "twinbar.h"
// Имя класса окна
char const szClassName[]   = "TWindowClass";
// Заголовок окна
char const szWindowTitle[] ="Шаблон Widows-программы, которая ничего не дела-
ет";
WNDCLASS wc;     // структура для регистрации класса окна
HINSTANCE hInst; //Идентификатор нашей программы
//Прототипы функций
//Начальной инициализации
BOOL InitApp(HINSTANCE);
//Главного окна
LRESULT CALLBACK  WndProc(HWND,
                  UINT,       //Код сообщения
                  WPARAM,//Специфическая для этого сообщения информация
                  LPARAM);

//////////////////////////////////////// Функция WinMain//////////////////////////////////////
int WINAPI WinMain(
   HINSTANCE hInstance,      //handle запускаемой программы
   HINSTANCE hPrevInstance,
   LPSTR lpCmdLine,          //Указатель на командную строку
   int nCmdShow              // Код состояния окна при запуске программы;
)
{
 MSG  msg;  // структура для работы с сообщениями
 HWND hwnd; // идентификатор главного окна приложения
 //Сохраним в глобальной переменной идентификатор приложения
 hInst=hInstance;
 // Инициализируем приложение
 if(!InitApp(hInstance)) return FALSE; //Если не удалось - выходим
 // Инициализируем библиотеку стандартных органов управления
 InitCommonControls();
 // После успешной инициализации приложения создаем главное окно
 hwnd = CreateWindow(
   szClassName,         // имя класса окна
   szWindowTitle,       // заголовок окна
   WS_OVERLAPPEDWINDOW, // стиль окна
   CW_USEDEFAULT,       //горизонтальная позиция окна (у нас - по умолчанию)
   CW_USEDEFAULT,       //вертикальная позиция окна
```

```
   CW_USEDEFAULT,      //ширина окна
   CW_USEDEFAULT,      //высота окна
   0,                  //идентификатор родительского окна
   0,                  // идентификатор меню или дочернего окна
   hInst,              //идентификатор программы
   NULL);              //указатель на дополнительные данные
  // Если создать окно не удалось, завершаем приложение
  if(!hwnd)    return FALSE;
  // Рисуем главное окно
  ShowWindow(hwnd, nCmdShow);
  UpdateWindow(hwnd);
 //Запускаем цикл обработки сообщений - его простейший вариант
  while(GetMessage(      //Функция выборки сообщения из очереди
              &msg,//Адрес структуры с сообщением
              0,   //Идентификатор окна, котрому надо переадресовать сообщ
              0,   //Начало диапазона отбираемых (фильтруемых) сообщений
              0))  //Конец диапазона отбираемых (фильтруемых) сообщений
  {
    TranslateMessage(&msg);//Транслирует виртуальные коды клавиш в символьные
    DispatchMessage(&msg);//Направляет сообщение нужной оконной функции
  }
  //Более сложный вариант использует ф-цию PeekMessage, определяющую наличие
  //сообщений из очереди без извлечения. Это дает возможность выполнять
  //несрочную работу при отсутствии в очереди сообщений. Этот фрагмент мы
  //приводим в комментарии.
  /*
    for(;;)//Бесконечный цикл до получения WM_QUIT
{
if(PeekMessage(&msg, NULL, 0, 0,PM_REMOVE))//Если в очереди есть сообщения
{
if(msg.message==WM_QUIT) break;//Если это конец работы
TranslateMessage(&msg);
DispatchMessage(&msg);
}
else //Если сообщений нет
{
//Тут можем делать что-то  несрочное
}
}
  */
  return msg.wParam;
}
//////////////////////////// Функция InitApp Выполняет регистрацию класса окна//////////
BOOL  InitApp(HINSTANCE hInstance)
{
  WNDCLASS wc;    // структура для регистрации
```

```
 // Преверяем, не было ли это приложение запущено ранее
 HWND hWnd = FindWindow(szClassName, NULL);//Ф-ция поиска копии
 if(hWnd)
 {   if(IsIconic(hWnd))//Если копия найдена и свернута в иконку
     ShowWindow(hWnd, SW_RESTORE);//Восстанавливаем
     SetForegroundWindow(hWnd);   //И выводим на передний план
     return FALSE;              //И выходим из процедуры инициализации
 }
 //Если же программа запускается впервые-заполняем поля оконной стуктуры
 wc.lpszMenuName  = MAKEINTRESOURCE(IDM_MENU1);//Имя меню
 wc.style         = CS_HREDRAW | CS_VREDRAW;//Стиль класса окна
 wc.lpfnWndProc   = (WNDPROC) WndProc;      //Указатель на оконную функцию
 wc.cbClsExtra    = 0;                    //Дополнит данные для класса окна
 wc.cbWndExtra    = 0;                     //Дополнит данные для окна
 wc.hInstance     = hInstance;             //Идентификатор программы
 wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);//Иконка для программы
 wc.hCursor       = LoadCursor(NULL, IDC_ARROW);    //Курсор мышки - стрелка
 wc.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);//Кисть для фона
 wc.lpszClassName = (LPSTR)szClassName;     //Имя класса окна
 return RegisterClass(&wc);           //Библиот ф-ция регистрации в ОС
}

//Пусть 2 пары переменных хранят размеры клиентской области окна и его позицию
int cwClient,chClient,xPos,yPos;
LRESULT CALLBACK WndProc( HWND hWnd,//Идентификатор окна, для которого
//эта ф-ция будет получать и обрабатывать сообщения
                  UINT msg,             //Код сообщения
                  WPARAM wParam,//Специфическая для сообщения информация
                  LPARAM lParam)
{
switch (msg)
  {//Использование макрокоманд разборки некоторых сообщений
HANDLE_MSG(hWnd, WM_CREATE,WndProc_OnCreate);
HANDLE_MSG(hWnd, WM_DESTROY,WndProc_OnDestroy);
HANDLE_MSG(hWnd, WM_SIZE,WndProc_OnSize);
HANDLE_MSG(hWnd, WM_MOVE,WndProc_OnMove);
HANDLE_MSG(hWnd, WM_CHAR,WndProc_OnChar);
HANDLE_MSG(hWnd, WM_LBUTTONDOWN,WndProc_OnLbuttonDown);
HANDLE_MSG(hWnd, WM_NOTIFY,WndProc_OnNotify);
HANDLE_MSG(hWnd, WM_MENUSELECT,WndProc_OnMenuSelect);
HANDLE_MSG(hWnd, WM_COMMAND,WndProc_OnCommand);//Разбрки команд
//Обработку всех других сообщений, не упомянутых у нас в case, поручим Windows
default:return(DefWindowProc(hWnd, msg, wParam, lParam));
   }}
//////////////////////////////Обработчик WM_CHAR//////////////////////////////////////////////
BOOL WndProc_OnChar(HWND hwnd, TCHAR ch, int rpt)
```

```
{
MessageBox(NULL, "Пришло сообщение WM_CHAR",
"Шаблон Windows - программы", MB_OK);
}
//Обработчик WM_LBUTTONDOWN
BOOL WndProc_OnLbuttonDown(HWND hwnd, BOOL fDblClk, int x, int y, UINT
keyFlags)
{MessageBox(NULL, "Пришло сообщение WM_LBUTTONDOWN",
"Шаблон Windows - программы", MB_OK);}
//Действия выполняемые при создании окна- обработка WM_CREATE
BOOL WndProc_OnCreate(HWND hWnd, LPCREATESTRUCT lpCreateStruct)
{// Создаем Toolbar
hwndTb = CreateToolbarEx(hWnd,WS_CHILD | WS_BORDER | WS_VISIBLE |
TBSTYLE_TOOLTIPS | CCS_ADJUSTABLE,//стиль органа Toolbar
   IDT_TOOLBAR,          // идентификатор органа Toolbar
   5,                    // количество пиктограмм
   hInst,                // идентификатор приложения
   IDB_TBBITMAP,         // идентификатор битового изображения
                         //   кнопок
   (LPCTBBUTTON)&tbButtons,// адрес описания кнопок-см. в файле twindow.h
   6,                    // количество кнопок
   16,16,                // ширина и высота кнопок
   16,16,                // ширина и высота пиктограмм
   sizeof(TBBUTTON));    // размер структуры в байтах
   if(hwndTb == NULL)    return FALSE;
//Создаем строку -окно статуса
    hwndSb = CreateWindowEx(
   0L,                          // расширенный стиль окна
   STATUSCLASSNAME,             // класс окна для Statusbar
   "",                          // заголовок окна отсутствует
   WS_CHILD | WS_BORDER |  // стиль окна - дочернее, с рамкой
   WS_VISIBLE | SBARS_SIZEGRIP,
   0, 0, 0, 0,           // координаты, ширина, высота
   hWnd,                 // идентификатор родительского окна
   (HMENU)IDS_STATUSBAR,   // идентификатор Statusbar
   hInst,                // идентификатор приложения
   NULL );               // доп. данные для окна
   if(hwndSb == NULL)    return FALSE;
   return TRUE;}
////////////////////////////////////Обработчик WM_DESTROY/////////////////////////////////////
void WndProc_OnDestroy(HWND hWnd)
{
MessageBox(NULL, "Пришло сообщение WM_DESTROY", "Шаблон Windows - про-
граммы",MB_OK);
PostQuitMessage(0);}
////////////////////////////////////Обработчик команд/////////////////////////////////////////////
```

```
void WndProc_OnCommand(HWND hWnd, int id,
  HWND hwndCtl, UINT codeNotify)
{  switch (id)
  {
case CM_EXIT: PostQuitMessage(0); break;
case CM_NEW: MessageBox(NULL, "Команда CM_NEW создания нового файла",
"Шаблон Windows - программы", MB_OK); break;
case CM_OPEN:MessageBox(NULL, "Команда CM_OPEN открытияия  файла",
"Шаблон Windows - программы", MB_OK); break;
case CM_SAVE: MessageBox(NULL, "Команда CM_SAVE сохранения  файла",
"Шаблон Windows - программы", MB_OK); break;
case CM_SAVEAS: MessageBox(NULL, "Команда CM_SAVEAS сохранения  файла",
"Шаблон Windows - программы", MB_OK); break;
case CM_SELECTFONT:{
MessageBox(NULL, "Команда CM_SELECTFONNT выбора шрифта", "Шаблон
Windows - программы", MB_OK);
break;
}
default:break;
  }}

////////////////////////////////////Обработчик WM_NOTIFY/////////////////////////////////////////
LRESULT WndProc_OnNotify(HWND hWnd, int idFrom, NMHDR* pnmhdr)
{
LPTOOLTIPTEXT lpToolTipText;
LPTBNOTIFY lptbn;
int nItem;
static CHAR szBuf[128];
switch(pnmhdr->code)
{// Если получили сообщение от ToolTips, загружаем из ресурсов
// соответствующую текстовую строку
case TTN_NEEDTEXT:
lpToolTipText = (LPTOOLTIPTEXT)pnmhdr;
LoadString(hInst, lpToolTipText->hdr.idFrom,szBuf, sizeof(szBuf));
lpToolTipText->lpszText = szBuf;
break;
// Возвращаем окну Toolbar характеристики кнопки,
// с номером, заданным в lptbn->iItem
case TBN_GETBUTTONINFO:
lptbn = (LPTBNOTIFY)pnmhdr;
nItem = lptbn->iItem;
lptbn->tbButton.iBitmap   = tbButtons[nItem].iBitmap;
lptbn->tbButton.idCommand = tbButtons[nItem].idCommand;
lptbn->tbButton.fsState   = tbButtons[nItem].fsState;
lptbn->tbButton.fsStyle   = tbButtons[nItem].fsStyle;
lptbn->tbButton.dwData    = tbButtons[nItem].dwData;
```

```
lptbn->tbButton.iString   = tbButtons[nItem].iString;

// Если запрашиваются характеристики несуществующей
// кнопки, возвращаем FALSE
return ((nItem < sizeof(tbButtons)/sizeof(tbButtons[0]))?TRUE : FALSE);

// Разрешаем удаление любой кнопки, кроме самой первой
case TBN_QUERYDELETE:
lptbn = (LPTBNOTIFY)pnmhdr;
nItem = lptbn->iItem;
return (nItem == 0)? FALSE : TRUE;

// Разрешаем вставку любой кнопки, кроме самой первой
case TBN_QUERYINSERT:
lptbn = (LPTBNOTIFY)pnmhdr;
nItem = lptbn->iItem;
return (nItem == 0)? FALSE : TRUE;

// В ответ на завершение операции перемещения перерисовываем Toolbar
case TBN_TOOLBARCHANGE:
SendMessage(hwndTb, TB_AUTOSIZE, 0L, 0L);
return TRUE;
default: break;
  }
  return FALSE;}
////////////////////////////////////Обработчик WM_SIZE/////////////////////////////////////////////////
void WndProc_OnSize(HWND hwnd, UINT state, int w, int h)
{MessageBox(NULL, "Пришло сообщение WM_SIZE", "Шаблон Windows - программы", MB_OK);}
////////////////////////////////////Обработчик WM_MOVE/////////////////////////////////////////////////
void WndProc_OnMove(HWND hwnd, int cx, int cy)
{
MessageBox(NULL, "Пришло сообщение WM_MOVE", "Шаблон Windows - программы", MB_OK);}
//Обработка сообщения о помеченной позиции меню
void WndProc_OnMenuSelect(HWND hwnd, HMENU hmenu, int item,HMENU hmenuPopup,
UINT flags)
{static char szBuf[128];
szBuf[0] = 0;
// Загружаем строку из ресурсов приложения
LoadString(hInst,item, szBuf, sizeof(szBuf));
// Отображаем строку в первой области Toolbar
SendMessage(hwndSb, SB_SETTEXT, 0, (LPARAM)szBuf);
}
```

Файл twinbar.h

```
#include <commctrl.h>
HWND hwndTb;       // идентификатор Toolbar
HWND hwndSb;       // идентификатор Statusbar

// Описание кнопок Toolbar
TBBUTTON tbButtons[] =
{
  { 0, CM_NEW,  TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0},
  { 1, CM_OPEN, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0},
  { 2, CM_SAVE, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0},
  { 3, CM_SAVEAS,   TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0},
  { 0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0L, 0},
  { 4, CM_EXIT,  TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0}
};

//Прототипы функций обработчиков сообщений
BOOL WndProc_OnCreate(HWND hWnd, LPCREATESTRUCT lpCreateStruct);
void WndProc_OnDestroy(HWND hWnd);
BOOL WndProc_OnChar(HWND hwnd, TCHAR ch, int rpt);
BOOL WndProc_OnLbuttonDown(HWND hwnd, BOOL fDblClk, int x, int y, UINT
keyFlags);
void WndProc_OnCommand(HWND hWnd, int id,HWND hwndCtl, UINT codeNotify);
LRESULT WndProc_OnNotify(HWND hWnd, int idFrom, NMHDR FAR* pnmhdr);
void WndProc_OnSize(HWND hwnd, UINT state, int cx, int cy);
void WndProc_OnMove(HWND hwnd,int cx, int cy);
void WndProc_OnMenuSelect(HWND hwnd, HMENU hmenu, int item,
HMENU hmenuPopup, UINT flags);
```

ПРОГРАММА № 4. Вывод текста в окна Windows.

ЗАДАНИЕ:

Составить программу, которая:

1.Содержит однострочный редактор для формирования строк, переносимых затем в главное окно для рисования предварительно выбранным с помощью стандартного диалога шрифтом. Перенос текста в главное окно должен осуществляться по нажатию кнопки, которую надо расположить на поверхности окна.

2.Программу снабдить полосой прокрутки текста по верикали.

3.В дополнение к приведенному образцу составьте обработчик команды меню открытия файла и сделайте вывод в главное окно вывод строк из текстового файла. Используйте для этого меню Work2 (после переименования например в "Вывод строк из текстового файла").

МЕТОДИЧЕСКОЕ ПОСОБИЕ ПО СОСТАВЛЕНИЮ ПРОГРАММЫ.

Вам вряд ли придется создавать самим текстовый редактор для Windows - во-первых есть очень мощные редакторы, во-вторых это хлопотное дело, а в третьих в API есть многофункциональный стандартный редактор текста Rich Edit для работы как с обычным текстом, так и с текстом в формате RTF(хранение текста вместе

с шрифтовым оформлением и оформлением параграфов). Мы используем его в следу-
ющей работе, а пока освоим элементарные приемы вывода текста в графическое окно- это может понадобиться для различных кратких комментариев изображений (надписи на графиках функций и пр.) или вычислительного процесса.
1.Прежде всего снимем копию созданного в предыдущей работе шаблона ничего не делающей программы и перейдем к заполнению обработчиков сообщений.
-В редакторе ресурсов изменим ничего не значащее название меню WORK1 названием реальной работы - например "Ввод текста через строку редактирования" , а код команды переименуем в CM_EDIT1.
2.В обработчике этой команды создадим однострочный редактор и кнопку подтверждения окончания ввода с помощью универсальной функции создания окон CreateWindow.
3.Работа обработчика сообщения от кнопки описана в комментариях приводимого ниже образца и в лекционном материале.
4.Для обработчика команды выбора шрифта составим функцию GetFont, в которой заполним структурку CHOOSEFONT и вызовем библиотечную функцию ChoozeFont для выбора типа шрифта в стандартном диалоге.
5.Вертикальная полоса прокрутки появится в главном окне, если его стиль перед регистрацией в 3-м параметре ф-ции CreateWindow будет содержать флаг WS_VSCROLL.
(Вы можете создать полосу прокрутки и динамически с помощью универсальной ф-ции например так:

```
#define IDC_SCROLLBAR 1111
HWND
hscroll=CreateWindow("scrollbar",NULL,WS_CHILD|WS_VISIBLE|SBS_VERT,
            xleft,ytop,xright,ybottom,hwnd,IDC_SCROLLBAR,NULL))
```

С 9-ю стилями полос прокрутки вы познакомитесь по системе помощи - они имеют префикс SBS_.
В функции главного окна надо предусмотреть обработку сообщения WM_SCROLL, параметр wParam которого содержит код действия, совершаемого пользователем над
полосой - для вертикальной полосы это SB_TOP, SB_LINEUP, SB_LINEDOWN, SB_PAGEUP,SB_PAGEDOWN,
SB_BOTTOM,SB_THUMPPOSITION,SB_THUMBACK.
-При обработке WM_CREATE мы инициализируем полосу с помощью ф-ции SetScrollRange(hWnd,SB_VERT,0,200,TRUE);
указав диапазон изменеия значений позиции ползунка и установив начальную позицию ф-цией sbPos=0; SetScrollPos(hWnd,SB_VERT,sbPos,TRUE) - она же используется для переустановки позиции при обработке сообщений от полосы.
6.Все вводимые строки мы заносим в массив buf (200 строк по 80 симв) и выводим его при обработке WM_PAINT с учетом позиции полосы прокрутки.
*/

Файл textout.c

```
#include "textout.h"
```

```
// Имя класса окна
char const szClassName[]   = "TWindowClass";
// Заголовок окна
char const szWindowTitle[] ="Шаблон Widows-программы, которая ничего не
делает";
WNDCLASS wc;    // структура для регистрации класса окна
HINSTANCE hInst; //Идентификатор программы
//Прототипы функций
//Начальной инициализации
BOOL InitApp(HINSTANCE);
//Главного окна
LRESULT CALLBACK  WndProc(HWND,
                  UINT,//Код сообщения
                  WPARAM,//Специфическая для этого сообщения информация
                  LPARAM);
//////////////////////////////////////////// Функция WinMain/////////////////////////////////////////////
#pragma argsused
int WINAPI WinMain(
   HINSTANCE hInstance,   //handle запускаемой программы
   HINSTANCE hPrevInstance,
   LPSTR lpCmdLine,        //Указатель на командную строку
   int nCmdShow            // Код состояния окна при запуске программы
)
{
MSG  msg;      // структура для работы с сообщениями
HWND hwnd; // идентификатор главного окна приложения
//Сохраним в глоб переменной идентификатор приложения
hInst=hInstance;
// Инициализируем приложение
if(!InitApp(hInstance)) return FALSE; //Если не удалось - выходим
// Инициализируем библиотеку стандартных органов управления
InitCommonControls();
// После успешной инициализации приложения создаем главное окно
hwnd = CreateWindow(
szClassName,         // имя класса окна
szWindowTitle,       // заголовок окна
WS_OVERLAPPEDWINDOW |WS_VSCROLL,// стиль окна
CW_USEDEFAULT,       //горизонтальная позиция окна (у нас - по умолчанию)
CW_USEDEFAULT,       //вертикальная позиция окна
CW_USEDEFAULT,       //ширина окна
CW_USEDEFAULT,       //высота окна
0,                 //идентификатор родительского окна
0,                 // идентификатор меню или дочернего окна
hInst,               //идентификатор программы
NULL);               //указатель на дополнительные данные
// Если создать окно не удалось, завершаем приложение
```

```
if(!hwnd)    return FALSE;
// Рисуем главное окно
ShowWindow(hwnd, nCmdShow  );
UpdateWindow(hwnd);
//Запускаем цикл обработки сообщений - его простейший вариант
while(GetMessage(      //Функция выборки сообщения из очереди
              &msg,//Адрес структуры с сообщением
              0,   //Идентификатор окна, котрому надо переадресовать сообщ
              0,   //Начало диапазона отбираемых (фильтруемых) сообщений
              0))  //Конец диапазона отбираемых (фильтруемых) сообщений
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);//Ф-ция распределяет сообщение нужной оконной
функции
  }
  return msg.wParam;
}
////////////////////// Функция InitApp Выполняет регистрацию класса окна/////////////////
BOOL
InitApp(HINSTANCE hInstance)
{ATOM aWndClass; // атом для кода возврата
WNDCLASS wc;    // структура для регистрации
// Преверяем, не было ли это приложение запущено ранее
HWND hWnd = FindWindow(szClassName, NULL);
if(hWnd)  {
if(IsIconic(hWnd)) ShowWindow(hWnd, SW_RESTORE);
SetForegroundWindow(hWnd);
return FALSE;
  }
//Заполним поля окнной структуры
wc.lpszMenuName  = MAKEINTRESOURCE(IDM_MENU1);//Имя меню
wc.style         = CS_HREDRAW | CS_VREDRAW;//Стиль класса окна
wc.lpfnWndProc   = (WNDPROC) WndProc;      //Указатель на оконную функцию
wc.cbClsExtra    = 0;                  //Дополнит данные для класса окна
wc.cbWndExtra    = 0;                   //Дополнит данные для окна
wc.hInstance     = hInstance;             //Идентификатор программы
wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);//Иконка для программы
wc.hCursor       = LoadCursor(NULL, IDC_ARROW);    //Курсор мышки
wc.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);//Кисть для фона
  wc.lpszClassName = (LPSTR)szClassName;     //Имя класса окна
  aWndClass = RegisterClass(&wc);           //Библиот ф-ция регистрации в ОС
   return (aWndClass != 0);
}
//Оконная функция - обработчик сообщений для главного окна программы
//Пусть 2 пары переменных хранят размеры клиентской области окна и его позицию
int cwClient,chClient,xPos,yPos;
```

```
LRESULT CALLBACK WndProc( HWND hWnd,
                  UINT msg, //Код сообщения
                  WPARAM wParam,//Специфическая для сообщения информация
                  LPARAM lParam)
{switch (msg)
  {//Макрокоманды разборки сообщений
HANDLE_MSG(hWnd, WM_CREATE,WndProc_OnCreate);
HANDLE_MSG(hWnd, WM_DESTROY,WndProc_OnDestroy);
HANDLE_MSG(hWnd, WM_NOTIFY,WndProc_OnNotify);
HANDLE_MSG(hWnd, WM_SIZE,WndProc_OnSize);
HANDLE_MSG(hWnd, WM_MOVE,WndProc_OnMove);
HANDLE_MSG(hWnd, WM_PAINT,WndProc_OnPaint);
HANDLE_MSG(hWnd, WM_VSCROLL,WndProc_OnVScroll);
HANDLE_MSG(hWnd, WM_MENUSELECT,WndProc_OnMenuSelect);
HANDLE_MSG(hWnd, WM_COMMAND,WndProc_OnCommand);
//Обработку всех других сообщений, не упомянутых у нас, поручим Windows
default:    return(DefWindowProc(hWnd, msg, wParam, lParam));
   }}
//Действия выполняемые при создании окна
#pragma argsused
BOOL WndProc_OnCreate(HWND hWnd, LPCREATESTRUCT lpCreateStruct)
{//Инициализируем вертикальную полосу просмотра
SetScrollRange(hWnd,SB_VERT,0,200,TRUE);
sbPos=0;
SetScrollPos(hWnd,SB_VERT,sbPos,TRUE);
// Создаем Toolbar
hwndTb = CreateToolbarEx(hWnd,
                  WS_CHILD | WS_BORDER | WS_VISIBLE |//Стили
                  TBSTYLE_TOOLTIPS | CCS_ADJUSTABLE,
   IDT_TOOLBAR,          // идентификатор органа Toolbar
   5,                    // количество пиктограмм
   hInst,                // идентификатор приложения
   IDB_TBBITMAP,         // идентификатор битового изображения
                         //   кнопок
   (LPCTBBUTTON)&tbButtons,// адрес описания кнопок
   6,                    // количество кнопок
   16,16,                // ширина и высота кнопок
   16,16,                // ширина и высота пиктограмм
   sizeof(TBBUTTON));     // размер структуры в байтах
   if(hwndTb == NULL)    return FALSE;
   //Создаем  StatusBar
   hwndSb = CreateWindowEx(
   0L,                   // расширенный стиль окна
   STATUSCLASSNAME,      // класс окна для Statusbar
   "",                   // заголовок окна отсутствует
   WS_CHILD | WS_BORDER |  // стиль окна
```

```
   WS_VISIBLE | SBARS_SIZEGRIP,
   CW_USEDEFAULT, CW_USEDEFAULT,// координаты, ширина, высота
   CW_USEDEFAULT,CW_USEDEFAULT,
   hWnd,                // идентификатор родительского окна
   (HMENU)IDS_STATUSBAR,   // идентификатор Statusbar
   hInst,                  // идентификатор приложения
   NULL );              // доп. данные для окна
   if(hwndSb == NULL)    return FALSE;
   return TRUE;    }
//////////////////////////Обработчик WM_DESTROY/////////////////////////////////////////
#pragma argsused
void WndProc_OnDestroy(HWND hWnd)
{PostQuitMessage(0);}
////////////////////////////////////////////WM_PAINT/////////////////////////////////////////////////
void WndProc_OnPaint(HWND hwnd)
{int i;
HDC hdc=BeginPaint(hwnd,&ps);
//Получим идентификатор шрифта при его создании по шаблону lf типа LOGFONT
hfont=CreateFontIndirect(&lf);
hOldFont=SelectFont(hdc,hfont); //Возьмем в контекст шрифт
GetTextMetrics(hdc,&tm);  //Запишем метрики текста в структуру TEXTMETRIC tm;
SetTextColor(hdc,cf.rgbColors); //Возьмем цвет из стуктуры CHOOSEFONT cf
for(i=sbPos,ytext=2*tm.tmHeight;i<=cbuf;i++,ytext+=tm.tmHeight)
{TextOut(hdc,xtext,ytext,buf[i],strlen(buf[i]));}
SelectFont(hdc,hOldFont);//Вернем старый шрифт
DeleteFont(hfont);       //Удаляем созданный шрифт
EndPaint(hwnd,&ps);
}
////////////////////////////////////////////////Обработчик команд/////////////////////////////////////
void WndProc_OnCommand(HWND hWnd, int id,HWND hwndCtl, UINT codeNotify)
{switch (id)
  {//////////////////////////Если в меню выбран ввод через 1-строчный редактор//////////
case CM_EDIT1:
{//Создадим дочернее окно однострочного редактора внизу над строкой статуса
hEdit = CreateWindow("edit", NULL,  WS_CHILD | WS_VISIBLE | WS_BORDER
|ES_LEFT |ES_AUTOHSCROLL,0,  clientRect.bottom-40, clientRect.right-80,
                20, hWnd, (HMENU)ID_EDIT, hInst, NULL);
// Создаем кнопку справа от строки ввода для сообщения об окончании ввода
hButton = CreateWindow("button", "OK",WS_CHILD | WS_VISIBLE
|BS_PUSHBUTTON,
                      clientRect.right-70,clientRect.bottom-40 ,50, 20,
                       hWnd, (HMENU)ID_BUTTON, hInst, NULL);
SetFocus(hEdit);      //Фокус ввода - в строку редактора
break;
}
//Если нажата кнопка окончания набора в строке ввода
```

```
case ID_BUTTON:
if(id==ID_BUTTON)
{
*((WORD*)buf[cbuf])=78;//В первое слово массива запишем его длину
//Пошлем редактору запрос перенести строку в buf
SendMessage(hEdit,EM_GETLINE,(WPARAM)1L,(LPARAM)(LPSTR)buf[cbuf]);
//Рисовать поручим обработчику WM_PAINT
InvalidateRect(hWnd,NULL,FALSE);
if(cbuf<199)cbuf++;
break;
}
case CM_EXIT: PostQuitMessage(0); break;
case CM_NEW: MessageBox(NULL, "Команда CM_NEW создания нового файла",
"Шаблон Windows - программы", MB_OK); break;
case CM_OPEN:MessageBox(NULL, "Команда CM_OPEN открытияия  файла",
"Шаблон Windows - программы", MB_OK); break;
case CM_SAVE: MessageBox(NULL, "Команда CM_SAVE сохранения  файла",
"Шаблон Windows - программы", MB_OK); break;
case CM_SAVEAS: MessageBox(NULL, "Команда CM_SAVEAS сохранения  файла",
"Шаблон Windows - программы", MB_OK); break;
////////////////////////////////////////////////Команда выбора шрифта//////////////////////////////
case CM_SELECTFONT:{
// Выбираем шрифт для вывода текста
memset(&lf, 0, sizeof(LOGFONT));
GetFont(hWnd,&lf,&cf);
lf.lfEscapement= lf.lfOrientation=450;//Можно задать выводпод углом
break;
}
default:break;
}  }
//////////////////////////////////////////// Функция WndProc_OnNotify ////////////////////////////
#pragma argsused
LRESULT WndProc_OnNotify(HWND hWnd , int idFrom, NMHDR* pnmhdr)
{
LPTOOLTIPTEXT lpToolTipText;
LPTBNOTIFY lptbn;
int nItem;
static CHAR szBuf[128];
switch(pnmhdr->code)
{
// Если получили сообщение от ToolTips, загружаем из ресурсов
// соответствующую текстовую строку
case TTN_NEEDTEXT:
lpToolTipText = (LPTOOLTIPTEXT)pnmhdr;
LoadString(hInst, lpToolTipText->hdr.idFrom,szBuf, sizeof(szBuf));
lpToolTipText->lpszText = szBuf;
```

```
break;
//Возвращаем Toolbar характеристики кнопки с номером,заданным в lptbn->iItem
case TBN_GETBUTTONINFO:
lptbn = (LPTBNOTIFY)pnmhdr;
nItem = lptbn->iItem;
lptbn->tbButton.iBitmap   = tbButtons[nItem].iBitmap;
lptbn->tbButton.idCommand = tbButtons[nItem].idCommand;
lptbn->tbButton.fsState   = tbButtons[nItem].fsState;
lptbn->tbButton.fsStyle   = tbButtons[nItem].fsStyle;
lptbn->tbButton.dwData    = tbButtons[nItem].dwData;
lptbn->tbButton.iString   = tbButtons[nItem].iString;
//Если запрашиваются характеристики несуществующей кнопки, возвращаем FALSE
return ((nItem < sizeof(tbButtons)/sizeof(tbButtons[0]))?TRUE : FALSE);
// Разрешаем удаление любой кнопки, кроме самой первой
case TBN_QUERYDELETE:
lptbn = (LPTBNOTIFY)pnmhdr;
nItem = lptbn->iItem;
return (nItem == 0)? FALSE : TRUE;
//Разрешаем вставку любой кнопки, кроме самой первой
case TBN_QUERYINSERT:
lptbn = (LPTBNOTIFY)pnmhdr;
nItem = lptbn->iItem;
return (nItem == 0)? FALSE : TRUE;
// В ответ на завершение операции перемещенияперерисовываем Toolbar
case TBN_TOOLBARCHANGE:
SendMessage(hwndTb, TB_AUTOSIZE, 0L, 0L);
return TRUE;
default: break;
  }
return FALSE;}

//Обработка сообщения об изменении размеров окна w,h-новые размеры окна
void WndProc_OnSize(HWND hwnd, UINT state, int w, int h)
{//Получим новые размеры главного окна
GetWindowRect(hwnd,&wndRect);
GetClientRect(hwnd,&clientRect);
//Если в окне присутствует строка ввода - уничтожим и создадим ее заново
if(hEdit!=NULL)
{//Сохраним в буфере обмена содержимое строки редактора
SendMessage(hEdit,EM_SETSEL,0L,-1L);//Вначале выделим текст
SendMessage(hEdit,WM_COPY,0L,0L);   //Затем сохраним выделенный
DestroyWindow(hEdit);DestroyWindow(hButton);//Уничтожим строку ввода и кнопку
//Создадим заново окно однострочного редактора внизу над строкой статуса
hEdit = CreateWindow("edit", NULL,  WS_CHILD | WS_VISIBLE | WS_BORDER
|ES_LEFT|ES_AUTOHSCROLL,0,  clientRect.bottom-40, clientRect.right-80,  20,
                    hwnd, (HMENU)ID_EDIT, hInst, NULL);
```

```
//Восстановим из буфера содержимое строки
SendMessage(hEdit,WM_PASTE,0L,0L);
// Создаем заново  кнопку
hButton = CreateWindow("button", "OK",WS_CHILD | WS_VISIBLE
|BS_PUSHBUTTON,
                clientRect.right-70,clientRect.bottom-40 ,50, 20,
                 hwnd, (HMENU)ID_BUTTON, hInst, NULL);
}
//Сообщим окну StatusBar об изменении размеров главного окна,
//чтобы оно скорректировало и свои размеры.
xtext=0;ytext=20;
SendMessage(hwndSb,WM_SIZE,w,h);
}
//Обработка сообщения об изменении позиции окна cx,cy-новые координаты окна
#pragma argsused
void WndProc_OnMove(HWND hwnd, int cx, int cy){  }
////////////////////////////////////Сообщение от полосы просмотра//////////////////////////
void WndProc_OnVScroll(HWND hwnd, HWND hwndCtl, UINT code, int pos)
{switch(code)
{
//case SB_PAGEUP:sbPos-=10;break;
//case SB_PAGEDOWN:sbPos+=10;break;
case SB_LINEDOWN:sbPos+=1;;break;
case SB_LINEUP:sbPos-=1;break;
case SB_TOP:sbPos=0;break;
case SB_BOTTOM:sbPos=200;break;
case SB_THUMBPOSITION:sbPos=pos;break;
case SB_THUMBTRACK:break;
}
if(sbPos>200) sbPos=200;
if(sbPos<1) sbPos=0;
SetScrollPos(hwnd,SB_VERT,sbPos,TRUE);
InvalidateRect(hwnd,NULL,TRUE);
}
//////////////////////Обработка сообщения о помеченной позиции меню//////////////////
#pragma argsused
void WndProc_OnMenuSelect(HWND hwnd, HMENU hmenu, int item,HMENU
hmenuPopup,
UINT flags)
{static char szBuf[128];
szBuf[0] = 0;
// Загружаем строку из ресурсов приложения
LoadString(hInst,item, szBuf, sizeof(szBuf));
// Отображаем строку в первой области Toolbar
SendMessage(hwndSb, SB_SETTEXT, 0, (LPARAM)szBuf);
}
```

```
////////////////////////////// Функция GetFont - выбор шрифта в стандартном диалоге////////
BOOL GetFont(HWND hWnd, LOGFONT *lf, CHOOSEFONT *cf)
{LPSTR szFontStyle[LF_FACESIZE];
memset(cf, 0, sizeof(CHOOSEFONT));
cf->lStructSize = sizeof(CHOOSEFONT);//Размер структуры
cf->hwndOwner = hWnd;              // Идентификатор окна
cf->lpLogFont = lf;                // Указатель на структуру LOGFONT
// Флаги, определяющие внешний вид диалоговой панели
cf->Flags = CF_SCREENFONTS | CF_USESTYLE| CF_EFFECTS;
cf->lCustData = 0L;                // Дополнительные данные
cf->rgbColors = RGB(255,0,0);      // Цвет текста
cf->lpfnHook = (FARPROC)NULL;      //Адрес функции фильтра
cf->lpTemplateName = (LPSTR)NULL;  //Адрес шаблона диалоговой панели
cf->hInstance = hInst;             //Идентификатор копии приложения
cf->lpszStyle = (LPSTR)szFontStyle;  //Стиль шрифта
cf->nFontType = SCREEN_FONTTYPE;   //Тип шрифта
// Ограничения на минимальный и максимальный  размер шрифта
cf->nSizeMin = 0; cf->nSizeMax = 0;
return ChooseFont(cf);             //Вызываем функцию выбора шрифта
}
```

Файл textout.h

```
#define STRICT      //Просим коппилятор осуществлять строгую проверку типов
#include <windows.h> //Включить файл интерфейса с библиотекой Windows API
#include <windowsx.h>//То же для связи с расширенной версией Windows API
#include <commctrl.h>
#include "afxres.h"
#include "resource.h"
RECT wndRect;      //Структура для размеров окна
RECT clientRect;   //Структура для размеров клиентской области окна
RECT sbRect;       //Для размеров  строки статуса
HWND hwndTb;       // идентификатор Toolbar
HWND hwndSb;       // идентификатор Statusbar
HWND hEdit;        //Идентификатор однострочного текстового редактора
HWND hButton;      //Кнопка
// Идентификатор редактора текста
#define ID_EDIT   11111
#define ID_BUTTON 22222
CHOOSEFONT cf;        //Структура с информацией для одноименной функции
LOGFONT lf;           //Структура с аттрибутами шрифтов
HFONT hfont, hOldFont; //Идентификаторы  шрифтов
TEXTMETRIC tm;
int xtext,ytext;
int sbPos;
PAINTSTRUCT ps;
```

```
char buf[200][80];
POINT pos;
int cbuf;
// Описание кнопок Toolbar
TBBUTTON tbButtons[] =
{
  { 0, CM_NEW,  TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0},
  { 1, CM_OPEN, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0},
  { 2, CM_SAVE, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0},
  { 3, CM_SAVEAS,   TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0},
  { 0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0L, 0},
  { 4, CM_EXIT,  TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0}
};

//Прототипы функций обработчиков сообщений
void WndProc_OnSetFocus(HWND hwnd,HWND hwndOldFocus);
void WndProc_OnKeyDown(HWND hwnd,UINT vk,BOOL down, int cRpt,UINT flag);
BOOL WndProc_OnCreate(HWND hWnd, LPCREATESTRUCT lpCreateStruct);
void WndProc_OnDestroy(HWND hWnd);
void WndProc_OnPaint(HWND hwnd);
void WndProc_OnCommand(HWND hWnd, int id,HWND hwndCtl, UINT codeNotify);
LRESULT WndProc_OnNotify(HWND hWnd, int idFrom, NMHDR FAR* pnmhdr);
void WndProc_OnSize(HWND hwnd, UINT state, int cx, int cy);
void WndProc_OnMove(HWND hwnd,int cx, int cy);
void WndProc_OnMenuSelect(HWND hwnd, HMENU hmenu, int item,
HMENU hmenuPopup, UINT flags);
GetFont(HWND hwnd,LOGFONT *lf,CHOOSEFONT* cf);
void WndProc_OnVScroll(HWND hwnd, HWND hwndCtl, UINT code, int pos);
```

ПРОГРАММА № 5. Использование органа управления RICH EDIT CONTROL.
ЗАДАНИЕ:
Составить программу - текстовый редактор на базе класса окна "RICHEDIT".
Предусмотреть возможность выбора для редактирования и сохранения файла
с произвольным именем, выбираемого с помощью стандартных диалогов.
Факультатив: предусмотреть возможность печати файла на принтере с
использованием для настройки печати стандартной панели диалога.
МЕТОДИЧЕСКОЕ ПОСОБИЕ
1)По созданию окна RICHEDIT
В качестве образца используйте программу rtfpad из 22-го тома Фроловых -
мы приводим ее ниже в несколько подкорректированном виде.
Перед созданием окна с помощью универсальной функции CreateWindowEx
необходимо загрузить в память библиотеку RICHED32.DLL с помощью функции
LoadLibrary:

```
HINSTANCE hRTFLib=LoadLibrary("RICHED32.DLL");
```

С сообщениями для редактора ознакомьтесь по системе помощи Win32 OnLine Help.
2)По работе с файлами через стандартные диалоги

В составе операционной системы Windows имеется DLL-библиотека commdlg.dll, экспортирующая среди прочих две функции, очень удобные для организации пользовательского интерфейса при открытии файлов.
Это функции GetOpenFileName и GetSaveFileName, которые выводят для выбора файла
стандартные панели диалога.
Внешний вид этих диалоговых панелей определяется структурой типа OPENFILENAME ,
определенной в файле commdlg.h (этот файл находится в каталоге include системы разработки Borland C++ или Microsoft Visual C++):
Вам необходимо заполнить поля этой стуктуры перед вызовом указанных функций.
*/

Файл rtfpad..

```
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include <richedit.h>
#include <string.h>
#include "resource.h"
#include "afxres.h"
#include "rtfpad.h"

HINSTANCE hInst;
char szAppName[]  = "RtfEditApp";
char szAppTitle[] = "Rich Text Editor RtfPad";
HWND hwndEdit;
HINSTANCE hRTFLib;
//////////////////////////////////////////// Функция WinMain////////////////////////////////////////
int APIENTRY
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,LPSTR lpCmdLine,
int nCmdShow)
{ WNDCLASSEX wc;  HWND hWnd; MSG msg;
  hInst = hInstance;
// Преверяем, не было ли это приложение запущено ранее
  hWnd = FindWindow(szAppName, NULL);
  if(hWnd)
  {    if(IsIconic(hWnd)) ShowWindow(hWnd, SW_RESTORE);
                                SetForegroundWindow(hWnd);
                                 return FALSE;
  }
// Загружаем библиотеку RICHED32.DLL
  hRTFLib = LoadLibrary("RICHED32.DLL");
  if(!hRTFLib)  return FALSE;
//Регистрируем класс окна
  memset(&wc, 0, sizeof(wc));
```

```
  wc.cbSize = sizeof(WNDCLASSEX);
  wc.hIconSm = LoadImage(hInst, MAKEINTRESOURCE(IDI_APPICONSM),
  IMAGE_ICON, 16, 16, 0);
  wc.style = 0;
  wc.lpfnWndProc = (WNDPROC)WndProc;
  wc.cbClsExtra  = 0;
  wc.cbWndExtra  = 0;
  wc.hInstance = hInst;
  wc.hIcon = LoadImage(hInst,MAKEINTRESOURCE(IDI_APPICON), IMAGE_ICON,
32, 32, 0);
  wc.hCursor = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
  wc.lpszMenuName = MAKEINTRESOURCE(IDR_APPMENU);
  wc.lpszClassName = szAppName;
  if(!RegisterClassEx(&wc))
  if(!RegisterClass((LPWNDCLASS)&wc.style)) return FALSE;
// Создаем главное окно приложения
  hWnd = CreateWindow(szAppName, szAppTitle, WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInst, NULL);
  if(!hWnd) return(FALSE);
// Отображаем окно и запускаем цикл обработки сообщений
  ShowWindow(hWnd, nCmdShow);  UpdateWindow(hWnd);
  while(GetMessage(&msg, NULL, 0, 0))
  {TranslateMessage(&msg);  DispatchMessage(&msg); }
  return msg.wParam;
}
//////////////////////////////////////////////// Функция WndProc//////////////////////////////
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{  switch(msg)
   {
    HANDLE_MSG(hWnd, WM_CREATE,     WndProc_OnCreate);
    HANDLE_MSG(hWnd, WM_DESTROY,    WndProc_OnDestroy);
    HANDLE_MSG(hWnd, WM_COMMAND,    WndProc_OnCommand);
    HANDLE_MSG(hWnd, WM_SIZE,       WndProc_OnSize);
    HANDLE_MSG(hWnd, WM_SETFOCUS,   WndProc_OnSetFocus);
   default: return(DefWindowProc(hWnd, msg, wParam, lParam));
   }
}
//////////////////////////// Функция WndProc_OnCreate/////////////////////////////////////
BOOL WndProc_OnCreate(HWND hWnd, LPCREATESTRUCT lpCreateStruct)
{
RECT rc;
// Определяем размеры внутренней области главного окна
GetClientRect(hWnd, &rc);
// Создаем орган управления Rich Edit
```

```
  hwndEdit = CreateWindowEx(0L, "RICHEDIT", "",
    WS_VISIBLE | WS_CHILD | WS_BORDER | WS_HSCROLL | WS_VSCROLL |
    ES_NOHIDESEL | ES_AUTOVSCROLL | ES_MULTILINE |
    ES_SAVESEL | ES_SUNKEN,
    0, 0, rc.right - rc.left, rc.bottom - rc.top,
    hWnd, (HMENU) IDC_RTFEDIT, hInst, NULL);
  if(hwndEdit == NULL)    return FALSE;
// Передаем фокус ввода органу управления Rich Edit
  SetFocus(hwndEdit);
  return TRUE;
}

//////////////////////////////////////// Функция WndProc_OnDestroy//////////////////////
#pragma warning(disable: 4098)
void WndProc_OnDestroy(HWND hWnd)
{// Уничтожаем орган управления Rich Edit
if(hwndEdit)    DestroyWindow(hwndEdit);
// Освобождаем библиотеку RICHED32.DLL
if(hRTFLib)  FreeLibrary(hRTFLib);
PostQuitMessage(0);
}
////////////////// ////////////////////////Функция WndProc_OnCommand//////////////////////
#pragma warning(disable: 4098)
void WndProc_OnCommand(HWND hWnd, int id, HWND hwndCtl, UINT codeNotify)
{
  CHARFORMAT cf;
  CHOOSEFONT chfnt;
  LOGFONT lf;
  HDC hDC;
  PARAFORMAT pf;
  switch (id)
{// Изменяем жирность символов
case ID_FORMAT_BOLD:    {cf.cbSize = sizeof(cf);
// Определяем формат символов
SendMessage(hwndEdit, EM_GETCHARFORMAT, TRUE, (LPARAM)&cf);
// Изменяем бит поля dwEffects, с помощью которого
// можно выделить символы как bold (жирное начертание)
    cf.dwMask = CFM_BOLD;
// Инвертируем бит, определяющий жирное начертание
cf.dwEffects ^= CFE_BOLD;
// Изменяем формат символов
SendMessage(hwndEdit, EM_SETCHARFORMAT, SCF_SELECTION,
(LPARAM)&cf);
break;
    }
// Устанавливаем или отменяем наклонное начертание символов
```

```
case ID_FORMAT_ITALIC:
   { cf.cbSize = sizeof(cf);
     SendMessage(hwndEdit, EM_GETCHARFORMAT, TRUE, (LPARAM)&cf);
     cf.dwMask = CFM_ITALIC;
     cf.dwEffects ^= CFE_ITALIC;
     SendMessage(hwndEdit, EM_SETCHARFORMAT, SCF_SELECTION,
(LPARAM)&cf);
     break;
   }
//Устанавливаем или отменяем выделение символов подчеркиванием
   case ID_FORMAT_UNDERLINE:
   { cf.cbSize = sizeof(cf);
     SendMessage(hwndEdit, EM_GETCHARFORMAT, TRUE, (LPARAM)&cf);
     cf.dwMask = CFM_UNDERLINE;
     cf.dwEffects ^= CFE_UNDERLINE;
     SendMessage(hwndEdit, EM_SETCHARFORMAT,SCF_SELECTION,
(LPARAM)&cf);
     break;   }
// Изменяем шрифт символов
case ID_FORMAT_FONT:
   { cf.cbSize = sizeof(cf);
// Определяем текущий формат символов
SendMessage(hwndEdit, EM_GETCHARFORMAT, TRUE, (LPARAM)&cf);
//Сбрасываем содержимое структур, которые будут использованы для выбора
шрифта
memset(&chfnt, 0, sizeof(chfnt)); memset(&lf, 0, sizeof(lf));
// Получаем контекст отображения
hDC = GetDC(hWnd);
// Если было задано выделение наклоном или жирным шрифтом,
// подбираем шрифт с соответствующими атрибутами
lf.lfItalic    = (BOOL)(cf.dwEffects & CFE_ITALIC);
lf.lfUnderline = (BOOL)(cf.dwEffects & CFE_UNDERLINE);
// Преобразуем высоту из TWIPS-ов в пикселы.
// Устанавливаем отрицательный знак, чтобы
// выполнялось преобразование и использовалось абсолютное
// значение высоты символов
lf.lfHeight = - cf.yHeight/20;
// Набор символов, принятый по умолчанию
lf.lfCharSet = ANSI_CHARSET;
// Качество символов, принятое по умолчанию
lf.lfQuality = DEFAULT_QUALITY;
// Выбираем семейство шрифтов
lf.lfPitchAndFamily = cf.bPitchAndFamily;
// Название начертания шрифта
lstrcpy(lf.lfFaceName, cf.szFaceName);
// Устанавливаем вес шрифта в зависимости от того,
```

```
// было использовано выделение жирным шрифтом или нет
if(cf.dwEffects & CFE_BOLD) lf.lfWeight = FW_BOLD;
else   lf.lfWeight = FW_NORMAL;
// Заполняем структуру для функции выбора шрифта
chfnt.lStructSize = sizeof(chfnt);
chfnt.Flags = CF_SCREENFONTS | CF_INITTOLOGFONTSTRUCT;
chfnt.hDC = hDC;
chfnt.hwndOwner   = hWnd;
chfnt.lpLogFont   = &lf;
chfnt.rgbColors   = RGB(0,0,0);
chfnt.nFontType   = SCREEN_FONTTYPE;
// Выводим на экран диалоговую панель для выбора шрифта
if(ChooseFont(&chfnt))
{// Можно использовать все биты поля dwEffects
cf.dwMask = CFM_BOLD | CFM_FACE | CFM_ITALIC |
          CFM_UNDERLINE | CFM_SIZE | CFM_OFFSET;
// Преобразование в TWIPS-ы
cf.yHeight = - lf.lfHeight * 20;
// Устанавливаем поле dwEffects
cf.dwEffects = 0;
if(lf.lfUnderline) cf.dwEffects |= CFE_UNDERLINE;
if(lf.lfWeight == FW_BOLD) cf.dwEffects |= CFE_BOLD;
if(lf.lfItalic)   cf.dwEffects |= CFE_ITALIC;
// Устанавливаем семейство шрифта
cf.bPitchAndFamily = lf.lfPitchAndFamily;
// Устанавливаем название начертания шрифта
lstrcpy(cf.szFaceName, lf.lfFaceName);
// Изменяем шрифтовое оформление символов
SendMessage(hwndEdit, EM_SETCHARFORMAT,SCF_SELECTION, (LPARAM)&cf);
     }
// Освобождаем контекст отображения
ReleaseDC(hWnd, hDC);
break;
   }
// Устанавливаем выравнивание параграфа по левой границе
// окна органа управления Rich Edit
case ID_FORMAT_PARAGRAPH_LEFT:
   {
pf.cbSize = sizeof(pf);
pf.dwMask = PFM_ALIGNMENT;
pf.wAlignment = PFA_LEFT;
// Изменяем тип выравнивания текущего параграфа
SendMessage(hwndEdit, EM_SETPARAFORMAT, 0, (LPARAM)&pf);
break;
   }
// Устанавливаем выравнивание параграфа по правой границе
```

```
// окна органа управления Rich Edit
case ID_FORMAT_PARAGRAPH_RIGHT:
    {
     pf.cbSize = sizeof(pf);
     pf.dwMask = PFM_ALIGNMENT;
     pf.wAlignment = PFA_RIGHT;
     SendMessage(hwndEdit, EM_SETPARAFORMAT, 0, (LPARAM)&pf);
                                        break;
    }
// Выполняем центровку текущего параграфа
                case ID_FORMAT_PARAGRAPH_CENTER:
    {
     pf.cbSize = sizeof(pf);
     pf.dwMask = PFM_ALIGNMENT;
     pf.wAlignment = PFA_CENTER;
     SendMessage(hwndEdit, EM_SETPARAFORMAT, 0, (LPARAM)&pf);
                                         break;
    }
// Реализуем стандартные функции меню Edit
   case ID_EDIT_UNDO:SendMessage(hwndEdit, EM_UNDO, 0, 0L);
                                        break;
     case ID_EDIT_CUT: SendMessage(hwndEdit, WM_CUT, 0, 0L);
                                        break;
   case ID_EDIT_COPY:SendMessage(hwndEdit, WM_COPY, 0, 0L);
     break;
  case ID_EDIT_PASTE:SendMessage(hwndEdit, WM_PASTE, 0, 0L);
                                        break;
case ID_EDIT_DELETE:SendMessage(hwndEdit, WM_CLEAR, 0, 0L);
                                        break;
// Выделяем весь текст, который есть в окне органа управления Rich Edit
                           case ID_EDIT_SELECTALL:
    { CHARRANGE charr;
     charr.cpMin = 0;  // от начала...
     charr.cpMax = -1; // ... и до конца текста
     SendMessage(hwndEdit, EM_EXSETSEL, 0, (LPARAM)&charr);
     break;
    }
// При создании нового текста удаляем текущее содержимое окна редактирования
            case ID_FILE_NEW: SetWindowText(hwndEdit,"\0");
                                        break;
case ID_FILE_OPEN:  FileOpen(hWnd);// загружаем файл для редактирования
                                        break;
case ID_FILE_SAVEAS:FileSaveAs(hWnd); // сохраняем текст в файле
                                        break;
           case ID_FILE_PRINT: FilePrint();      // печатаем текст
                                        break;
```

```
case ID_FILE_EXIT:  PostQuitMessage(0); // завершаем работу приложения
                                        break;
case ID_HELP_ABOUT: MessageBox(hWnd,
       "Rich Text Editor RtfPad, v.1.0\n"
       "(C) Alexandr Frolov, 1995\n"
       "Email: frolov@glas.apc.org",
       szAppTitle, MB_OK | MB_ICONINFORMATION);
                                          break;
                                       default:                                  break;
  }}
//////////////////////////////////////////////// Функция WndProc_OnSize//////////////////
#pragma warning(disable: 4098)
void WndProc_OnSize(HWND hwnd, UINT state, int cx, int cy)
{ MoveWindow(hwndEdit, 0, 0, cx, cy, TRUE);}
//////////////////////////////////////////////// Функция WndProc_OnSetFocus/////////////
#pragma warning(disable: 4098)
void WndProc_OnSetFocus(HWND hwnd, HWND hwndOldFocus)
{ // Когда главное окно нашего приложения получает
  // фокус ввода, оно передает фокус ввода окну
  // органа управления Rich Edit
  SetFocus(hwndEdit);
}
////////////////////////////////////// Функция FileSaveAs/////////////////////////////////////
void FileSaveAs(HWND hwnd)
{
  OPENFILENAME ofn;
  char szFile[256] = "untitled.rtf";
  char szDirName[512];
  char szFileTitle[256];
// Фильтр допускает сохранение текста в файле с
// расширением имени rtf, txt, или любым другим
  char szFilter[256] =
    "Rich Text Files\0*.rtf\0Text Files\0*.txt\0"
    "Any Files\0*.*\0";
  HFILE      hFile;
  OFSTRUCT   of;
  EDITSTREAM es;
  memset(&ofn, 0, sizeof(OPENFILENAME));
// Определяем путь к текущему каталогу
  GetCurrentDirectory(sizeof(szDirName), szDirName);
// Заполняем структуру для выбора выходного файла
  ofn.lStructSize     = sizeof(OPENFILENAME);
  ofn.hwndOwner       = hwnd;
  ofn.lpstrFilter     = szFilter;
  ofn.lpstrInitialDir = szDirName;
  ofn.nFilterIndex    = 1;
```

```
  ofn.lpstrFile       = szFile;
  ofn.nMaxFile        = sizeof(szFile);
  ofn.lpstrFileTitle  = szFileTitle;
  ofn.nMaxFileTitle   = sizeof(szFileTitle);
  ofn.lpstrDefExt = "rtf";
  ofn.Flags = OFN_OVERWRITEPROMPT | OFN_HIDEREADONLY;
// Выводим на экран диалоговую панель, предназначенную
// для выбора выходного файла
if(GetSaveFileName(&ofn))
{// Если файл выбран, открываем его для записи или создаем
    if (*ofn.lpstrFile)
    { hFile = OpenFile(ofn.lpstrFile, &of, OF_CREATE);
// Устанавливаем параметры функции обратного вызова,
// которая будет выполнять запись
      es.dwCookie    = (DWORD)hFile;
      es.dwError     = 0;
      es.pfnCallback = SaveCallback;
// Если расширение файла rtf, файл сохраняется как
// rtf-файл. В противном случае он сохраняется как
// обычный текстовый файл
strupr(&ofn.lpstrFile[ofn.nFileExtension]);
if(!strncmp(&ofn.lpstrFile[ofn.nFileExtension], "RTF", 3))
   SendMessage(hwndEdit, EM_STREAMOUT, SF_RTF, (LPARAM)&es);
else SendMessage(hwndEdit, EM_STREAMOUT, SF_TEXT, (LPARAM)&es);
_lclose(hFile);// Закрываем файл
// Сбрасываем признак изменения содержимого окна  редактора текста
      SendMessage(hwndEdit, EM_SETMODIFY, FALSE, 0L);
    }  }}
//////////////////////////////////////// Функция SaveCallback//////////////////////////////////
DWORD CALLBACK
SaveCallback(DWORD dwCookie, LPBYTE pbBuff, LONG cb, LONG *pcb)
{// Выполняем запись блока данных длиной cb байт
  cb = _lwrite((HFILE)dwCookie, pbBuff, cb);
  *pcb = cb;
  return 0;
}
//////////////////////////////////////////////// Функция FileOpen/////////////////////////////////////
void FileOpen(HWND hwnd)
{ OPENFILENAME ofn;
  char szFile[256];
  char szDirName[256];
  char szFileTitle[256];
  char szFilter[256] = "Rich Text Files\0*.rtf\0Text Files\0*.txt\0" "Any Files\0*.*\0";
  HFILE      hFile;
  OFSTRUCT   of;
  EDITSTREAM es;
```

```
  memset(&ofn, 0, sizeof(OPENFILENAME));
  GetCurrentDirectory(sizeof(szDirName), szDirName);
  szFile[0] = '\0';
// Подготавливаем структуру для выбора входного файла
  ofn.lStructSize     = sizeof(OPENFILENAME);
  ofn.hwndOwner       = hwnd;
  ofn.lpstrFilter     = szFilter;
  ofn.lpstrInitialDir = szDirName;
  ofn.nFilterIndex    = 1;
  ofn.lpstrFile       = szFile;
  ofn.nMaxFile        = sizeof(szFile);
  ofn.lpstrFileTitle  = szFileTitle;
  ofn.nMaxFileTitle   = sizeof(szFileTitle);
  ofn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST;
  // Выводим на экран диалоговую панель, предназначенную
  // для выбора входного файла
  if(GetOpenFileName(&ofn))
  {// Если файл выбран, открываем его для чтения
    if (*ofn.lpstrFile)
    { hFile = OpenFile(ofn.lpstrFile, &of, OF_READ);
      // Устанавливаем параметры функции обратного вызова,
      // которая будет выполнять чтение
      es.dwCookie    = (DWORD)hFile;
      es.dwError     = 0;
      es.pfnCallback = OpenCallback;
      // Если расширение файла rtf, файл загружается как
      // rtf-файл. В противном случае он загружается как
      // обычный текстовый файл
      strupr(&ofn.lpstrFile[ofn.nFileExtension]);
      if(!strncmp(&ofn.lpstrFile[ofn.nFileExtension], "RTF", 3))
        SendMessage(hwndEdit, EM_STREAMIN, SF_RTF, (LPARAM)&es);
      else  SendMessage(hwndEdit, EM_STREAMIN, SF_TEXT, (LPARAM)&es);
      _lclose(hFile);// Закрываем файл
// Сбрасываем признак изменения содержимого окна редактора текста
      SendMessage(hwndEdit, EM_SETMODIFY, FALSE, 0L);
    }  }}
////////////////////////////////////// Функция OpenCallback//////////////////////////////////////
DWORD CALLBACK
OpenCallback(DWORD dwCookie, LPBYTE   pbBuff, LONG cb, LONG *pcb)
{  // Выполняем чтение блока данных длиной cb байт
  *pcb = _lread((HFILE)dwCookie, pbBuff, cb);
  if(*pcb <= 0)  *pcb = 0;
  return 0;}
////////////////////////////////////////////// Функция FilePrint//////////////////////////////////////
void FilePrint(void)
{
```

```
  FORMATRANGE fr;
  DOCINFO docInfo;
  LONG lLastChar, lTextSize;
  PRINTDLG pd;
  int nRc;
  HDC hPrintDC;
// Инициализируем поля структуры PRITDLG
  memset(&pd, 0, sizeof(pd));
  pd.lStructSize = sizeof(PRINTDLG);
  pd.hwndOwner   = hwndEdit;
  pd.hInstance   = (HANDLE)hInst;
  pd.Flags       = PD_RETURNDC | PD_NOPAGENUMS |
    PD_NOSELECTION | PD_PRINTSETUP | PD_ALLPAGES;
  pd.nFromPage   = 0xffff;
  pd.nToPage     = 0xffff;
  pd.nMinPage    = 0;
  pd.nMaxPage    = 0xffff;
  pd.nCopies     = 1;
  // Выводим на экран диалоговую панель, предназначенную
  // для печати документа
  if(PrintDlg(&pd) == TRUE)
  { hPrintDC = pd.hDC;
  // Инициализируем поля структуры FORMATRANGE
   memset(&fr, 0, sizeof(fr));
    // Будем печатать с использованием контекста
    // принтера, полученного от функции PrintDlg
    fr.hdc = fr.hdcTarget = hPrintDC;
    // Печатаем весь документ
    fr.chrg.cpMin = 0;
    fr.chrg.cpMax = -1;
    // Устанавливаем размеры страницы в TWIPS-ах
    fr.rcPage.top = 0;
    fr.rcPage.left = 0;
    fr.rcPage.right  = MulDiv(GetDeviceCaps(hPrintDC, PHYSICALWIDTH),
      1440, GetDeviceCaps(hPrintDC, LOGPIXELSX));
    fr.rcPage.bottom = MulDiv(GetDeviceCaps(hPrintDC, PHYSICALHEIGHT),
      1440, GetDeviceCaps(hPrintDC, LOGPIXELSY));
    fr.rc = fr.rcPage;
    // Оставляем поля
if(fr.rcPage.right > 2*3*1440/4+1440)fr.rc.right -= (fr.rc.left = 3*1440/4);
if(fr.rcPage.bottom > 3*1440) fr.rc.bottom -= (fr.rc.top = 1440);
// Заполняем поля структуры DOCINFO
    memset(&docInfo, 0, sizeof(DOCINFO));
    docInfo.cbSize = sizeof(DOCINFO);
    docInfo.lpszOutput = NULL;
    docInfo.lpszDocName = "RtfPad document";
```

```
    // Начинаем печать документа
    nRc = StartDoc(hPrintDC, &docInfo);
// Если произошла ошибка, получаем и выводим на экран код ошибки
    if(nRc < 0)
    { char szErr[128];
      DWORD dwErr = GetLastError();
      wsprintf(szErr, "Print Error %ld", dwErr);
      MessageBox(NULL, szErr,szAppTitle, MB_OK | MB_ICONEXCLAMATION);
      DeleteDC(hPrintDC);
      return;
    }
    // Начинаем печать страницы
    StartPage(hPrintDC);
    lLastChar = 0;
    // Определяем длину текста в байтах
    lTextSize = SendMessage(hwndEdit, WM_GETTEXTLENGTH, 0, 0);
    // Цикл по всем страницам документа
    while (lLastChar < lTextSize)
    {// Форматируем данные для принтера и печатаем их
lLastChar = SendMessage(hwndEdit, EM_FORMATRANGE, TRUE,(LPARAM) &fr);
if(lLastChar < lTextSize)
     { // Завершаем печать очередной страницы
       EndPage(hPrintDC);
       // Начинаем новую страницу
       StartPage(hPrintDC);
       fr.chrg.cpMin = lLastChar;
       fr.chrg.cpMax = -1;
      }
    }
// Удаляем информацию, которая хранится в органе управления Rich Edit
SendMessage(hwndEdit, EM_FORMATRANGE, TRUE, (LPARAM)NULL);
// Завершаем печать страницы
EndPage(hPrintDC);
// Завершаем печать документа
 EndDoc(hPrintDC);
// Удаляем контекст принтера
DeleteDC(hPrintDC);
  }
}
```

Файл rtfpad.h

```
// Описание функций
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);
BOOL WndProc_OnCreate(HWND hWnd, LPCREATESTRUCT lpCreateStruct);
void WndProc_OnDestroy(HWND hWnd);
```

```
void WndProc_OnCommand(HWND hWnd, int id,   HWND hwndCtl, UINT codeNoti-
fy);
LRESULT WndProc_OnNotify(HWND hWnd, int idFrom, NMHDR FAR* pnmhdr);
void WndProc_OnSize(HWND hwnd, UINT state, int cx, int cy);
void WndProc_OnSetFocus(HWND hwnd, HWND hwndOldFocus);

void FileSaveAs(HWND hwnd);
DWORD CALLBACK SaveCallback(DWORD dwCookie, LPBYTE   pbBuff,
  LONG cb, LONG *pcb);

void FileOpen(HWND hwnd);
DWORD CALLBACK OpenCallback(DWORD dwCookie, LPBYTE   pbBuff,
  LONG cb, LONG *pcb);
void FilePrint(void);

#define IDC_RTFEDIT 1236
```

Файл rtfpad.rc

```
//Microsoft Visual C++ generated resource script.
#include "resource.h"
#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////////////
// Generated from the TEXTINCLUDE 2 resource.
#include "afxres.h"
////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////////////
// Menu
IDR_APPMENU MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New",                    ID_FILE_NEW
        MENUITEM SEPARATOR
        MENUITEM "&Open...",                ID_FILE_OPEN
        MENUITEM "&Save as...",             ID_FILE_SAVEAS
        MENUITEM SEPARATOR
        MENUITEM "&Print...",               ID_FILE_PRINT
        MENUITEM SEPARATOR
        MENUITEM "E&xit",                   ID_FILE_EXIT
    END
    POPUP "&Edit"
    BEGIN
        MENUITEM "&Undo",                   ID_EDIT_UNDO
        MENUITEM SEPARATOR
        MENUITEM "Cu&t",                    ID_EDIT_CUT
        MENUITEM "&Copy",                   ID_EDIT_COPY
```

```
        MENUITEM "&Paste",                      ID_EDIT_PASTE
        MENUITEM "&Delete",                     ID_EDIT_DELETE
        MENUITEM SEPARATOR
        MENUITEM "&Select All",                 ID_EDIT_SELECTALL
    END
    POPUP "&Format"
    BEGIN
        MENUITEM "&Bold",                       ID_FORMAT_BOLD
        MENUITEM "&Italic",                     ID_FORMAT_ITALIC
        MENUITEM "&Underline",                  ID_FORMAT_UNDERLINE
        MENUITEM SEPARATOR
        MENUITEM "&Font...",                    ID_FORMAT_FONT
        MENUITEM SEPARATOR
        POPUP "&Paragraph"
        BEGIN
            MENUITEM "&Left",                   ID_FORMAT_PARAGRAPH_LEFT
            MENUITEM "&Right",                  ID_FORMAT_PARAGRAPH_RIGHT
            MENUITEM "&Center",                 ID_FORMAT_PARAGRAPH_CENTER
        END
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About...",                   ID_HELP_ABOUT
    END
END
#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
// TEXTINCLUDE
1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END
3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

//////////////////////////////////////////////////////////////////////////////
#endif    // APSTUDIO_INVOKED
```

```
//////////////////////////////////////////////////////////////////////////
// Icon
IDI_APPICON             ICON    DISCARDABLE     "rtfpad.ico"
IDI_APPICONSM           ICON    DISCARDABLE     "rtfpadsm.ico"
//////////////////////////////////////////////////////////////////////////
// String Table
STRINGTABLE DISCARDABLE
BEGIN
   ID_FILE_EXIT           "Quits the application"
END
#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////
// Generated from the TEXTINCLUDE 3 resource.
//////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

Файл resource.h

```
//{{NO_DEPENDENCIES}}
// Microsoft Visual C++ generated include file.
// Used by RTFPAD.RC
#define IDR_APPMENU                 102
#define IDI_APPICON                 103
#define IDI_APPICONSM               104
#define ID_FILE_EXIT                40001
#define ID_HELP_ABOUT               40003
#define ID_FORMAT_BOLD              40010
#define ID_FORMAT_ITALIC            40011
#define ID_FORMAT_UNDERLINE         40012
#define ID_FORMAT_FONT              40013
#define ID_FORMAT_PARAGRAPH_LEFT    40014
#define ID_FORMAT_PARAGRAPH_RIGHT   40015
#define ID_FORMAT_PARAGRAPH_CENTER  40016
#define ID_EDIT_DELETE              40021
#define ID_FILE_SAVEAS              40024
#define ID_EDIT_SELECTALL           40028
// Next default values for new objects
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE    121
#define _APS_NEXT_COMMAND_VALUE     40029
#define _APS_NEXT_CONTROL_VALUE     1000
#define _APS_NEXT_SYMED_VALUE       101
#endif
#endif
```

ПРОГРАММА № 6. Вывод bitmap в окна Windows.

ЗАДАНИЕ:
Составить программу, которая иллюстрирует работу знакомого вам классического алгоритма расстановки N не бьющих друг друга ферзей на доске NxN
Придумайте и осуществите программу графической иллюстрации другого вычислительного алгоритма на ваш выбор.
Методическая помощь
В файле ресурсов создайте 3 объекта типа bitmap - черный, белый квадраты и квадрат с изображением ферзя.
Доску нарисуйте в обработчике WM_PAINT, там же вызовите генератор расстановок
- но после анализа флага, взводимого через меню.
В подпрограмму генерации расстановок добавьте функции рисования - квадрата с ферзем при установке его в поле и без при освобождении.
Вот и все - огстальное тривиально.
*/

```
#define STRICT       //Просим коппилятор осуществлять строгую проверку типов
#include <windows.h> //Включить файл интерфейса с библиотекой Windows API
#include <windowsx.h>//То же для связи с расширенной версией Windows API
#include <commctrl.h>
#include <values.h>
#include "afxres.h"
#include "ferzi.h"
#include "resource.h"

// Имя класса окна
char const szClassName[]   = "TWindowClass";
// Заголовок окна
char const szWindowTitle[] =
"Пример Widows-программы, которая расставляет не бьющих друг друга ферзей";

WNDCLASS wc;    // структура для регистрации класса окна
HINSTANCE hInst; //Идентификатор программы

//Прототипы функций
//Начальной инициализации
BOOL InitApp(HINSTANCE,LPSTR);
//Главного окна
LRESULT CALLBACK  WndProc(HWND,//Идентификатор окна, для которого эта ф-ция


//будет получать и обрабатывать сообщения
                  UINT,//Код сообщения
                  WPARAM,//Специфическая для этого сообщения информация
                  LPARAM);
```

```
// Функция WinMain
#pragma argsused //Чтобы не было предупреждений компилятора о неиспользуемых
//аргументах. Другой способ добиться того же - опустить имена аргументов ,
//оставив толтко типы
int WINAPI WinMain(
   HINSTANCE hInstance,   //handle запускаемой программы
   HINSTANCE hPrevInstance,//handle ранее запущенной копии этой же программы-

                                                                        //в
Win32 всегда NULL и не используется (остался
                        //от Win16)
   LPSTR lpCmdLine,        //Указатель на командную строку
   int nCmdShow            // Код состояния окна при запуске программы;
//возможные значения -
SW_HIDE,SW_MINIMIZE,SW_RESTORE,SW_SHOWNORMAL,SW_SHOW,
//SW_SHOWMAXIMIZED,SW_SHOWMINIMIZED,SW_SHOWMINNOACTIVE,SW
_SHOWNA,
//SW_SHOWNOACTIVATE,SW_SHOWNORMAL.
  )
{
 MSG  msg; // структура для работы с сообщениями
 HWND hwnd; // идентификатор главного окна приложения
 //Сохраним в глоб переменной идентификатор приложения
 hInst=hInstance;
 // Инициализируем приложение
 if(!InitApp(hInstance,lpCmdLine)) return FALSE; //Если не удалось - выходим
  // Инициализируем библиотеку стандартных органов управления
 InitCommonControls();
 // После успешной инициализации приложения создаем главное окно
 hwnd = CreateWindow(
   szClassName,        // имя класса окна
   szWindowTitle,      // заголовок окна
   WS_OVERLAPPEDWINDOW, // стиль окна перекрываемое (другие возможные
значения

                                          //посмотрите в Help-системе)
   CW_USEDEFAULT,      //горизонтальная позиция окна (у нас - по умолчанию)
   CW_USEDEFAULT,      //вертикальная позиция окна
   CW_USEDEFAULT,      //ширина окна
   CW_USEDEFAULT,      //высота окна
   0,                  //идентификатор родительского окна
   0,                  // идентификатор меню или дочернего окна
   hInst,              //идентификатор программы
   NULL);              //указатель на дополнительные данные
 // Если создать окно не удалось, завершаем приложение
```

```
  if(!hwnd)    return FALSE;
  // Рисуем главное окно
  ShowWindow(hwnd, nCmdShow  );
  UpdateWindow(hwnd);
 //Запускаем цикл обработки сообщений - его простейший вариант
  while(GetMessage(      //Функция выборки сообщения из очереди
                &msg,//Адрес структуры с сообщением
                0,   //Идентификатор окна, котрому надо переадресовать сообщ
                0,   //Начало диапазона отбираемых (фильтруемых) сообщений
                0))  //Конец диапазона отбираемых (фильтруемых) сообщений
  {
    DispatchMessage(&msg);//Ф-ция распределяет сообщение нужной оконной
функции
    TranslateMessage(&msg);
  }
  return msg.wParam;
}

////////////////////////////// Выполняет регистрацию класса окна//////////////////////
BOOL InitApp(HINSTANCE hInstance,LPSTR lpcl)
{
hbitmap[0]=LoadBitmap(hInst,MAKEINTRESOURCE(IDB_BITMAP1)),
hbitmap[1]=LoadBitmap(hInst,MAKEINTRESOURCE(IDB_BITMAP2)),
hbitmap[2]=LoadBitmap(hInst,MAKEINTRESOURCE(IDB_BITMAP3));
N=atoi(lpcl);
if(N<4){
MessageBox(NULL,"Размер доски должен быть больше 3 ",
"Ошибочен аргумент ком строки",MB_OK | MB_ICONEXCLAMATION);
return FALSE;}
ND=2*N-1;                  //Количество диагоналей
//Выделим память под все массивы сразу как под массив результата
F=(USHORT *)malloc((2*N+2*ND)*sizeof(unsigned));
if(F==NULL)
{MessageBox(NULL,"Что-то с памятью моей стало ",
  "Ошибка выделения памяти",MB_OK | MB_ICONEXCLAMATION);
  return FALSE;}
//Определим указатели на другие массивы в выделенной памяти
COL=F+N;DS=COL+N;DR=DS+ND;
//"Объединичим" все массивы сразу -т.е. обозначим свободными все поля
for(int i=0;i<N+2*ND;i++) COL[i]=1;
//Проверяем, не было ли это приложение запущено ранее
  HWND hWnd = FindWindow(szClassName, NULL);
  if(hWnd)
  {    if(IsIconic(hWnd))  ShowWindow(hWnd, SW_RESTORE);
                           SetForegroundWindow(hWnd);
    return FALSE;
```

```
 }
 ATOM aWndClass; // атом для кода возврата
 WNDCLASS wc;    // структура для регистрации
 wc.lpszMenuName  = MAKEINTRESOURCE(IDM_MENU1);//Имя меню
 wc.style         = CS_HREDRAW | CS_VREDRAW;//Стиль класса окна
 wc.lpfnWndProc   = (WNDPROC) WndProc;      //Указатель на оконную функцию
 wc.cbClsExtra    = 0;                      //Дополнит данные для класса окна
 wc.cbWndExtra    = 0;                      //Дополнит данные для окна
 wc.hInstance     = hInstance;              //Идентификатор программы
 wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);//Иконка для программы
 wc.hCursor       = LoadCursor(NULL, IDC_ARROW);    //Курсор мышки
 wc.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);//Кисть для фона
 wc.lpszClassName = (LPSTR)szClassName;     //Имя класса окна
 aWndClass = RegisterClass(&wc);            //Библиот ф-ция регистрации в ОС
 return (aWndClass != 0);}

////////////////////////////////////Оконная функция/////////////////////////////////////////////////////////
LRESULT CALLBACK WndProc( HWND hWnd,
                          UINT msg, //Код сообщения
                          WPARAM wParam,//Специфическая для сообщения информация
                          LPARAM lParam)
{switch (msg)
  {
   HANDLE_MSG(hWnd, WM_CREATE,     WndProc_OnCreate);
   HANDLE_MSG(hWnd, WM_DESTROY,    WndProc_OnDestroy);
   HANDLE_MSG(hWnd, WM_COMMAND,    WndProc_OnCommand);
   HANDLE_MSG(hWnd, WM_PAINT,      WndProc_OnPaint);
   HANDLE_MSG(hWnd, WM_SIZE,       WndProc_OnSize);
  default:return(DefWindowProc(hWnd, msg, (UINT)wParam, lParam));
    }
}
////////////////////////////////////////////WM_CREATE/////////////////////////////////////////////////
BOOL WndProc_OnCreate(HWND hWnd, LPCREATESTRUCT lpCreateStruct)
{ return TRUE;}
////////////////////////////////////////////WM_DESTROY/////////////////////////////////////////////////
void WndProc_OnDestroy(HWND hWnd)
{PostQuitMessage(0);
return ;//FORWARD_WM_DESTROY(hWnd, DefWindowProc);
}
////////////////////////////////////////////WM_COMMAND/////////////////////////////////////////////////
void WndProc_OnCommand(HWND hWnd, int id,HWND hwndCtl, UINT codeNotify)
{switch(id)
{case CM_WORK1:
{FlagMenu=1;
InvalidateRect(hWnd,NULL,TRUE);}
default:break;
```

```
}}

//////////////////////////////////////////WM_SIZE/////////////////////////////////////////////////
void WndProc_OnSize(HWND hwnd, UINT state, int cx, int cy)
{cxClient = cx ; cyClient = cy;InvalidateRect(hwnd,NULL,TRUE);}
////////////////////////////Функция  рисования  -WM_PAINT//////////////////////////////////
void WndProc_OnPaint(HWND hWnd)
{short i,j;PAINTSTRUCT ps;
hdc=BeginPaint(hWnd,&ps);
//Создаем совместимый контекст памяти для каждого квадрата
for(i=0;i<3;i++) hMemDC[i]=CreateCompatibleDC(hdc);
//Выбираем bitmap в контексты
for(i=0;i<3;i++) hOldbm[i]=(HBITMAP)SelectObject(hMemDC[i],hbitmap[i]);
//Режимы отображения
for(i=0;i<3;i++)SetMapMode(hMemDC[i],GetMapMode(hdc));
//Преобразования в логические координаты
GetObject(hbitmap[0],sizeof(BITMAP),(LPSTR)&bm);
ptSize.x=bm.bmWidth; ptSize.y=bm.bmHeight;
DPtoLP(hdc,&ptSize,1);ptOrg.x=0;ptOrg.y=0;
for(i=0;i<3;i++)DPtoLP(hMemDC[i],&ptSize,1);
//Рисуем доску заданного размера в левом верхнем углу окна
for(i=0;i<N;i++)
for(j=0;j<N;j++)
{//BitBlt(hdc,j*ptSize.x/2,i*ptSize.y/2,ptSize.x/2,ptSize.y/2,hMemDC[(j+(i+1))&1],i*ptO
rg.x//,j*ptOrg.y,SRCCOPY);
StretchBlt(hdc,j*cxClient/(2*N),i*cxClient/(2*N),cxClient/(2*N),cxClient/(2*N),
hMemDC[(j+(i+1))&1],0,0,ptSize.x,ptSize.y,SRCCOPY);}
//Вызов алгоритма  генерации - только по требованию через меню
if(FlagMenu){generation(hdc);FlagMenu=0;}
//Возвращаем все на свои места
for(i=0;i<3;i++)SelectObject(hMemDC[i],hOldbm[i]);
for(i=0;i<3;i++)DeleteDC(hMemDC[i]);
EndPaint(hWnd,&ps);}

/////////////////Функция генерации расстановок//////////////////////////////
void generation(HDC hdc)
{USHORT col=0;int Click;
do{//Если поле[row][col] свободно
if(isfree(row,col))
//займем его занеся номер столбца в массив расстановки и
//обозначив нулями занятость столбца и диагоналей
{setfild(row,col);
//Рисуем в занятой клетке ферзя
StretchBlt(hdc,col*cxClient/(2*N),row*cxClient/(2*N),cxClient/(2*N),cxClient/(2*N),
hMemDC[2],0,0,ptSize.x,ptSize.y,SRCCOPY);
Click=0;
```

```
while(Click<667770000) Click++;//просто задержка - более грамотно - по времени
row++;
//Если все строки заняты
if(row==N)
//печатаем очередную конфигурацию искомых элементов
{number++;printconf(hdc);}
/*иначе рекурсивно вызываем саму себя, но уже с новой исследуемой строкой */
else generation(hdc);
//Возвращаемся на строку назад для исследования в ней следующего столбца
row--;
//Освобождаем ранее занятое в ней поле[row][col]
freefild(row,col);
//Рисуем освобождение поля - просто соответствующий квадрат без ферзя
StretchBlt(hdc,col*cxClient/(2*N),row*cxClient/(2*N),cxClient/(2*N),cxClient/(2*N),
hMemDC[(col+row+1)&1],0,0,ptSize.x,ptSize.y,SRCCOPY);
}//if isfree
//Исследуем следующий столбец в строке row
col++;
}while(col<N);}
//////////////////////////////////Функция печати очередной расстановки//////////////////////
void printconf(HDC hdc)
{int i;char s[32],s1[8];TEXTMETRIC tm;
GetTextMetrics(hdc,&tm);
sprintf(s,"Номер расстановки:%-6d",number);
for(i = 0;i<N; i++)
{sprintf(s1," %d",F[i]);strcat(s,s1);}
TextOut(hdc,(cxClient*0.6),number*tm.tmHeight,s,strlen(s));
}
```

Файл fetzi.h.

```
#ifndef __FERZI_H
#define __FERZI_H
#include <commctrl.h>
#include <stdio.h>
#include "ferzi.h"
int cxClient,cyClient,FlagMenu;
USHORT * F,              //Адрес массива для хранения варианта расстановки
*COL,                    //то же для индикации занятости столбца
*DS,                     //для индикации занятости диагонали, у которой
                         //постоянна сумма индексов столбцов и строк
*DR,                     //то же для постоянной разности индек
N,                       //для размерности матрицы
ND,                      //для количества диагоналей
row,                     //для текущего индекса строки
number;
BITMAP bm;
POINT ptSize,ptOrg;
```

```
HDC hdc,hMemDC[3];
HBITMAP hOldbm[3],hbitmap[3];
//Прототипы функций обработчиков сообщений
BOOL WndProc_OnCreate(HWND hWnd, LPCREATESTRUCT lpCreateStruct);
void WndProc_OnDestroy(HWND hWnd);
BOOL WndProc_OnChar(HWND hwnd, TCHAR ch, int rpt);
BOOL WndProc_OnLbuttonDown(HWND hwnd, BOOL fDblClk, int x, int y, UINT
keyFlags);
void WndProc_OnCommand(HWND hWnd, int id,HWND hwndCtl, UINT codeNotify);
void WndProc_OnSize(HWND hwnd, UINT state, int cx, int cy);
void WndProc_OnMove(HWND hwnd,int cx, int cy);
void WndProc_OnPaint(HWND hwnd);
void generation(HDC hdc);
void printconf(HDC hdc);
/*Нам будет удобно составить предварительно несколько вспомогательных
подпрограмм - это улучшит читабельность программы */
///////////////Функция, определяющая, свободно ли поле для установки///////
int isfree(unsigned row, unsigned col)
{if(COL[col] && DS[row+col] && DR[N-1+row-col]) return 1;
else return 0;}
//////////////////////////////////Функция, занимающая поле//////////////////////////////
void setfild(unsigned row,unsigned col)
{F[row]=col;COL[col]=0;DS[row+col]=0;DR[N-1+row-col]=0;}
//Функция, освобождающая поле
void freefild(unsigned row,unsigned col)
{COL[col]=1;DS[row+col]=1;DR[N-1+row-col]=1;}
#endif
```

ПРОГРАММА № 7. Работа с контейнерами в среде Windows.

ЗАДАНИЕ:

Составить программу, которая реализует основные функции работы с контейнерами-
пополнение, просмотр, поиск объектов, удаление, сохранение в файлах и заполнение из файлов. В качестве основы можете взять любой из контейнеров, которые вы создавали в консольном текстовом режиме (контейнер студентов с вложенным контейнером сданных работ). В приведенном примере контейнеры – простые массивы.

Работу с файлами организуйте через стандартные файловые диалоги .

МЕТОДИЧЕСКАЯ ПОМОЩЬ

Основным отличием Windows- реализации будет наличие диалоговых панелей для ввода данных и извлечение информации из редактируемых полей диалогов. Диалоги создайте с помощью редактора ресурсов.

В качестве методической помощи приводим откомментированный вариант программы,
который вам предлагается усовершенствовать,дополнить и видоизменить по вашему усмотрению.

```
*/
```

Файл wmainarray.cpp.

```
#define STRICT      //Просим коппилятор осуществлять строгую проверку типов
#include <windows.h> //Включить файл интерфейса с библиотекой Windows API
#include <windowsx.h>//То же для связи с расширенной версией Windows API
#include <commctrl.h>
#include <cstring.h>
#include "afxres.h"
#include "warray.hpp"
//Создаем  пока неинициализированный указатель на контейнерный объект
//сведений о студентах
stArray* listStud=NULL;
int FlagExit;       //Флаг желания прекратить работу
int stCnt;          //Количество студентов в контейнере
BOOL NewStud;
int iStud,iPrg,  cStud, cPrg;
char buf[128];
//Переменные для приема данных из полей ввода
char Group[16], StudName[32], ProgName[128];
int StudNumber,  ProgNumber,  ProgCnt,  err;
Student* st=NULL;  //Указатель на создаваемого в куче студента
cpArray* cpa;          //Указатель на контейнер программ
OPENFILENAME ofn;  //структура для выбора файла через стандартный диалог
char szFile[256];         //буфер для записи пути к выбранному файлу
char szFileTitle[256];   //буфер для записи имени  выбранного файла
//фильтр расширений имени файлов
char szFilter[256]="DataFiles\0*.dat;\0Any fILES\0*.*\0\0";
// Имя класса окна
char const szClassName[]   = "ContainerWindowClass";
// Заголовок окна
char const szWindowTitle[] =
"Пример Widows-программы, которая работает с контейнерами";
WNDCLASS wc;    // структура для регистрации класса окна
HINSTANCE hInst; //Идентификатор программы

//////////////////////////////////////Прототипы функций//////////////////////////////////
//Формиирования структуры для работы с файлми
void StructFile(void);
//Начальной инициализации  окна и программы
BOOL InitApp(HINSTANCE);
//Главного окна
LRESULT CALLBACK  WndProc(HWND,
                    UINT,       //Код сообщения
                    WPARAM,//Специфическая для этого сообщения информация
```

```
                LPARAM);


//Диалога при добавке студента
BOOL CALLBACK DlgAddStud(HWND hdlg,UINT msg,WPARAM wParam,LPARAM
lParam);
//Диалога при просмотре контейнера
BOOL CALLBACK DlgViewStud(HWND hdlg,UINT msg,WPARAM
wParam,LPARAM lParam);
//Диалога при поиске объекта
BOOL CALLBACK DlgSearchStud(HWND hdlg,UINT msg,WPARAM
wParam,LPARAM lParam);
//Диалога при добавлении работы стауденту
BOOL CALLBACK DlgAddProg(HWND hdlg,UINT msg,WPARAM wParam,LPARAM
lParam);
////////////////////////////////////////// Функция WinMain//////////////////////////////////
int WINAPI WinMain(
   HINSTANCE hInstance,   //handle запускаемой программы
   HINSTANCE hPrevInstance,
   LPSTR lpCmdLine,          //Указатель на командную строку
   int nCmdShow              // Код состояния окна при запуске программы;
  )
{
 listStud=new stArray(0,0,1);//Сразу создадим массив студентов
 MSG  msg; // структура для работы с сообщениями
 HWND hwnd; // идентификатор главного окна приложения
 //Сохраним в глоб переменной идентификатор приложения
 hInst=hInstance;
 // Инициализируем приложение
 if(!InitApp(hInstance)) return FALSE; //Если не удалось - выходим
  // Инициализируем библиотеку стандартных органов управления
 InitCommonControls();
 // После успешной инициализации приложения создаем главное окно
 hwnd = CreateWindow(
   szClassName,        // имя класса окна
   szWindowTitle,      // заголовок окна
   WS_OVERLAPPEDWINDOW,
   CW_USEDEFAULT,      //горизонтальная позиция окна (у нас - по умолчанию)
   CW_USEDEFAULT,      //вертикальная позиция окна
   CW_USEDEFAULT,      //ширина окна
   CW_USEDEFAULT,      //высота окна
   0,                  //идентификатор родительского окна
   0,                  // идентификатор меню или дочернего окна
   hInst,              //идентификатор программы
   NULL);              //указатель на дополнительные данные
 // Если создать окно не удалось, завершаем приложение
```

```
 if(!hwnd)    return FALSE;
 // Рисуем главное окно
 ShowWindow(hwnd, nCmdShow  );  UpdateWindow(hwnd);
//Запускаем цикл обработки сообщений - его простейший вариант
 while(GetMessage(      //Функция выборки сообщения из очереди
              &msg,//Адрес структуры с сообщением
              0,   //Идентификатор окна, котрому надо переадресовать сообщ
              0,   //Начало диапазона отбираемых (фильтруемых) сообщений
              0))  //Конец диапазона отбираемых (фильтруемых) сообщений
 {
   DispatchMessage(&msg);//Ф-ция распределяет сообщение нужной оконной
функции
   TranslateMessage(&msg);
 }
 delete listStud;
 return msg.wParam;
}
////////////////////////////////////////////// Функция InitApp//////////////////////////////////////////
#include "resource.h"
BOOL  InitApp(HINSTANCE hInstance)
{WNDCLASS wc;    // структура для регистрации
// Преверяем, не было ли это приложение запущено ранее
 HWND hWnd = FindWindow(szClassName, NULL);
 if(hWnd)
 {  if(IsIconic(hWnd))
ShowWindow(hWnd, SW_RESTORE);etForegroundWindow(hWnd);
   return FALSE;  }
 wc.lpszMenuName  = MAKEINTRESOURCE(IDM_MENU1);//Имя меню
 wc.style         = CS_HREDRAW | CS_VREDRAW;//Стиль класса окна
 wc.lpfnWndProc   = (WNDPROC) WndProc;      //Указатель на оконную функцию
 wc.cbClsExtra    = 0;                  //Дополнит данные для класса окна
 wc.cbWndExtra    = 0;                   //Дополнит данные для окна
 wc.hInstance     = hInstance;            //Идентификатор программы
 wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);//Иконка для программы
 wc.hCursor       = LoadCursor(NULL, IDC_ARROW);    //Курсор мышки
 wc.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);//Кисть для фона
 wc.lpszClassName = (LPSTR)szClassName;     //Имя класса окна
 return RegisterClass(&wc);           //Библиот ф-ция регистрации в ОС
}
//Пусть 2 пары переменных хранят размеры клиентской области окна и его позицию
int cwClient,chClient,xPos,yPos;
LRESULT CALLBACK WndProc( HWND hWnd,
                  UINT msg, //Код сообщения
                  WPARAM wParam,//Специфическая для сообщения информация
                  LPARAM lParam)
{switch (msg)
```

```
  {
   HANDLE_MSG(hWnd, WM_CREATE,WndProc_OnCreate);
   HANDLE_MSG(hWnd, WM_DESTROY,WndProc_OnDestroy);
   HANDLE_MSG(hWnd, WM_NOTIFY,WndProc_OnNotify);
   HANDLE_MSG(hWnd, WM_SIZE,WndProc_OnSize);
   HANDLE_MSG(hWnd, WM_MOVE,WndProc_OnMove);
   HANDLE_MSG(hWnd, WM_MENUSELECT,WndProc_OnMenuSelect);
   HANDLE_MSG(hWnd, WM_COMMAND,WndProc_OnCommand);
   default:
   return(DefWindowProc(hWnd, msg, wParam, lParam));
   }}
//////////////////////////Действия выполняемые при создании окна//////////////////////
BOOL WndProc_OnCreate(HWND hWnd, LPCREATESTRUCT lpCreateStruct)
{// Создаем Toolbar
  hwndTb = CreateToolbarEx(hWnd,WS_CHILD | WS_BORDER | WS_VISIBLE |
TBSTYLE_TOOLTIPS |    CCS_ADJUSTABLE,
   IDT_TOOLBAR,           // идентификатор органа Toolbar
   5,                     // количество пиктограмм
   hInst,                 // идентификатор приложения
   IDB_TBBITMAP,          // идентификатор битового изображения
                    //   кнопок
   (LPCTBBUTTON)&tbButtons,// адрес описания кнопок
   6,                     // количество кнопок
   16,16,                 // ширина и высота кнопок
   16,16,                 // ширина и высота пиктограмм
   sizeof(TBBUTTON));     // размер структуры в байтах
   if(hwndTb == NULL)    return FALSE;
   //hwndSb=CreateStatusWindow(1L,"Status string",hWnd,IDS_STATUSBAR);
     hwndSb = CreateWindowEx(
   0L,                    // расширенный стиль окна
   STATUSCLASSNAME,       // класс окна для Statusbar
   "",                    // заголовок окна отсутствует
   WS_CHILD | WS_BORDER |  // стиль окна
   WS_VISIBLE | SBARS_SIZEGRIP,
   0, 0, 0, 0,            // координаты, ширина, высота
   hWnd,                  // идентификатор родительского окна
   (HMENU)IDS_STATUSBAR,   // идентификатор Statusbar
   hInst,                 // идентификатор приложения
   NULL );                // доп. данные для окна
   if(hwndSb == NULL)    return FALSE;
   return TRUE;}
//////////////////////////////////////WM_DESTROY/////////////////////////////////////////////////////
void WndProc_OnDestroy(HWND hWnd)
{
PostQuitMessage(0);
// FORWARD_WM_DESTROY(hWnd, DefWindowProc);
```

```
}
//////////////////////////////Подпрограмма - Обработчик команд//////////////////////
void WndProc_OnCommand(HWND hWnd,int id,HWND hwndCtl, UINT codeNotify)
{///////////////Переключатель в зависимости от кода команды//////////////////
  switch (id)
  {///////////////Команда завершения программы//////////////////////
case CM_EXIT: PostQuitMessage(0); break;
//////////////////////////Команда ввода нового студента//////////////////
case CM_ADDSTUD:
{//Вызываем панель диалога для ввода данных о студентах
DialogBox(hInst,MAKEINTRESOURCE(IDD_DIALOG1),hWnd,DlgAddStud);
break; }
//////////////////////////Команда сохранения контейнера студентов///////////
case CM_SAVEAS:
{StructFile();     //Заполним структуру для работы с файлом
if(GetSaveFileName(&ofn))//Если пользователь выбрал файл
{//открываем выбранный файл для записи
ofstream fout(ofn.lpstrFile);
int cnt=(*listStud).GetItemsInContainer();
fout<<cnt<<'\n';
for(int i=0;i<cnt;i++) fout<<(*listStud)[i]<<'\n';
fout.close();
}
break;}
//////////////////////////Начальная инициализация контейнера из файла//////////
case CM_OPEN:
{//Если контейнер еще не создан - создадим
if(listStud==NULL) listStud=new stArray(0,0,1);
StructFile();     //Заполним структуру для работы с файлом
if(GetOpenFileName(&ofn))//Если пользователь выбрал файл
{int cnt,i;
//открываем выбранный файл для чтения
ifstream fin(ofn.lpstrFile);
fin>>cnt;              //Читаем количество студентов
for(int i=0;i<cnt;i++)
{st=new Student(fin);(*listStud).Add(*st);}
fin.close();
}
break;}
////////////////////////////////////////Просмотр контейнера//////////////////////////////
case CM_VIEW:
{cStud=listStud->GetItemsInContainer();//Колич студентов
if(cStud==0)
{MessageBox(NULL,"Контейнер пуст - нечего смотреть",
"Для сведения",MB_OK | MB_ICONINFORMATION);
break;}
```

```
//Вызываем панель диалога для ввода данных о студентах
DialogBox(hInst,MAKEINTRESOURCE(IDD_DIALOG1),hWnd,DlgViewStud);
break;}

//////////////////////Поиск данных о студенте//////////////////////////
case CM_SEARCH:
{cStud=listStud->GetItemsInContainer();//Колич студентов
if(cStud==0)
{MessageBox(NULL,"Контейнер пуст - нечего искать черную кошку в темной
комнате\
если ее там нет ","Для сведения",MB_OK | MB_ICONINFORMATION);
break;}
//Вызываем панель диалога для поска данных о студентах
DialogBox(hInst,MAKEINTRESOURCE(IDD_DIALOG1),hWnd,DlgSearchStud);
break;}

////////////////////////////////////Добавить программу студенту//////////////////////////
case CM_ADDPROG:
{cStud=listStud->GetItemsInContainer();//Колич студентов
if(cStud==0)
{MessageBox(NULL,"Контейнер пуст - некому добавлять ",
"Для сведения",MB_OK | MB_ICONINFORMATION);
break;}
//Вызываем панель диалога для поска данных о студентах
DialogBox(hInst,MAKEINTRESOURCE(IDD_DIALOG1),hWnd,DlgAddProg);
break;
}
////////////////////////////////////Команда меню выбора шрифта/////////////////////////////////
case CM_SELECTFONT:{
MessageBox(NULL, "Команда CM_SELECTFONNT выбра шрифта", "Шаблон Win-
dows - программы", MB_OK);
break;}
default:return;// FORWARD_WM_COMMAND(hWnd, id, hwndCtl, codeNotify, Def-
WindowProc);
  }}
////////////////////////////// Функция WndProc_OnNotify//////////////////////////////////////////
LRESULT WndProc_OnNotify(HWND hWnd, int idFrom, NMHDR* pnmhdr)
{
 LPTOOLTIPTEXT lpToolTipText;
 LPTBNOTIFY lptbn;
 int nItem;
 static CHAR szBuf[128];
 switch(pnmhdr->code)
 {// Если получили сообщение от ToolTips, загружаем из ресурсов
   // соответствующую текстовую строку
   case TTN_NEEDTEXT:
```

```
      lpToolTipText = (LPTOOLTIPTEXT)pnmhdr;
       LoadString(hInst, lpToolTipText->hdr.idFrom,   szBuf, sizeof(szBuf));
       lpToolTipText->lpszText = szBuf;
       break;

    // Возвращаем окну Toolbar характеристики кнопки, с номером в lptbn->iItem
    case TBN_GETBUTTONINFO:
      lptbn = (LPTBNOTIFY)pnmhdr;  nItem = lptbn->iItem;
      lptbn->tbButton.iBitmap   = tbButtons[nItem].iBitmap;
      lptbn->tbButton.idCommand = tbButtons[nItem].idCommand;
      lptbn->tbButton.fsState   = tbButtons[nItem].fsState;
      lptbn->tbButton.fsStyle   = tbButtons[nItem].fsStyle;
      lptbn->tbButton.dwData    = tbButtons[nItem].dwData;
      lptbn->tbButton.iString   = tbButtons[nItem].iString;
// Если запрашиваются характеристики несуществующей
// кнопки, возвращаем FALSE
return ((nItem < sizeof(tbButtons)/sizeof(tbButtons[0]))?TRUE : FALSE);
// Разрешаем удаление любой кнопки, кроме самой первой
 case TBN_QUERYDELETE:
lptbn = (LPTBNOTIFY)pnmhdr;  nItem = lptbn->iItem;
return (nItem == 0)? FALSE : TRUE;
// Разрешаем вставку любой кнопки, кроме самой первой
case TBN_QUERYINSERT:
lptbn = (LPTBNOTIFY)pnmhdr; nItem = lptbn->iItem; return (nItem == 0)? FALSE :
TRUE;
// В ответ на завершение операции перемещения перерисовываем Toolbar
case TBN_TOOLBARCHANGE:
SendMessage(hwndTb, TB_AUTOSIZE, 0L, 0L);
return TRUE;
default: break;  }
 return FALSE;}
////////////////////////////////////Изменились размеры окна/////////////////
//w,h-новые размеры окна
void WndProc_OnSize(HWND hwnd, UINT state, int w, int h)
{return;// FORWARD_WM_SIZE(hwnd, state, w, h, DefWindowProc);}
////////////////////////////Измененилась позиция окна//////////////////////
//cx,cy-новые координаты окна
void WndProc_OnMove(HWND hwnd, int cx, int cy)
{return;// FORWARD_WM_MOVE(hwnd, cx, cy, DefWindowProc);}
//////////////////////Обработка сообщения о помеченной позиции меню
void WndProc_OnMenuSelect(HWND hwnd, HMENU hmenu, int item,HMENU
hmenuPopup,
UINT flags)
{
static char szBuf[128]; szBuf[0] = 0;
// Загружаем строку из ресурсов приложения
```

```
LoadString(hInst,item, szBuf, sizeof(szBuf));
// Отображаем строку в первой области Toolbar
SendMessage(hwndSb, SB_SETTEXT, 0, (LPARAM)szBuf);
return;// FORWARD_WM_MENUSELECT(hwnd, hmenu, item,hmenuPopup, flags,
DefWindowProc);}


//////////////////////////Функция диалога для ввода данных о новом студенте////////////////
BOOL CALLBACK DlgAddStud(HWND hdlg,UINT msg,WPARAM wParam,LPARAM
lParam)
{//Сюда мы попадаем при посылке пользователем любого сообщения
//из панели диалога
switch (msg)
{////////////////////Инициализация окна диалога///////////////
case WM_INITDIALOG:
{st=NULL;  NewStud=TRUE;SetWindowText(hdlg,"Ввод данных для нового
студента");
return TRUE;}
/////////////////////////////////////Обработка команд от окна диалога//////////////////
case WM_COMMAND:
switch(wParam)
{//////////////////////////////////////Прекращение ввода////////////////////////////////
case IDCANCEL:
{EndDialog(hdlg,0); return TRUE; }
////////////////////Конец ввода в окне диалога о новом студенте//////////////////////
case IDC_FINI://Пользователь закончил ввод
{//Если вводится новый студент
if(NewStud)
{//Получаем сведения о студенте
GetDlgItemText(hdlg,IDC_EGROUP,Group,16);//Группу
StudNumber=GetDlgItemInt(hdlg,IDC_ESTUDNUMBER,&err,FALSE);//Номер
студента GetDlgItemText(hdlg,IDC_ESTUDNAME,StudName,32);
//Фамилию
ProgCnt=GetDlgItemInt(hdlg,IDC_EWORKCNT,&err,FALSE);//Колич вводимых
работ
//Если о студенте не все сказано
if(strlen(Group)==0 || strlen(StudName)==0 || StudNumber==0)
{MessageBox(NULL,"Не введена группа или фамилия студента","Ошибка
ввода",MB_OK | MB_ICONEXCLAMATION);
return TRUE;
}
string sg(Group),sn(StudName);
//Конструируем его
if(NewStud ) st=new Student(sg,StudNumber,sn);
//Читаем сведения об  очередной работе
ProgNumber=GetDlgItemInt(hdlg,IDC_EPROGNUMBER,&err,FALSE);//№ программы
```

```
GetDlgItemText(hdlg,IDC_EPROGNAME,ProgName,128);//Название программы
//Если они есть-добавим в массив программ студента
if((ProgNumber!=0) && (strlen(ProgName)!=0))
{string sp(ProgName);
ComputerProgram cp=*new ComputerProgram(ProgNumber,sp);
(st->lstPrg())->Add(cp); st->ProgCnt+=1;
//Корректируем поле количества программ
SetDlgItemInt(hdlg,IDC_EWORKCNT,st->ProgCnt,FALSE);
//Добавляем студента в контейнер
(*listStud).Add(*st);++stCnt;}}
//Если студент тот же
else{//Читаем сведения об  очередной работе
ProgNumber=GetDlgItemInt(hdlg,IDC_EPROGNUMBER,&err,FALSE);/№ программы
GetDlgItemText(hdlg,IDC_EPROGNAME,ProgName,128);//Название программы
//Если они есть
if((ProgNumber!=0) && (strlen(ProgName)!=0))
{//Ищем студента в массиве.Это специфика прямого хранения - объект копируется
//в контейнер и адрес копии отличается от адреса оригинала
int index=(*listStud).Find(*st);
if(index== INT_MAX)
{MessageBox(NULL,"Не найден  студент","Ошибка ",MB_OK |
MB_ICONEXCLAMATION);return FALSE;}
//Если студент найден
string sp(ProgName); ComputerProgram cp=*new ComputerProgram(ProgNumber,sp);
(*listStud)[index].lstPrg()->Add(cp);
SetDlgItemInt(hdlg,IDC_EWORKCNT,++(*listStud)[index].ProgCnt,FALSE);
}//if
}//else
break;}
//////////////////////////////Кнопка “Следующий студент”//////////////////////////////////////
case IDC_NEXTSTUD:
{NewStud=TRUE;
MessageBox(NULL,"Введите группу,номер,фамилию студента,номер и название
программы", "Порядок ввода",MB_OK | MB_ICONINFORMATION);
//Очистим все поля
SetDlgItemText(hdlg,IDC_EGROUP,"");
SetDlgItemText(hdlg,IDC_ESTUDNUMBER,"");
SetDlgItemText(hdlg,IDC_ESTUDNAME,"");
SetDlgItemText(hdlg,IDC_EWORKCNT,"");
SetDlgItemText(hdlg,IDC_EPROGNUMBER,"");
SetDlgItemText(hdlg,IDC_EPROGNAME,"");
break;}
//////////////////////////////Кнопка “Следующая работа”//////////////////////////////////////
case IDC_NEXTWORK:
{NewStud=FALSE;
MessageBox(NULL,"Введите номер и название программы",
```

```
"Порядок ввода",MB_OK | MB_ICONINFORMATION);
//Очистим поля работы
SetDlgItemText(hdlg,IDC_EPROGNUMBER,"");
SetDlgItemText(hdlg,IDC_EPROGNAME,"");
break;}
}//switch wparam
}//switch msg
return FALSE;}

//////////////////////////////////////////////Функция просмотра//////////////////////////////
BOOL CALLBACK DlgViewStud(HWND hdlg,UINT msg,WPARAM
wParam,LPARAM lParam)
{//Сюда мы попадаем при посылке пользователем любого сообщения
//из панели диалога
//////////////////////////////////////В звисимости от кода сообщения//////////////////
switch (msg)
{////////////////////////////////////Инициализация окна диалога///////////////
case WM_INITDIALOG:
{iStud=0;iPrg=0;
cpa=(*listStud)[iStud].lstPrg();//Массив программ i-го студента
cPrg=cpa->GetItemsInContainer();
//Соответствующий заголовок окна
SetWindowText(hdlg,"Просмотр содержимого контейнера");
//Заполняем все  поля
strcpy(buf,(*listStud)[iStud].Group().c_str());//Копир с преобр string в char*
SetDlgItemText(hdlg,IDC_EGROUP,buf);
SetDlgItemInt(hdlg,IDC_ESTUDNUMBER,(*listStud)[iStud].Number(),FALSE);
strcpy(buf,(*listStud)[iStud].Name().c_str());
SetDlgItemText(hdlg,IDC_ESTUDNAME,buf);
SetDlgItemInt(hdlg,IDC_EWORKCNT,(*listStud)[iStud].GetProgCnt(),FALSE);
SetDlgItemInt(hdlg,IDC_EPROGNUMBER,(*cpa)[iPrg].PrgNumber(),FALSE);
strcpy(buf,(*cpa)[iPrg].PrgName().c_str());
SetDlgItemText(hdlg,IDC_EPROGNAME,buf);
return TRUE;}
////////////////////////////////////Обработка команд от окна диалога/////////////////
case WM_COMMAND:
switch(wParam)
{///////////////Прекращение ввода//////////////////////////////
case IDCANCEL:
{EndDialog(hdlg,0);return TRUE;}
//////////////////////////////////////////////Следующий студент//////////////////////////////
case IDC_NEXTSTUD:
{//Если можно - наращиваем индекс студента для след. вывода
//определяем параметры его работ и выводим
if(iStud<(cStud-1))
{iStud++; iPrg=0;
```

```
cpa=(*listStud)[iStud].lstPrg();//Массив программ i-го студента
cPrg=cpa->GetItemsInContainer();
//Заполняем все  поля
strcpy(buf,(*listStud)[iStud].Group().c_str());//Копир с преобр string в char*
SetDlgItemText(hdlg,IDC_EGROUP,buf);
SetDlgItemInt(hdlg,IDC_ESTUDNUMBER,(*listStud)[iStud].Number(),FALSE);
strcpy(buf,(*listStud)[iStud].Name().c_str());
SetDlgItemText(hdlg,IDC_ESTUDNAME,buf);
SetDlgItemInt(hdlg,IDC_EWORKCNT,(*listStud)[iStud].GetProgCnt(),FALSE);
SetDlgItemInt(hdlg,IDC_EPROGNUMBER,(*cpa)[iPrg].PrgNumber(),FALSE);
strcpy(buf,(*cpa)[iPrg].PrgName().c_str());
SetDlgItemText(hdlg,IDC_EPROGNAME,buf);
}
break;}
////////////////////////////////////////////////////Следующая работа//////////////////////////////
case IDC_NEXTWORK:
{
//Если можно - наращиваем индекс работы
if(iPrg<(cPrg-1))
{
iPrg++;
//Заполняем   поля работы
SetDlgItemInt(hdlg,IDC_EPROGNUMBER,(*cpa)[iPrg].PrgNumber(),FALSE);
strcpy(buf,(*cpa)[iPrg].PrgName().c_str());
SetDlgItemText(hdlg,IDC_EPROGNAME,buf);
}
break;}
}//switch wparam
}//switch msg
return FALSE;}
////////////////////////////////////////////////////Функция поиска//////////////////////////////
BOOL CALLBACK DlgSearchStud(HWND hdlg,UINT msg,WPARAM
wParam,LPARAM lParam)
{//Сюда мы попадаем при посылке пользователем любого сообщения
//из панели диалога
cStud=listStud->GetItemsInContainer();
////////////////////////////////////////////////В звисимости от кода сообщения//////////////////
switch (msg)
{//////////////////Инициализация окна диалога///////////////
case WM_INITDIALOG:
{st=NULL; iPrg=0;
MessageBox(NULL,"Введите группу и фамилию студента",
"Порядок ввода",MB_OK | MB_ICONINFORMATION);
//Меняем заголовок многократно используемого окна диалога
SetWindowText(hdlg,"Поиск данных о заданном студенте");
return TRUE;}
```

```
//////////////////////////////////////Обработка команд от окна диалога//////////////////
case WM_COMMAND:
switch(wParam)
{///////////////Прекращение ввода////////////////////////////////
case IDCANCEL:
{EndDialog(hdlg,0); return TRUE;}
///////////////Завершение ввода задания на поиск//////////////////
case  IDC_FINI:
{//Читам введенные данные  запроса (группу и фамилию)
GetDlgItemText(hdlg,IDC_EGROUP,Group,16);//Группу
GetDlgItemText(hdlg,IDC_ESTUDNAME,StudName,32);//Фамилию
//Если о студенте не все сказано
if(strlen(Group)==0 || strlen(StudName)==0)
{MessageBox(NULL,"Не введена группа или фамилия студента","Ошибка
ввода",MB_OK | MB_ICONEXCLAMATION);
return TRUE;}
string sg(Group),sn(StudName);
//Конструируем его
st=new Student(sg,StudNumber,sn);
//Ищем студента в массиве.
iStud=(*listStud).Find(*st);
if(iStud== INT_MAX) {MessageBox(NULL,"Заданный студент не найден","Ошибка
",
MB_OK | MB_ICONEXCLAMATION);return FALSE;}
//Если нашли - получим и выведем номер студента и количество программ
SetDlgItemInt(hdlg,IDC_ESTUDNUMBER,(*listStud)[iStud].Number(),FALSE);
cpa=(*listStud)[iStud].lstPrg();//Массив программ найденного студента
cPrg=cpa->GetItemsInContainer();//Их количество
SetDlgItemInt(hdlg,IDC_EWORKCNT,cPrg,FALSE);
//Выводим первую работу из массива работ
SetDlgItemInt(hdlg,IDC_EPROGNUMBER,(*cpa)[iPrg].PrgNumber(),FALSE);
strcpy(buf,(*cpa)[iPrg].PrgName().c_str());
SetDlgItemText(hdlg,IDC_EPROGNAME,buf);
break;}
//////////////////////////////////////////////////Следующая работа//////////////////////////////
case IDC_NEXTWORK:
{//Если можно - наращиваем индекс работы
if(iPrg<(cPrg-1)) iPrg++;
//Заполняем   поля работы
SetDlgItemInt(hdlg,IDC_EPROGNUMBER,(*cpa)[iPrg].PrgNumber(),FALSE);
strcpy(buf,(*cpa)[iPrg].PrgName().c_str());
SetDlgItemText(hdlg,IDC_EPROGNAME,buf);
break;}
//////////////////////////////////////////////////Искать следующего студента//////////////////////
case IDC_NEXTSTUD:
{iPrg=0;
```

```
//Очистим все поля
SetDlgItemText(hdlg,IDC_EGROUP,"");
SetDlgItemText(hdlg,IDC_ESTUDNUMBER,"");
SetDlgItemText(hdlg,IDC_ESTUDNAME,"");
SetDlgItemText(hdlg,IDC_EWORKCNT,"");
SetDlgItemText(hdlg,IDC_EPROGNUMBER,"");
SetDlgItemText(hdlg,IDC_EPROGNAME,"");
MessageBox(NULL,"Введите группу и фамилию студента",
"Порядок ввода",MB_OK | MB_ICONINFORMATION);
break;}
}//switch wparam
}//switch msg
return FALSE;}
//////////////////////////////Функция добавления программы заданному студенту//////////
BOOL CALLBACK DlgAddProg(HWND hdlg,UINT msg,WPARAM wParam,LPARAM
lParam)
{//Сюда мы попадаем при посылке пользователем любого сообщения
cStud=listStud->GetItemsInContainer();//Определим количество студентов
cpArray* cpa=(*listStud)[iStud].lstPrg();//Массив программ i-го студента
cPrg=cpa->GetItemsInContainer();        //Количество работ у студента
//////////////////////////////////В звисимости от кода сообщения/////////////////
switch (msg)
{////////////////////////////////////////////Инициализация окна диалога///////////////
case WM_INITDIALOG:
{st=NULL;  NewStud=TRUE;  iPrg=0;
MessageBox(NULL,"Введите группу,фамилию студента, номер и название
программы",
"Порядок ввода",MB_OK | MB_ICONINFORMATION);
//Меняем заголовок многократно используемого окна диалога
SetWindowText(hdlg,"Добавление программ заданному студенту");
return TRUE;}
/////////////////////////////////////////Обработка команд от окна диалога//////////////////
case WM_COMMAND:
switch(wParam)
{////////////////Прекращение ввода////////////////////////////////
case IDCANCEL:
{EndDialog(hdlg,0); return TRUE; }
//////////////////////////////Завершение ввода задания на поиск//////////////////
case  IDC_FINI:
{//Читаем введенные данные  запроса (группу и фамилию)
GetDlgItemText(hdlg,IDC_EGROUP,Group,16);      //Группу
GetDlgItemText(hdlg,IDC_ESTUDNAME,StudName,32);//Фамилию
//Если о студенте не все сказано
if(strlen(Group)==0 || strlen(StudName)==0)
{MessageBox(NULL,"Не введена группа или фамилия студента","Ошибка ввода",
MB_OK | MB_ICONEXCLAMATION);
```

```
return TRUE;}
string sg(Group),sn(StudName);
//Конструируем студента
st=new Student(sg,StudNumber,sn);
//Ищем студента в массиве.
iStud=(*listStud).Find(*st);
if(iStud== INT_MAX) {MessageBox(NULL,"Заданный студент не найден","Ошибка
",
MB_OK | MB_ICONEXCLAMATION);return FALSE;}
//Если нашли - получим и выведем его номер и количество программ
SetDlgItemInt(hdlg,IDC_EWORKCNT,(*listStud)[iStud].Number(),FALSE);
cpArray* cpa=(*listStud)[iStud].lstPrg();//Массив программ найденного студента
cPrg=cpa->GetItemsInContainer();        //Их количество
SetDlgItemInt(hdlg,IDC_EWORKCNT,cPrg,FALSE);
//Читаем сведения о работе
ProgNumber=GetDlgItemInt(hdlg,IDC_EPROGNUMBER,&err,FALSE);//Номер
программы
GetDlgItemText(hdlg,IDC_EPROGNAME,ProgName,128);//Название программы
//Если они есть-добавим в массив программ студента
if((ProgNumber!=0) && (strlen(ProgName)!=0))
{string sp(ProgName);
ComputerProgram cp=*new ComputerProgram(ProgNumber,sp);
(*listStud)[iStud].lstPrg()->Add(cp);
((*listStud)[iStud].ProgCnt)+=1;
//Корректируем поле количества программ
SetDlgItemInt(hdlg,IDC_EWORKCNT,(*listStud)[iStud].ProgCnt,FALSE);
}
break;}
//////////////////////////////////////////////////////Следующая работа////////////////////////////////
case IDC_NEXTWORK:
{MessageBox(NULL,"Введите номер и название программы",
"Порядок ввода",MB_OK | MB_ICONINFORMATION);
//Очистим поля работы - пользователь их заполнит и нажмет "Конец ввода"
SetDlgItemText(hdlg,IDC_EPROGNUMBER,"");
SetDlgItemText(hdlg,IDC_EPROGNAME,"");
break;}
//////////////////////////////////////////////////////Искать следующего студента//////////////////////
case IDC_NEXTSTUD:
{MessageBox(NULL,"Введите группу,фамилию студента, номер и название
программы",
"Порядок ввода",MB_OK | MB_ICONINFORMATION);
//Очистим все поля
SetDlgItemText(hdlg,IDC_EGROUP,"");
SetDlgItemText(hdlg,IDC_ESTUDNUMBER,"");
SetDlgItemText(hdlg,IDC_ESTUDNAME,"");
SetDlgItemText(hdlg,IDC_EWORKCNT,"");
```

```
SetDlgItemText(hdlg,IDC_EPROGNUMBER,"");
SetDlgItemText(hdlg,IDC_EPROGNAME,"");
break;}
}//switch wparam
}//switch msg
return FALSE;}

//////////Формирование структуры для работы с файлами через стандартный диалог///
void StructFile(void)
{//инициализация имени выбираемого файла не нужна - создаем пустую строку
szFile[0]='\0';
//записываем нулевые значения во все поля структуры для выбора файла
memset(&ofn, 0, sizeof(OPENFILENAME));
//инициализируем нужные нам поля
ofn.lStructSize= sizeof(OPENFILENAME);//размер структуры
ofn.hwndOwner=NULL;                  //идентификатор родительского окна
ofn.lpstrFilter=szFilter;            //адрес строки фильтра
ofn.nFilterIndex=1;                  //номер позиции выбора в начале
//адрес буфера для записи пути выбранного файла
ofn.lpstrFile=szFile;
//размер буфера для записи пути выбранного файла
ofn.nMaxFile=sizeof(szFile);
//адрес буфера для записи имени выбранного файла
ofn.lpstrFileTitle=szFileTitle;
//размер буфера для записи имени выбранного файла
ofn.nMaxFileTitle=sizeof(szFileTitle);
//в качестве начального каталога для поиска выбираем текущий каталог
ofn.lpstrInitialDir=NULL;
//определяем режимы выбора файла
ofn.Flags=OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST|
OFN_HIDEREADONLY;
}

//Файл warray.cpp содержит реализации объявленных в Array.hpp функций
//и объявления  необходимых данных
#include<conio.h>
#include<io.h>
#include<stdio.h>
#include<stdlib.h>
#include"warray.hpp"
//Конструктор  программы из потока
ComputerProgram::ComputerProgram(ifstream & ifs)
{ifs>>ProgNumber;   getline(ifs,ProgName);}
//Оператор присвоения для класса программ
ComputerProgram& ComputerProgram::operator=(const ComputerProgram& cp)
{
```

```
ProgNumber=cp.ProgNumber;//Копируем номер программы
ProgName=cp.ProgName;
return *this;
}
//Вывод  программ в поток
ofstream& operator<<(ofstream & ofs,ComputerProgram & cp)
{ofs<<cp.ProgNumber<<'\n'<<cp.ProgName<<'\n';return ofs;}
//Чтение программ  из потока
ifstream& operator>>(ifstream & ifs,ComputerProgram& cp)
{ifs.ignore('\n',10);ifs>>cp.ProgNumber;ifs>>cp.ProgName;}
return ifs;
}

//Конструктор копирования класса студент
Student::Student(const Student& st)
{
group=st.group;number=st.number;ProgCnt=st.ProgCnt;StudName=st.StudName;
*(listProg)=*(st.listProg);
}
//Конструктор  студента из потока
Student::Student(ifstream& ifs)
{
getline(ifs,group);//ifs.ignore(10,'\n');
ifs>>number;//ifs.ignore(10,'\n');
getline(ifs,StudName);//ifs.ignore(10,'\n');
ifs>>ProgCnt;
ifs.ignore(10,'\n');//ifs.ignore(10,'\n');
listProg=new cpArray(0,0,1); //Создаем пустой контейнер программ
for(short i=0;i<ProgCnt;i++) //Добавляем в него все программы из потока
{(*listProg).Add(*new ComputerProgram(ifs));}
}
//Оператор присвоения для класса студент
Student& Student::operator=(const Student& st)
{
group=st.group;              //Копируем группу
number=st.number;          //Номер
StudName=st.StudName; //ФИО
ProgCnt=st.ProgCnt;        //Количество программ в контейнере
*listProg=*st.listProg;       //сам контейнер
return *this;
}
//Чтение студента из потока
ifstream& operator>>(ifstream & ifs,Student& st)
{
ifs>>st.group>>st.number>>st.StudName>>st.ProgCnt; //Читаем группу, номер,ФИО
for(short i=0;i<st.ProgCnt;i++)
```

```
{ComputerProgram* cp = new ComputerProgram;
ifs>>(*cp);   st.listProg->Add(*cp);
}
return ifs;
}

//Вывод студента в поток
ofstream& operator<<(ofstream& ofs,Student& st)
{ofs<<st.group<<'\n'<<st.number<<'\n'<<st.StudName<<'\n'<<st.ProgCnt<<'\n';
for(short i=0;i<st.ProgCnt;i++)ofs<<(*st.listProg)[i];
ofs.flush();
return ofs;
}
```

//**Заголовочный файл warray.hpp** с объявлениями классов Array.hpp

```
#include <fstream.h>
#include <classlib\arrays.h>
#include <cstring.h>
#define STRICT      //Просим коппилятор осуществлять строгую проверку типов
#include <windows.h> //Включить файл интерфейса с библиотекой Windows API
#include <windowsx.h>//То же для связи с расширенной версией Windows API
#include <commctrl.h>
#include <classlib\arrays.h>
#include "afxres.h"
#pragma headerstop
#include "resource.h"
//Объявление класса ComputerProgram. Мы умышленно сделали его простым -
//вы можете дополнить его другими полями, а в таком виде он понадобится
//нам  в следующих примерах программ
class ComputerProgram{
//Приватные (по умолчанию) данные-члены класса
short ProgNumber; //Номер программы
string ProgName;  //Название программы (тема)
//Общедоступные функции - члены класса
public:
//Набор конструкторов
//Конструктор без параметроов (по умолчанию)
ComputerProgram(){ProgNumber=0;ProgName="";};
//Конструктор с заданными параметрами
ComputerProgram(short pnmb,string pname):ProgNumber(pnmb)
{ProgName=pname;}
//Конструктор копирования
ComputerProgram(ComputerProgram& cp)
{ProgName=cp.ProgName;ProgNumber=cp.ProgNumber;}
//Конструктор из потока
ComputerProgram(ifstream & ifs);
```

```
//Деструктор
~ComputerProgram(){};
//Функции чтения приватных данных
short PrgNumber(){return ProgNumber;}
string PrgName(){return ProgName;}
//Оператор присвоения
ComputerProgram& operator=(const ComputerProgram& cp);
//Чтение объекта из потока
friend ifstream& operator>>(ifstream & ifs,ComputerProgram & cp);
//Вывод в поток
friend ofstream& operator<<(ofstream & ofs,ComputerProgram & cp);
//Оператор сравнения на равенство - делаем по номерам
int operator==(const ComputerProgram& cp)
{return ProgNumber==cp.ProgNumber ? 1 : 0;}

//Оператор сравнения на меньше - делаем по номерам
int operator<(ComputerProgram& cp)
{return ProgNumber<cp.ProgNumber ? 1 : 0;}
};

//Прежде всего для удобства переименуем класс массива - контейнера
typedef TArrayAsVector<ComputerProgram> cpArray;
//Теперь объявим шаблон класса Student
class Student{
string group;              //Группа
short number;              //Номер студента
string StudName;           // Фамилия И.О.
cpArray* listProg;         //Контейнер со сданными программами
public:
short ProgCnt;             //Количество программ в массиве программ
//Конструктор по умолчанию
Student()
{group="";number=0;StudName="";ProgCnt=0;
listProg= new cpArray(0,0,1); //Пустой контейнер со сданными программами;
}
//Конструктор с заданными параметрами
Student(string gr,short n,string sn,short pcnt):number(n),ProgCnt(pcnt)
{group=gr;StudName=sn;listProg=new cpArray(1,0,1);}
//Конструктор копирования
Student(const Student& st);
//Конструктор из потока
Student(ifstream & ifs);
//Деструктор
~Student(){delete listProg;}
//Функции для чтения значений закрытых членов-данных
string Group(){return group;}
```

```
short Number() {return number;}
string Name(){return StudName;}
short GetProgCnt(){return ProgCnt;}
//Функция для наращивания счетчика прграмм в контейнере
void plusPrgCnt(){ProgCnt+=1;}
cpArray* lstPrg(){return listProg;}
//Оператор присвоения для класса студент
Student& operator=(const Student& st);
//Чтение студента из потока
friend ifstream& operator>>(ifstream & is,Student& st);
//Вывод студента в поток
friend ofstream& operator<<(ofstream & os,Student& st);
//Оператор сравнения на равенство - делаем по номерам
int operator==(const Student& st)
{return (group==st.group && StudName==st.StudName) ? 1:0;}
//Оператор сравнения на меньше - делаем по номерам
int operator<(Student& st){return StudName<st.StudName ;}
};

//Переименовываем для краткости последующих записей класс библиотечного
//контейнера с с попутным указанием типа его элементов
typedef TArrayAsVector<Student> stArray;
HWND hwndTb;       // идентификатор Toolbar
HWND hwndSb;       // идентификатор Statusbar
// Описание кнопок Toolbar
TBBUTTON tbButtons[] =
{
  { 0, CM_NEW,  TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0},
  { 1, CM_OPEN, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0},
  { 2, CM_SAVE, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0},
  { 3, CM_SAVEAS,   TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0},
  { 0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0L, 0},
  { 4, CM_EXIT,  TBSTATE_ENABLED, TBSTYLE_BUTTON, 0L, 0}
};

//Прототипы функций обработчиков сообщений
BOOL WndProc_OnCreate(HWND hWnd, LPCREATESTRUCT lpCreateStruct);
void WndProc_OnDestroy(HWND hWnd);
void WndProc_OnCommand(HWND hWnd, int id,HWND hwndCtl, UINT codeNotify);
LRESULT WndProc_OnNotify(HWND hWnd, int idFrom, NMHDR FAR* pnmhdr);
void WndProc_OnSize(HWND hwnd, UINT state, int cx, int cy);
void WndProc_OnMove(HWND hwnd,int cx, int cy);
//void WndProc_OnDrawItem(HWND hwnd, const DRAWITEMSTRUCT *
lpDrawItem);
void WndProc_OnMenuSelect(HWND hwnd, HMENU hmenu, int item,
HMENU hmenuPopup, UINT flags);
```

ПРОГРАММА № 8. Стандартный орган для просмотра древовидных структур в Windows.

ЗАДАНИЕ:

Составить программу, которая в диалогах принимает от пользователя ввод древовидной структуры - массив групп студентов, внутри групп - массивы фамилий студентов, внутри студентов - массивы названий выполненных работ. Предусмотрите также файловое сохранение дерева и конструирование дерева из файла.

В меню услуг программы предусмотрите просмотр дерева и реализуйте его через стандартный элемент Tree View

МЕТОДИЧЕСКАЯ ПОМОЩЬ.

Орган управления Tree View может быть создан либо вызовом функции CreateWindow с параметром WC_TREEVIEW, либо с помощью редактора ресурсов. Для вставки элементов в дерево лучше всего использовать макрокоманду TreeView_InsertItem, которая посылает органу TreeView сообщение TVM_INSERTITEM.

У макрокоманды 2 параметра - идентификатор Tree View и указатель на структуру TV_INSERTSTRUCT с 3-мя полями - идентификаторы родителя, элемента, после которого будет вставлен текущий элемент и структура TV_ITEM с информацией о вставляемом элементе - ее состав изучите через систему помощи, а пример заполнения полей приведен нами в функции InsTreeItem.

Приводимый пример расчитан на то, что древовидная структура создана в памяти другими средствами, а орган Tree View используется только для удобного просмотра - например, поиск нужного элемента значительно облегчается. В целом же это не лучший вариант - лучше бы объединить процедуры создания и просмотра дерева.

Для каждого уровня вложенности при инициализации диалога (через меню "Просмотр") мы вначале определяем количественные характеристики - количество групп, массив количеств студентов в каждой группе, 2-мерный массив количеств работ укаждого студента каждой группы, используя функцию контейнеров GetItemsInContainer.

Затем создаем массивы идентификаторов вставляемых элементов - массив идентификаторов групп (мы считаем их все корневыми), 2-мерный массив идентификаторов студентов и 3 - мерный массив идентификаторов работ (пока не заполненные).

Затем в циклах осушествляем вставки по уровням, ссылаясь на идентификаторы и тексты вставляемых элементов, попутно заполняя массивы идентификаторов значениями, возвращаемыми макрокомандой - для вложенных уровней.

Обычно отображаемые тексты ветвей дерева сопровождают миниатюрными битовыми

изображениями - мыне это не сделали, оставив на самостоятельную проработку.

Последовательность работ здесь следующая:

-создается список изображений вызовом функции ImageListCreate.

- список заполняется с помощью функции ImageList_Add.

-список подключается к дереву макрокомандой TreeView_SetImageList

Битовые изображения должны при этом быть в наличии (например в ресурсах
с загрузкой перед добвкой в список через LoadBitmap
*/

Файл treeMain.cpp.

```
#define STRICT       //Просим коппилятор осуществлять строгую проверку типов
#include <windows.h> //Включить файл интерфейса с библиотекой Windows API
#include <windowsx.h>//То же для связи с расширенной версией Windows API
#include <commctrl.h>
#include <cstring.h>
#include "afxres.h"
#include "treedata.hpp"
#include "resource.h"
GARRAY* gArray;       //Массив групп

OPENFILENAME ofn;  //структура для выбора файла через стандартный диалог
char szFile[256];          //буфер для записи пути к выбранному файлу
char szFileTitle[256];     //буфер для записи имени  выбранного файла
//фильтр расширений имени файлов
char szFilter[256]="DataFiles\0*.dat;\0Any fILES\0*.*\0\0";

HINSTANCE hInst;
char szAppName[]  = "TreeViewApplication";
char szAppTitle[] = "Пример программы, использующей орган Tree View ";
HWND hwndTree;

// Описание функций
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);
BOOL WINAPI DlgAddGroup(HWND hWnd, UINT msg, WPARAM wParam,
LPARAM lParam);
BOOL WINAPI DlgAddStud(HWND hWnd, UINT msg, WPARAM wParam, LPARAM
lParam);
BOOL WINAPI DlgAddWork(HWND hWnd, UINT msg, WPARAM wParam,
LPARAM lParam);
BOOL WndProc_OnCreate(HWND hWnd, LPCREATESTRUCT lpCreateStruct);
void WndProc_OnDestroy(HWND hWnd);
void WndProc_OnCommand(HWND hWnd, int id, HWND hwndCtl, UINT codeNotify);
LRESULT WndProc_OnNotify(HWND hWnd, int idFrom, NMHDR FAR* pnmhdr);
void WndProc_OnSize(HWND hwnd, UINT state, int cx, int cy);
void WndProc_OnDrawItem(HWND hwnd, const DRAWITEMSTRUCT * lpDrawItem);
void StructFile(void);
HTREEITEM InsTreeItem(HTREEITEM hParent, LPSTR szText, HTREEITEM hAfter,
  int iImage, int iSelectedImage,int child);

#define COL_WHITE     (RGB (255, 255, 255))
BOOL CALLBACK
```

```
DlgProcTree (HWND hwnd, UINT mMsg, WPARAM wParam, LPARAM lParam);
BOOL tv_BuildRootFolder (HWND hwndLV);
void tv_ExpandChild (HWND hwndLV, LPARAM lParam);
BOOL tv_FetchMacro (LPARAM lParam, HWND hwndCtrl);

#define IDC_TREEVIEW 1235
////////////////////// Функция WinMain//////////////////////////////
#pragma argsused
int APIENTRY
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
      LPSTR lpCmdLine, int nCmdShow)
{  gArray=new GARRAY(0,0,1);//Массив групп
  MSG msg;
  // Инициализируем библиотеку стандартных органов управления
  InitCommonControls();
  hInst = hInstance;
  // Преверяем, не было ли это приложение запущено ранее
  HWND hWnd = FindWindow(szAppName, NULL);
  if(hWnd)
  {
   if(IsIconic(hWnd))  ShowWindow(hWnd, SW_RESTORE);SetForegroundWin-
dow(hWnd);
    return FALSE;
  }
  WNDCLASS wc;    // структура для регистрации
  wc.lpszMenuName  = MAKEINTRESOURCE(IDM_MENU1);//Имя меню
  wc.style         = CS_HREDRAW | CS_VREDRAW;//Стиль класса окна
  wc.lpfnWndProc   = (WNDPROC) WndProc;      //Указатель на оконную функцию
  wc.cbClsExtra    = 0;                    //Дополнит данные для класса окна
  wc.cbWndExtra    = 0;                     //Дополнит данные для окна
  wc.hInstance     = hInstance;             //Идентификатор программы
  wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);//Иконка для программы
  wc.hCursor       = LoadCursor(NULL, IDC_ARROW);    //Курсор мышки
  wc.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);//Кисть для фона
  wc.lpszClassName = (LPSTR)szAppName;     //Имя класса окна
  RegisterClass(&wc);
  hWnd = CreateWindow(szAppName,         // имя класса окна
                szAppTitle,       // заголовок окна
                WS_OVERLAPPEDWINDOW, // стиль окна
                CW_USEDEFAULT,       //горизонтальная позиция окна
                CW_USEDEFAULT,       //вертикальная позиция окна
                CW_USEDEFAULT,       //ширина окна
                CW_USEDEFAULT,       //высота окна
                0,                 //идентификатор родительского окна
0,              // идентификатор меню или дочернего окна
hInst,          //идентификатор программы
```

```
NULL);        //указатель на дополнительные данные
// Если создать окно не удалось, завершаем приложение
if(!hWnd)
{MessageBox(NULL,"Не удалось создать окно - CreateWindow","Ошибка",
MB_OK | MB_ICONWARNING);
return FALSE;
}
  // Отображаем окно и запускаем цикл обработки сообщений
  ShowWindow(hWnd, nCmdShow);   UpdateWindow(hWnd);
  while(GetMessage(&msg, NULL, 0, 0))
  {  TranslateMessage(&msg);    DispatchMessage(&msg);  }
  return msg.wParam;
}
////////////////////// Функция WndProc/////////////////////////////////////////////////////
LRESULT WINAPI WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM
lParam)
{  switch(msg)
   {  HANDLE_MSG(hWnd, WM_CREATE,     WndProc_OnCreate);
     HANDLE_MSG(hWnd, WM_DESTROY,    WndProc_OnDestroy);
     HANDLE_MSG(hWnd, WM_COMMAND,    WndProc_OnCommand);
     HANDLE_MSG(hWnd, WM_SIZE,       WndProc_OnSize);
     default:return(DefWindowProc(hWnd, msg, wParam, lParam));
   }}

////////////////////////////////////Добавить группу//////////////////////////////////
#pragma argsused
BOOL WINAPI DlgAddGroup(HWND hdlg, UINT msg, WPARAM wParam, LPARAM
lParam)
{//Сюда мы попадаем при посылке пользователем любого сообщения
//из панели диалога
switch (msg)
{//////////////////Инициализация окна диалога///////////////
case WM_INITDIALOG:
{SetWindowText(hdlg,"Создание новой группы");return TRUE;}
////////////////////////////////Обработка команд от окна диалога//////////////////
case WM_COMMAND:
switch(wParam)
{//////////////////////////////////Прекращение ввода//////////////////////////////
case IDCANCEL:
{EndDialog(hdlg,0);return TRUE;}
////////////////////////////////Конец ввода в окне диалога ///////////////////////
case IDOK://Пользователь закончил ввод
{char gr[16];
//Получаем название группы
GetDlgItemText(hdlg,IDC_EGROUP,gr,16);
//Конструируем ее
```

```
Group* group=new Group(string(gr),NULL);
int n=gArray->Find(*group);
if(strlen(gr)==0 || n!=INT_MAX)//Если не введена или найдена совпадающая
{
SetDlgItemText(hdlg,IDC_EMSG,"Группа или не введена или уже есть в
контейнере");
SetDlgItemText(hdlg,IDC_EGROUP,"");
delete group;
return TRUE;
}
//Добавляем группу в контейнер
gArray->Add(*group);
SetDlgItemText(hdlg,IDC_EMSG,"Группа принята в контейнер");
SetDlgItemText(hdlg,IDC_EGROUP,"");
return TRUE;
}//case idok
}//switch wparam
}//switch msg
return FALSE;
}
////////////////////////////////Добавить студента/////////////////////////////////////
#pragma argsused
BOOL WINAPI DlgAddStud(HWND hdlg, UINT msg, WPARAM wParam, LPARAM
lParam)
{//Сюда мы попадаем при посылке пользователем любого сообщения
//из панели диалога
switch (msg)
{//////////////////////////////////Инициализация окна диалога///////////////
case WM_INITDIALOG:
{SetWindowText(hdlg,"Новый студент");return TRUE;}
//////////////////////////////////Обработка команд от окна диалога//////////////////
case WM_COMMAND:
switch(wParam)
{////////////////Прекращение ввода//////////////////////////////
case IDCANCEL:
{EndDialog(hdlg,0);return TRUE;}
////////////////////Конец ввода в окне диалога ///////////////////////
case IDOK://Пользователь закончил ввод
{char gr[16],st[32];
//Получаем  группу
GetDlgItemText(hdlg,IDC_EGROUP,gr,16);
//Конструируем группу и ищем
Group* group=new Group(string(gr),NULL);
int gn=gArray->Find(*group);
if(gn==INT_MAX)//Если не нашли
{
```

```
SetDlgItemText(hdlg,IDC_EMSG,"Такой группы нет");
delete group;
return TRUE;
}
GetDlgItemText(hdlg,IDC_ESTUDNAME,st,32);
//Конструируем студента и ищем
Student* stud=new Student(string(""),string(st));
int sn=(*gArray)[gn].GetStArray()->Find(*stud);
//Если нашли
if(sn!=INT_MAX)
{SetDlgItemText(hdlg,IDC_EMSG,"Такой у нас уже есть - давайте другого");
delete stud;delete group;
return TRUE;
}
(*gArray)[gn].GetStArray()->Add(*stud);
SetDlgItemText(hdlg,IDC_EMSG,"Студент принят в наш уютный контейнер");
delete group;
break;
}}}
return FALSE;
}
//////////////////////////////////Добавить работу//////////////////////////////////////
#pragma argsused
BOOL WINAPI DlgAddWork(HWND hdlg, UINT msg, WPARAM wParam, LPARAM
lParam)
{
switch (msg)
{////////////////////Инициализация окна диалога///////////////
case WM_INITDIALOG:
{SetWindowText(hdlg,"Создание новой группы"); return TRUE;}
//////////////////////////////////Обработка команд от окна диалога//////////////////
case WM_COMMAND:
switch(wParam)
{case IDCANCEL:
{EndDialog(hdlg,0);return TRUE;}
////////////////////Конец ввода в окне диалога /////
case IDOK://Пользователь закончил ввод
{char gr[16],st[32],w[128];
//Получаем  группу и студента и работу
GetDlgItemText(hdlg,IDC_EGROUP,gr,16);
GetDlgItemText(hdlg,IDC_ESTUDNAME,st,32);
GetDlgItemText(hdlg,IDC_EWORK,w,128);
//Конструируем группу и студента
Group* group=new Group(string(gr),NULL);
Student* stud=new Student(string(""),string(st));
int gn=gArray->Find(*group);//Ищем группу
```

```
if(gn==INT_MAX)//Если не найдена группа
{
SetDlgItemText(hdlg,IDC_EMSG,"Такой группы нет");
delete group;delete stud;
return TRUE;
}
//Ищем студента
int sn=(*gArray)[gn].GetStArray()->Find(*stud);
if(sn==INT_MAX)
{SetDlgItemText(hdlg,IDC_EMSG,"Такого студента нет");
delete group;delete stud;
return TRUE;
}
//Конструируем работу
CP* cp=new CP(string(""),string(w));
(*(*gArray)[gn].GetStArray())[sn].GetCPArray()->Add(*cp);
SetDlgItemText(hdlg,IDC_EMSG,"Работа принята в контейнер");
if(group!=NULL) delete group;
if(stud!=NULL)delete stud;
break;
}//case idok
}//switch wparam
}//switch msg
return FALSE;
}
////////////////// Функция WndProc_OnCreate//////////////////////
#pragma argsused
BOOL WndProc_OnCreate(HWND hWnd, LPCREATESTRUCT lpCreateStruct)
{return TRUE;}
////////////////////////////////// Функция InsTreeItem//////////////////////////////
HTREEITEM InsTreeItem(HTREEITEM hParent, LPSTR szText,
  HTREEITEM hAfter, int iImage, int iSelectedImage,int child)
{
  TV_INSERTSTRUCT tvins;
  HTREEITEM hItem;
  ZeroMemory(&tvins,sizeof(tvins));
  tvins.item.mask = TVIF_TEXT|TVIF_CHILDREN|TVIF_PARAM | TVIF_IMAGE
|TVIF_SELECTEDIMAGE;
  tvins.item.pszText = szText;
  tvins.item.cchTextMax = strlen(szText);
  tvins.item.iImage = iImage;
  tvins.item.iSelectedImage = iSelectedImage;
  tvins.item.cChildren=child;
  tvins.hInsertAfter = hAfter;
  tvins.hParent = hParent;
  hItem = TreeView_InsertItem(hwndTree, &tvins);
```

```
  return hItem;
}

////////////////////////// Функция WndProc_OnDestroy//////////////////////
#pragma argsused
void WndProc_OnDestroy(HWND hWnd)
{int i,j;
for(i=0;i<gcnt;i++)
for(j=0;j<scnt[i];j++) delete[] ProgItem[i][j];;
for(i=0;i<gcnt;i++)    delete[] ProgItem[i];
                       delete[] ProgItem;
for(i=0;i<gcnt;i++)    delete[] StudItem[i];
                       delete[] StudItem;
                       delete[] GroupItem;
for(i=0;i<gcnt;i++)    delete[]   pcnt[i];
                      delete[]   pcnt;
  DestroyWindow(hwndTree);  PostQuitMessage(0);}

////////////////////// Функция WndProc_OnCommand//////////////////////
#pragma argsused
void WndProc_OnCommand(HWND hWnd, int id, HWND hwndCtl, UINT codeNotify)
{ DestroyWindow(hwndTree);
  switch (id)
  {case CM_EXIT:{ PostQuitMessage(0); break;}
   case CM_ADDGROUP:
   {   DialogBox(hInst,MAKEINTRESOURCE(IDD_DIALOG1),hWnd,DlgAddGroup);
   break;
   }
   case CM_ADDSTUD:
   {   DialogBox(hInst,MAKEINTRESOURCE(IDD_DIALOG2),hWnd,DlgAddStud);
   break;
   }
   case CM_ADDWORK:
   {   DialogBox(hInst,MAKEINTRESOURCE(IDD_DIALOG3),hWnd,DlgAddWork);
   break;
   }
//////////////////////////////////////////////Просмотр дерева//////////////////////////
   case CM_VIEW:
   { DialogBox (hInst, MAKEINTRESOURCE (IDD_DIALOG), NULL, DlgProcTree) ;
  break;
   }
//////////////////////////////////Дерево из файла//////////////////////////////
   case CM_OPEN:
   {//Если контейнер еще не создан - создадим
if(gArray==NULL) gArray=new GARRAY(0,0,1);
StructFile();     //Заполним структуру для работы с файлом
```

```
if(GetOpenFileName(&ofn))//Если пользователь выбрал файл
{int cnt;
//открываем выбранный файл для чтения
ifstream fin(ofn.lpstrFile);
fin>>cnt;              //Читаем количество групп
for(int i=0;i<cnt;i++)
{Group* g=new Group(fin);
(*gArray).Add(*g);
}
fin.close();}
break;}
//////////////////////////////////Сохранить дерево в файле//////////////////////////////////////
case CM_SAVEAS:
{StructFile();     //Заполним структуру для работы с файлом
if(GetSaveFileName(&ofn))//Если пользователь выбрал файл
{//открываем выбранный файл для записи
ofstream fout(ofn.lpstrFile);
int cnt=(*gArray).GetItemsInContainer();
fout<<cnt<<'\n';
for(int i=0;i<cnt;i++) fout<<(*gArray)[i]<<'\n';
fout.close();
}
break;}
default:                                  break;
 }}
//////////////////// Функция WndProc_OnSize//////////////////////////
#pragma argsused
void WndProc_OnSize(HWND hwnd, UINT state, int cx, int cy) {}

///////Формирование структуры для работы с файлами через стандартный диалог///
void StructFile(void)
{//инициализация имени выбираемого файла не нужна - создаем пустую строку
szFile[0]='\0';
//записываем нулевые значения во все поля структуры для выбора файла
memset(&ofn, 0, sizeof(OPENFILENAME));
//инициализируем нужные нам поля
ofn.lStructSize= sizeof(OPENFILENAME);//размер структуры
ofn.hwndOwner=NULL;                  //идентификатор родительского окна
ofn.lpstrFilter=szFilter;            //адрес строки фильтра
ofn.nFilterIndex=1;                  //номер позиции выбора в начале
//адрес буфера для записи пути выбранного файла
ofn.lpstrFile=szFile;
//размер буфера для записи пути выбранного файла
ofn.nMaxFile=sizeof(szFile);
//адрес буфера для записи имени выбранного файла
ofn.lpstrFileTitle=szFileTitle;
```

```
//размер буфера для записи имени выбранного файла
ofn.nMaxFileTitle=sizeof(szFileTitle);
//в качестве начального каталога для поиска выбираем текущий каталог
ofn.lpstrInitialDir=NULL;
//определяем режимы выбора файла
ofn.Flags=OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST|
OFN_HIDEREADONLY;
}

////////////////////////////////////Функция диалога с органом Tree View ////////////////////////
#pragma argsused
BOOL CALLBACK
DlgProcTree(HWND hwnd, UINT mMsg, WPARAM wParam, LPARAM lParam)
    { switch (mMsg)
        { case WM_INITDIALOG :
            {//Определим количесвенные х-ки каждого уровня вложенности
 int i,j,k,
 gcnt=gArray->GetItemsInContainer();//количество групп - число
 scnt=new int[gcnt];          //Массив количеств студентов в группах
 for(i=0;i<gcnt;i++)          //Количество студентов в каждой группе
 scnt[i]=(*gArray)[i].GetStArray()->GetItemsInContainer();
 //Количества программ у каждого студента каждой группы - 2-мерный массив
 pcnt= new int*[gcnt];        //Указатель на массив
 for(i=0;i<gcnt;i++) pcnt[i]=new int[scnt[i]];
 for(i=0;i<gcnt;i++)
 for(j=0;j<scnt[i];j++)
 pcnt[i][j]=(*(*gArray)[i].GetStArray())[j].GetCPArray()->GetItemsInContainer();
 //Массив элементов 1-го уровня вложенности
 hRootItem=new HTREEITEM[gcnt];//
 //Разместим в куче 2- мерный массив 2-го уровня
 StudItem=new HTREEITEM*[gcnt];
 for(i=0;i<gcnt;i++) StudItem[i]=new HTREEITEM[scnt[i]];
 //Разместим в куче 3- мерный массив 3-го уровня
 ProgItem= new HTREEITEM**[gcnt];
 for(i=0;i<gcnt;i++) ProgItem[i]=new HTREEITEM*[scnt[i]];
 for(i=0;i<gcnt;i++)
 for(j=0;j<scnt[i];j++)
 ProgItem[i][j]=new HTREEITEM[pcnt[i][j]];
 hwndTree = GetDlgItem (hwnd, IDC_MESSAGES) ;//Получим идентификатор
//Вставляем корневые элементы (группы)
for(i=0;i<gcnt;i++)
hRootItem[i]=InsTreeItem((HTREEITEM)TVI_ROOT,
(LPSTR)(*gArray)[i].GetName().c_str(),
(HTREEITEM)TVI_FIRST, 0, 0,1);
// Вставляем вложенные элементы первого уровня (студенты)
string bs("  ");
```

```
for(i=0;i<gcnt;i++)
for(j=0;j<scnt[i];j++)
StudItem[i][j] = InsTreeItem(hRootItem[i],
(LPSTR)((*(*gArray)[i].GetStArray())[j].GetNumber() +bs+
(*(*gArray)[i].GetStArray())[j].GetName()).c_str(),
(HTREEITEM)TVI_LAST, 0, 0,1);
//Вставляем вложенные элементы 2-го уровня (работы)
for(i=0;i<gcnt;i++)
{for(j=0;j<scnt[i];j++)
{for(k=0;k<pcnt[i][j];k++)
{  InsTreeItem(  StudItem[i][j],
  (LPSTR)((*(*(*gArray)[i].GetStArray())[j].GetCPArray())[k].GetNumber()+bs+
  (*(*(*gArray)[i].GetStArray())[j].GetCPArray())[k].GetName()).c_str(),
  (HTREEITEM)TVI_LAST, 0, 0,0);
  }  }  }
return TRUE ;}
//////////////////////////////////////Команды/////////////////////////////////////////////////
case WM_COMMAND :
switch (LOWORD(wParam))
{case IDOK : EndDialog (hwnd, 0) ; break ;// Close dialog}
default :return FALSE ;
}}
```

Файл treedata.cpp.

```
#include<conio.h>
#include<io.h>
#include<stdio.h>
#include<stdlib.h>
#include"treedata.hpp"
//////////////////////////////////////////////ПРОГРАММА/////////////////////////////////////
//Конструктор  программы из потока
CP::CP(ifstream & ifs)
{getline(ifs,CPNumber);getline(ifs,CPName);}
//Оператор присвоения для класса программ
CP& CP::operator=(const CP& cp)
{CPNumber=cp.CPNumber;//Копируем номер программы
CPName=cp.CPName;
return *this;
}
//Вывод  программы в поток
ofstream& operator<<(ofstream & ofs,CP & cp)
{ofs<<cp.CPNumber<<'\n'<<cp.CPName<<'\n';
return ofs;
}
//Чтение программ  из потока
ifstream& operator>>(ifstream & ifs,CP& cp)
```

```
{getline(ifs,cp.CPNumber);getline(ifs,cp.CPName);return ifs;}

//////////////////////////////////////////////СТУДЕНТ//////////////////////////////////
//Конструктор копирования класса студент
Student::Student(const Student& st)
{SNumber=st.SNumber;    SName=st.SName;
if(cpArray!=NULL) cpArray->Flush(); else cpArray= new CPARRAY(0,0,1);
CPARRAY* scpa=st.GetCPArray();
int i, cnt;
if(scpa!=NULL)
{cnt=scpa->GetItemsInContainer();
for(i=0;i<cnt;i++) cpArray->Add((*scpa)[i]);
}}
//Конструктор  студента из потока
Student::Student(ifstream& ifs)
{int cnt;
getline(ifs,SNumber);  getline(ifs,SName);
cpArray=new CPARRAY(0,0,1); //Создаем пустой контейнер программ
ifs>>cnt;
for(short i=0;i<cnt;i++) //Добавляем в него все программы из потока
{(*cpArray).Add(*new CP(ifs));}
}
//Оператор присвоения для класса студент
Student& Student::operator=(const Student& st)
{SNumber=st.SNumber;     //Номер
SName=st.SName;         //ФИО
if(cpArray!=NULL) cpArray->Flush(); else cpArray= new CPARRAY(0,0,1);
CPARRAY* scpa=st.GetCPArray();
int i, cnt;
if(scpa!=NULL)
{cnt=scpa->GetItemsInContainer();
for(i=0;i<cnt;i++) cpArray->Add((*scpa)[i]);
}
return *this;}
//Чтение студента из потока
ifstream& operator>>(ifstream & ifs,Student& st)
{int cnt;
getline(ifs,st.SNumber);  getline(ifs,st.SName);
ifs>>cnt;
for(short i=0;i<cnt;i++)  {CP* cp = new CP;
ifs>>(*cp);  st.cpArray->Add(*cp);
}
return ifs;}

//Вывод студента в поток
ofstream& operator<<(ofstream& ofs,Student& st)
```

```
{char s[8];
ofs<<st.SNumber<<'\n'<<st.SName<<'\n';
CPARRAY*cpa=st.GetCPArray();
int cnt=cpa->GetItemsInContainer();
ofs<<cnt<<'\n';
for(short i=0;i<cnt;i++)
{itoa(i,s,10);(*cpa)[i].CPNumber=string(s);ofs<<(*cpa)[i];
}
ofs.flush();
return ofs;
}

//////////////////////////ГРУППА//////////////////////////////////////////
//////////////////////Конструктор копирования//////////////////////////
Group::Group(const Group& gr)
{GName=gr.GName;
if(stArray!=NULL) stArray->Flush(); else stArray= new STARRAY(0,0,1);
STARRAY* gspa=gr.GetStArray();
int i, cnt;
if(gspa!=NULL)
{cnt=gspa->GetItemsInContainer();
for(i=0;i<cnt;i++) stArray->Add((*gspa)[i]);
}}
//////////////////////Конструктор  из потока//////////////////////////
Group::Group(ifstream& ifs)
{int cnt;
getline(ifs,GName);
stArray=new STARRAY(0,0,1); //Создаем пустой контейнер программ
ifs>>cnt;
for(short i=0;i<cnt;i++) //Добавляем в него все программы из потока
{(*stArray).Add(*new Student(ifs));}
}

//////////////////////Оператор присвоения//////////////////////////
Group& Group::operator=(const Group& gr)
{GName=gr.GName;        //ФИО
if(stArray!=NULL) stArray->Flush(); else stArray= new STARRAY(0,0,1);
STARRAY* gspa=gr.GetStArray();
int i, cnt;
if(gspa!=NULL)
{cnt=gspa->GetItemsInContainer();
for(i=0;i<cnt;i++) stArray->Add((*gspa)[i]);
}
return *this;}
//////////////////Чтение группы из потока//////////////////////
ifstream& operator>>(ifstream & ifs,Group& gr)
```

```
{int cnt;
getline(ifs,gr.GName);  ifs>>cnt;
for(short i=0;i<cnt;i++) gr.stArray->Add(*new Student(ifs));
return ifs;}
//////////////////////Вывод группы  в поток//////////////////////////////
ofstream& operator<<(ofstream& ofs,Group& gr)
{char s[8];
ofs<<gr.GName<<'\n';
STARRAY*sta=gr.GetStArray();
int cnt=sta->GetItemsInContainer();
ofs<<cnt<<'\n';
for(short i=0;i<cnt;i++)
{itoa(i,s,10);  (*sta)[i].SNumber=string(s); ofs<<(*sta)[i];}
ofs.flush();
return ofs;}
```

//Заголовочный файл treedata.hpp с объявлениями классов Array.hpp

```
#define STRICT       //Просим коппилятор осуществлять строгую проверку типов
#include <fstream.h>
#include <classlib\arrays.h>
#include <cstring.h>
#include <windows.h> //Включить файл интерфейса с библиотекой Windows API
#include <windowsx.h>//То же для связи с расширенной версией Windows API
#include <commctrl.h>
#include <classlib\arrays.h>
#include "afxres.h"
//Объявление класса CP.
class CP{//Приватные (по умолчанию) данные-члены класса
string CPName;  //Название программы (тема)
//Общедоступные функции - члены класса
public:
string CPNumber; //Номер программы
//Набор конструкторов
//Конструктор без параметроов (по умолчанию)
CP(){CPNumber="";CPName="";};
//Конструктор с заданными параметрами
CP(string pnmb,string pname)
{CPNumber=pnmb;CPName=pname;}
//Конструктор копирования
CP(CP& cp)
{CPName=cp.CPName;CPNumber=cp.CPNumber;}
//Конструктор из потока
CP(ifstream & ifs);
//Деструктор
```

```
~CP(){};
//Функции чтения приватных данных
string GetNumber(){return CPNumber;}
string GetName(){return CPName;}
//Оператор присвоения
CP& operator=(const CP& cp);
//Чтение объекта из потока
friend ifstream& operator>>(ifstream & ifs,CP & cp);
//Вывод в поток
friend ofstream& operator<<(ofstream & ofs,CP & cp);
//Оператор сравнения на равенство - делаем по номерам
int operator==(const CP& cp)
{return CPNumber==cp.CPNumber ? 1 : 0;}
//Оператор сравнения на меньше - делаем по номерам
int operator<(CP& cp)
{return CPNumber<cp.CPNumber ? 1 : 0;}
};

//Прежде всего для удобства переименуем класс массива - контейнера
typedef TArrayAsVector<CP> CPARRAY;

//Теперь объявим шаблон класса Student
class Student{
public:
string  SNumber;         //Номер студента
string  SName;           // Фамилия И.О.
CPARRAY* cpArray;        //Адрес контейнера со сданными программами
//Конструктор по умолчанию
Student(){SNumber="";SName="";
cpArray= new CPARRAY(0,0,1); //Пустой контейнер со сданными программами;
}
//Конструктор с заданными параметрами
Student(string n,string sn):SNumber(n),SName(sn){}
//Конструктор с заданными параметрами
Student(string n,string sn,CPARRAY *cpa):SNumber(n),SName(sn)
{if(cpa==NULL)cpArray= new CPARRAY(0,0,1);
}
//Конструктор копирования
Student(const Student& st);
//Конструктор из потока
Student(ifstream & ifs);
//Деструктор
~Student(){delete cpArray;}
//Функции для чтения значений закрытых членов-данных
string GetNumber()const {return SNumber;}
string GetName()const   {return SName;}
```

```
CPARRAY* GetCPArray()const {return cpArray;}
//Оператор присвоения для класса студент
Student& operator=(const Student& st);
//Чтение студента из потока
friend ifstream& operator>>(ifstream & is,Student& st);
//Вывод студента в поток
friend ofstream& operator<<(ofstream & os,Student& st);
//Оператор сравнения на равенство - делаем по номерам
int operator==(const Student& st){return (SName==st.SName) ? 1:0;}
//Оператор сравнения на меньше - делаем по номерам
int operator<(Student& st){return SName<st.SName ;}
};

//Переименовываем для краткости последующих записей класс библиотечного
//контейнера с с попутным указанием типа его элементов
typedef TArrayAsVector<Student> STARRAY;


//////////////////////Класс "Группа"////////////////////////////////
class Group{
string  GName;           // Название группы
STARRAY* stArray;        //Адрес контейнера студентов
public:
//Конструктор по умолчанию
Group(){GName="";stArray= new STARRAY(0,0,1); //Пустой контейнер со сданными
программами;
}
//Конструктор с заданными параметрами
Group(string sn,STARRAY*sta):GName(sn)
{if(sta==NULL)stArray= new STARRAY(0,0,1);}
//Конструктор копирования
Group(const Group& st);
//Конструктор из потока
Group(ifstream & ifs);
//Деструктор
~Group(){delete stArray;}
//Функции для чтения значений закрытых членов-данных
const string GetName()const {return GName;}
STARRAY* GetStArray()const {return stArray;}
//Оператор присвоения
Group& operator=(const Group& );
//Чтение группы из потока
friend ifstream& operator>>(ifstream & is,Group& );
//Вывод группы в поток
friend ofstream& operator<<(ofstream & os,Group&);
//Оператор сравнения на равенство - делаем по номерам
```

```
int operator==(const Group& gr){return (GName==gr.GName) ? 1:0;}
//Оператор сравнения на меньше - делаем по номерам
int operator<(Group& gr)  {return (GName<gr.GName)? 1:0;}
};
typedef TArrayAsVector<Group> GARRAY;
  HTREEITEM* hRootItem, * GroupItem, **  StudItem, *** ProgItem;
  int gcnt,*scnt,**pcnt;
```

ПРОГРАММА № 9. Грфический интерпретатор заданных параметрически функций
ЗАДАНИЕ:
Составить программу, которая принимает от пользователя параметрические формулы функций и строит соответствующий им график.
Создайте меню динамически, без использования редактора ресурсов.
Предусмотрите блокирование взаимосвязанных пунктов меню.
Предусмотрите масштабирование графиков по желанию пользователя и при изменении размеров окна.
Дополните приведенную в виде образца программу возможностью поворота графика функции в заданном пользователем направлении.
Создайте помощь пользователю.
МЕТОДИЧЕСКАЯ ПОМОЩЬ.
Методика создания систем помощи изложена в предоставляемом исполняемом файле.
Остальная работа является по существу одним из вариантов пройденного в предыдущих работах - меняется только прикладная сущность задачи, но вы ее уже выполняли в консольном режиме.
Все объявления данных и тексты функций подробно комментированы.
Приведенный пример содержит не относящиеся к теме работы подпрограммы построения вращающихся многоугольников в точках, отмеченных щелчком мыши – вы можете опустить эти фрагменты в настоящей работе и вынести их в отдельную программу
*/

Файл intrpr.cpp

```
#define STRICT       //Строгая проверка типов
#include <windows.h>
#include <windowsx.h>
#include <string.h>
#include <math.h>
#include <commdlg.h>
#include <mem.h>
#pragma hdrstop
#include "Menu.hpp"          //Здесь коды команд меню
#include "Dialog.hpp"        //Здесь идентификаторы элементов панелей диадлога
#include "interpr.hpp"       //Здесь объявления данных для интерпретации

int ObjectOk,gb, EvalOk;
int VertexCount,             //Количество вершин многоугольника
```

```
   PrevVertexCount;
POINT*pt,pa1[10];        //Массивы точек кривой и вершин многоугольника
double psi=M_PI/8;       //Угол поворота многоугольника вокруг его центра
double Radius;           //Радиус окружности, описанной вокруг иногоугольн
int cc;                  //Счетчик вызовов ф-ции PolygonBuilder
char szBuf[256];
static CHOOSEFONT cf;//Структура с информацией для одноименной функции
                      //выбора шрифтов в диалоге - она заполняет
                      //структуру LOGFONT
 static LOGFONT lf;          //Структура с аттрибутами шрифтов
 static HFONT hfont, hOldFont; //Описатели шрифтов
 HDC hdc;
//Структура для смещений координатных осей в зависимости от вида кривой
struct OFF{long x,y;}off;
//Координаты центра многоугольника
struct {long x,y;}PolyCenter;
//Массив точек
struct {double x;double y;}pa[9];
//Организующая часть программы
//Прототипы функций
//Главного окна
LRESULT CALLBACK  WndProc(HWND, UINT, WPARAM, LPARAM);
//Диалога по объекту
LRESULT CALLBACK DlgProc(HWND , UINT , WPARAM , LPARAM );
//Графического иллюстратора результатов
void GraphBuilder();
//Построителя многоугольников
void PolygonBuilder();
//Функция подготовки данных и вызова ChooseFont
GetFont(HWND hwnd,LOGFONT *lf,CHOOSEFONT* cf);
// Имя класса окна
char const szClassName[]   = "WinIdentClass";
// Заголовок окна
char const szWindowTitle[] =
"Графический интерпретатор функций, заданных параметрически";
WNDCLASS wc;    // структура для регистрации класса окна

// Функция WinMain Получает управление при запуске приложения
#pragma argsused
HINSTANCE hInst; //Идентификатор программы
HWND hwnd;      // идентификатор главного окна приложения
ATOM aWndClass; // атом для кода возврата
//Составим функцию инициализации главного окна приложения
//Ее аргументы - те же, что у WinMain
BOOL InitWindow(HINSTANCE hInstance,HINSTANCE hPrevInstance,
                LPSTR lpszCmdLine,int nCmdShow)
```

```
{//Заполняем поля оконной структуры и регистрируем класс окна в ОС
wc.style = CS_HREDRAW | CS_VREDRAW;;   // Стиль окна
// Указатель на функцию окна для обработки сообщений
wc.lpfnWndProc = (WNDPROC) WndProc;
// Размер дополнительной области данных в описании класса окна
wc.cbClsExtra = 0;
// Размер дополнительной области данных для каждого окна этого класса
wc.cbWndExtra = 0;
// Идентификатор приложения, которое создало данный класс
wc.hInstance = hInstance;
// Идентификатор пиктограммы, используемой для окно данного класса
wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
// Идентификатор курсора, используемого  для окно данного класса
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
// Цвет фона окна
wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
// Идентификатор меню
wc.lpszMenuName = (LPSTR)NULL;
//Имя, которое присваивается создаваемому классу
wc.lpszClassName = (LPSTR)szClassName;
// Регистрация класса
aWndClass = RegisterClass(&wc);
// В случае неудачи
  if(!aWndClass)  return FALSE;
// После успешной инициализации приложения создаем главное окно приложения
    hwnd = CreateWindow(
    szClassName,         // имя класса окна
    szWindowTitle,       // заголовок окна
    WS_OVERLAPPEDWINDOW, // стиль окна
    CW_USEDEFAULT,       // задаем размеры и расположение
    CW_USEDEFAULT,       // окна, принятые по умолчанию
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    0,                   // идентификатор родительского окна
    0,                   // идентификатор меню
    hInstance,           // идентификатор приложения
    NULL);               // указатель на дополнительные параметры
// Если создать окно не удалось, завершаем приложение
if(!hwnd)  return FALSE;
// Рисуем окно. Для этого после функции ShowWindow, рисующей окно,
//вызываем функцию UpdateWindows, посылающую  WM_PAINT в функцию окна
ShowWindow(hwnd, nCmdShow); UpdateWindow(hwnd);
return TRUE;
}

//Главная функция
```

```
int WINAPI
WinMain(HINSTANCE hInstance,// идентификатор текущей копии приложения
HINSTANCE hPrevInstance,       // идентификатор предыдущей копии приложения
LPSTR     lpszCmdLine,         // указатель на командную строку
int       nCmdShow)            // способ отображения главного окна приложения
{
MSG  msg;                      // структура для работы с сообщениями
hInst=hInstance;
//Вызовем функцию инициализации главного окна
BOOL b=InitWindow(hInstance,hPrevInstance,lpszCmdLine,nCmdShow);
if(b)// При успешной инициализации запускаем цикл обработки сообщений
for(;;)//Бесконечный цикл до получения WM_QUIT
{if(PeekMessage(&msg, NULL, 0, 0,PM_REMOVE))//Если в очереди есть сообщения
{if(msg.message==WM_QUIT) break;//Если это конец работы
TranslateMessage(&msg);
DispatchMessage(&msg);
}
else
{//Если очередь пуста - здесь можем делать что-то несрочное -пока не делаем ничего
}}
// После выхода из цикла освободим память под динамическими массивами
delete xy[0];delete xy[1];delete t;
//Возвращаем значение WParam, переданное функции PostQuitMessage
return msg.wParam;
}
//В программе нам понадобится меню и мы составим фенкцию для его
программного
//формирования
//Программный способ построения меню
//Набор описателей  главного меню и его позиций
HMENU hMenuBar,hMenuData,hMenuFont,hMenuHelp;
void MenuBuilder(HWND hwnd)
{// Создаем пустое меню верхнего уровня
hMenuBar = CreateMenu();
// Подключаем меню к главному окну приложения
SetMenu(hwnd, hMenuBar);
//Создаем выпадающее меню ввода данных
hMenuData = CreatePopupMenu();
//Добавляем строки в выпадающее меню
AppendMenu(hMenuData,MF_ENABLED |
MF_STRING,CM_INPUTFORMULA,"&Ввод формулы");
AppendMenu(hMenuData,MF_GRAYED |
MF_STRING,CM_ARRAYEVALUTOR,"&Расчет массивов ...");
AppendMenu(hMenuData,MF_GRAYED |
MF_STRING,CM_GRAPHBUILDER,"&Построение графика");
```

```
AppendMenu(hMenuData,MF_ENABLED | MF_STRING,CM_EXIT,"&Завершение
работы");
//Добавляем строку и временное меню в меню верхнего уровня
AppendMenu(hMenuBar,MF_ENABLED | MF_POPUP,(UINT)hMenuData,"&Меню
услуг");
//Создаем меню выбора шрифта для текстовых строк
hMenuFont = CreateMenu();
AppendMenu(hMenuBar,MF_ENABLED | MF_STRING,CM_FONT,"
Выбор&Шрифта  ");
//Создаем выпадающее меню помощи
hMenuHelp = CreatePopupMenu();
//Добавляем строки в выпадающее меню
AppendMenu(hMenuHelp,MF_ENABLED | MF_STRING,CM_HELPHELP,
"Как создать систему помощи в вашей программе");
AppendMenu(hMenuHelp,MF_ENABLED | MF_STRING,CM_HELPINTERPR,
"Как работать с программой интерпретации");
//AppendMenu(hMenuBar,MF_ENABLED | MF_STRING,CM_HELP,"Помощь");
AppendMenu(hMenuBar,MF_ENABLED |
MF_POPUP,(UINT)hMenuHelp,"&Помощь");
// Перерисовываем меню
DrawMenuBar(hwnd);
                                        }

//Функция выполнения расчетов
void evalutor()
{int i; ObjectOk=gb=EvalOk=0;
//Приведение всех строк к одному регистру и очистка
for(i=0;i<5;i++)
{touppercase(formula[i]);clearformula(formula[i]);}
//Вычисление макс, мин значений параметра и шага
tmax=get_exp(formula[2],0);
tmin=get_exp(formula[3],0);
tstep=get_exp(formula[4],0);
//Вычисляем количество точек
sc=(int)ceil(((tmax-tmin)/tstep));
//Выделение памяти под массивы после определения их размеров
t=new double[sc];     //Для параметра
xy[0]=new double[sc]; //Для x - координаты
xy[1]=new double[sc]; //Для y - координаты
//Теперь заполняем массивы
 for(index=0;index<sc;index++)
 t[index]=tmin+index*tstep;
 for(i=0;i<2;i++)
 { for(index=0;index<sc;index++)     xy[i][index]=get_exp(formula[i],  t[index]);
   }
//Отмечаем успешное завершение вычислений
```

```
ObjectOk=1;
}
//Функция построения многоугольника
void PolygonBuilder()
{if(VertexCount>=3 && VertexCount<10)
{double _x,_y;  int i;
HPEN hpen2=CreatePen(PS_SOLID,5,RGB(0,255,0));//Выберем перо для рисования
HPEN hpen1=CreatePen(PS_SOLID,5,RGB(255,255,255));//Для стирания - цвет фона
//При первом вызове (например при смене количества вершин)
if(!cc)
{SelectObject(hdc,hpen1);            //Выбираем перо цвета фона
Polyline(hdc,pa1,PrevVertexCount+1);//Стираем предыдущий многоугольник
//Заполняем массив "центрального" расположения
for(i=0;i<VertexCount;i++)
{pa[i].x=Radius*cos(i*2*M_PI/VertexCount);
pa[i].y=Radius*sin(i*2*M_PI/VertexCount);
cc=1;
}}
//При последующих вызовах
else
for(i=0;i<VertexCount;i++)
{ SelectObject(hdc,hpen1);
  Polyline(hdc,pa1,VertexCount+1);//Стираем
 _x=pa[i].x;_y=pa[i].y;          //Сохраняем центральеые координаты
 pa[i].x=_x*cos(psi)-_y*sin(psi); //Поворачиваем
 pa[i].y=_x*sin(psi)+_y*cos(psi);
}
//Смещаем в заданную точку
for(i=0;i<VertexCount;i++)
{pa1[i].x=PolyCenter.x+pa[i].x;
pa1[i].y=PolyCenter.y+pa[i].y;
}
//Чтобы замкнуть многоугольник последняя вершина = первой
pa1[VertexCount].x=pa1[0].x;
pa1[VertexCount].y=pa1[0].y;
SelectObject(hdc,hpen2);
Polyline(hdc,pa1,VertexCount+1);//Нарисуем
DeleteObject(hpen1);DeleteObject(hpen2);
}}
// Функция GetFont - выбор шрифта в стандартном диалоге
BOOL GetFont(HWND hWnd, LOGFONT *lf, CHOOSEFONT *cf)
{  LPSTR szFontStyle[LF_FACESIZE];
  // Записываем нулевые значения во все поля  структуры, которая будет
использована //для  выбора шрифта
  memset(cf, 0, sizeof(CHOOSEFONT));
   cf->lStructSize = sizeof(CHOOSEFONT); // Размер структуры
```

```
   cf->hwndOwner = hWnd;                    // Идентификатор окна
   cf->lpLogFont = lf;                      // Указатель на структуру LOGFONT
  // Флаги, определяющие внешний вид диалоговой панели
  cf->Flags = CF_SCREENFONTS | CF_USESTYLE| CF_EFFECTS;
   cf->lCustData = 0L;                      // Дополнительные данные
   cf->rgbColors = RGB(0,0,0);              // Цвет текста
    cf->lpfnHook = (FARPROC)NULL;           // Адрес функции фильтра
    cf->lpTemplateName = (LPSTR)NULL;       // Адрес шаблона диалоговой панели
   cf->hInstance = hInst;                   // Идентификатор копии прилоджения
    cf->lpszStyle = (LPSTR)szFontStyle;     // Стиль шрифта
   cf->nFontType = SCREEN_FONTTYPE;         // Тип шрифта
  // Ограничения на минимальный и максимальный размер шрифта
  cf->nSizeMin = 0;   cf->nSizeMax = 0;
   return ChooseFont(cf);                   // Вызываем функцию выбора шрифта
}
//Функция WndProc НЕ ВЫЗЫВАЕТСЯ ни из одной функции приложения.
//Эту функцию вызывает Windows в процессе обработки сообщений.
//Для этого адрес функции WndProc указывается при регистрации класса окна.
//Функция выполняет обработку сообщений для главного окна приложения
int cxClient,cyClient;
LRESULT CALLBACK
WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{  PAINTSTRUCT ps;
//Выполняем обработку сообщений. Идентификатор сообщения msg
  switch (msg)
  {  // Это сообщение приходит, когда вы завершаете работу приложения
case WM_DESTROY:
{// Уничтожаем созданные ранее меню и таймер
DestroyMenu(hMenuData);DestroyMenu(hMenuFont);
DestroyMenu(hMenuHelp);DestroyMenu(hMenuBar);
KillTimer(hwnd,1);
//Инициируем завершение работы приложения
PostQuitMessage(0); return 0;
}
 case WM_SIZE://Определяем размеры клиентской области окна
 { cxClient=LOWORD(lParam);   cyClient=HIWORD(lParam);
   Radius=sqrt(cxClient*cxClient+cyClient*cyClient)/20;
   cc=0;
   InvalidateRect(hwnd,NULL,TRUE);
   if(ObjectOk && gb && EvalOk) GraphBuilder();
   return 0;
   }

   //При создании окна снабдим его меню
case WM_CREATE:
{MenuBuilder(hwnd);            //Нарисуем меню
```

```
SetTimer(hwnd,1,300,NULL);  //Создадим таймер
return 0;}
case WM_PAINT:
{hdc=BeginPaint(hwnd,&ps);
//Если есть что рисовать
if(ObjectOk && EvalOk && gb)
{GraphBuilder();  PolygonBuilder();}
EndPaint(hwnd,&ps);
return 0;
}
//Обработка символьных клавиш
case WM_CHAR:
{//Однобуквенная строка для символьного представления количества вершин
char s[2];s[0]=(char)wParam;s[1]=0;//Не забудем закрыть строку нулем
PrevVertexCount=VertexCount;
VertexCount=atoi(s);//Преобразуем строку в число
if(VertexCount<3 || VertexCount>9) VertexCount=6;
//Если количество вершин изменилось - объявляем вызов PolygonBuilder как 1-й
if(PrevVertexCount!=VertexCount)cc=0;
PolyCenter.x=cxClient/4;PolyCenter.y=cyClient/4;
InvalidateRect(hwnd,NULL,TRUE);
return 0;
}

//Обработка левой кнопки мыши - мы  сохраняем координаты курсора
case WM_LBUTTONDOWN://Нажатие левой кнопки
{PolyCenter.x=LOWORD(lParam); PolyCenter.y=HIWORD(lParam);
return 0;
}
//Отпускание правой кнопки - меняем знак угла поворота многоугольника
case WM_RBUTTONUP:
{psi=-psi;return 0;}
//Обработка "тиков" таймера
case WM_TIMER:
{//По таймеру рисуем многоугольник
HDC hdc1=GetDC(hwnd);
PolygonBuilder();
ReleaseDC(hwnd,hdc1);
return 0;
}
//Обработка команд меню
case WM_COMMAND:
{
switch(wParam)
{
  //Команда завершения работы
```

```
case CM_EXIT:
{DestroyMenu(hMenuData);DestroyMenu(hMenuBar);DestroyWindow(hwnd);
     KillTimer(hwnd,1);
     PostQuitMessage(0);
return 0;}
   //Команда ввода данных об объекте
   case CM_INPUTFORMULA:
{    InvalidateRect(hwnd,NULL,TRUE); //Очищаем окно
    //Признак невведенных данных - после успешного ввода присвоим 1
  FormulaOk=EvalOk=gb=0;
    //Вызов функции формирования окна диалога
    DialogBox(hInst,MAKEINTRESOURCE(DIALOG),hwnd,(DLGPROC)DlgProc);
    return 0;
}
 //Команда выполнения расчетов
case CM_ARRAYEVALUTOR:
{gb=0;EvalOk=0;
hfont=CreateFontIndirect(&lf),hOldFont;
HDC hdc2= GetDC(hwnd);
hOldFont=SelectFont(hdc2,hfont); //Выбираем шрифт в контекст
SetTextColor(hdc2,cf.rgbColors);
//Отлавливание исключений, генерируемых вычислительной частью программы
try{evalutor();}
catch (InputErrors ir)
{int le=lstrlen(err[ir]);
SetTextColor(hdc,cf.rgbColors);
MessageBeep(MB_ICONQUESTION);
TextOut(hdc2,0.2*cxClient,0.8*cyClient,err[ir],le);
return 0;
}
//Если не было исключения  - работа продолжается
char*str="Расчет массива завершен!";
int ls=lstrlen(str);
//Можно выбрать шрифт из семейства стандартных
//lf.lfPitchAndFamily=FF_SCRIPT;//FF_DONTCARE;//FF_SWISS;//FF_MODERN;//FF
_ROMAN;
//Можно определить название шрифта
//GetTextFace(hdc2,80,szBuf);
//Добавим к нему выводимую строку
//lstrcat(szBuf,str);
TextOut(hdc2,(cxClient-ls)/4,cyClient-4*ls,str,ls); //Вывод строки
SelectFont(hdc2,hOldFont);                          //Выбираем старый шрифт
DeleteFont(hfont);                                  //Удаляем созданный шрифт
ReleaseDC(hwnd,hdc2);
EnableMenuItem(hMenuData,GetMenuItemID(hMenuData,2),MF_ENABLED);
EvalOk=1;
```

```
return 0;
}
//Команда построения графиков
case CM_GRAPHBUILDER:
{
gb=1;//Высказываем желание строить график
InvalidateRect(hwnd,NULL,TRUE);
return 0;
}
case CM_FONT:
{memset(&lf, 0, sizeof(LOGFONT)); // Выбираем шрифт для вывода текста
GetFont(hwnd,&lf,&cf);
lf.lfEscapement= lf.lfOrientation=450;//Под углом 45 градусов к горизонту
return 0;
}
//Обработка команд меню помощи
case CM_HELPHELP:
{WinHelp(hwnd,"help.hlp",HELP_CONTENTS,0);return 0;}
case CM_HELPINTERPR:
{//Здесь вы укажете созданный вами файл помощи
//WinHelp(hwnd,"interpr.hlp",HELP_CONTENTS,0);
return 0;
}
default: return 0;
   }   }}
// Все сообщения, которые не обрабатываются нашей  функцией окна, ДОЛЖНЫ
//передаваться функции DefWindowProc
 return DefWindowProc(hwnd, msg, wParam, lParam);
}
//Обрабртчик сообщений для диалога о формуле
#pragma argsused
LRESULT CALLBACK DlgProc(HWND hdlg, UINT uMsg, WPARAM wParam,
LPARAM lParam)
{switch (uMsg)
{case WM_INITDIALOG:
{return TRUE;}
case WM_COMMAND:
switch(wParam)
{case IDOK://Пользователь закончил ввод формулы
{//Получаем введенные строки
GetDlgItemText(hdlg,IDC_EDIT1,formula[0],70);
GetDlgItemText(hdlg,IDC_EDIT2,formula[1],70);
GetDlgItemText(hdlg,IDC_EDIT3,formula[2],70);
GetDlgItemText(hdlg,IDC_EDIT4,formula[3],70);
GetDlgItemText(hdlg,IDC_EDIT5,formula[4],70);
//Закомметированная ниже функция почему-то работает "через раз" -
```

```
//поэтому использована вышестоящая
//SendDlgItemMessage(hdlg,IDC_EDIT2,EM_GETLINE,(WPARAM)0,(LPARAM)(LPS
TR//)buf);
//После ввода данных разблокируем меню расчетов
EnableMenuItem(hMenuData,GetMenuItemID(hMenuData,1),MF_ENABLED);
EndDialog(hdlg,0);
return TRUE;
}
case IDCANCEL://
{EndDialog(hdlg,0);return TRUE;}
}}
 return FALSE;
}
//В конце - обеспечение графики
double ymax,ymin,xmax,xmin,mstbx,mstby;
//Вспомогательная функция определения масштаба и смещения осей
void getCoord(double** array)
{long i;
//Определим наибольшее, наименьшее значение , диапазон изменения x,y
xmax=xmin=array[0][0];ymax=ymin=array[1][0];//Для начала пусть так
 for(i=0;i<sc;i++)
 { if(array[0][i]>xmax)xmax=array[0][i];
 if(array[0][i]<xmin)xmin=array[0][i];
 if(array[1][i]>ymax)ymax=array[1][i];
 if(array[1][i]<ymin)ymin=array[1][i];
 }
//Смещение оси x по вертикали и масштаб по y
if(ymax!=ymin)
 {
 if(ymax>0 && ymin >=0) {mstby=cyClient/(ymax-ymin);off.x=cyClient;}
 if(ymax>0 && ymin <=0) {mstby=cyClient/(ymax-ymin);off.x=ymax*mstby;}
 if(ymax<0)             {mstby=cyClient/fabs(ymin);off.x=0;}
 }

 //Смещение оси y по горизонтали и масштаб по x
if(xmax!=xmin)
 { //Если хмакс и хмин положительны
 if(xmax>0 && xmin >=0) {mstbx=cxClient/(xmax-xmin);off.y=0;}
 if(xmax>0 && xmin <=0) {mstbx=cxClient/(xmax-xmin);off.y=fabs(xmin)*mstbx;}
 if(xmax<0 )            {mstbx=cxClient/(xmax-xmin);off.y=cxClient;}
 } }
//Собственно функция рисования
void GraphBuilder()
{int i;
//HDC hdc;
pt=new POINT[sc+1];
```

```
//Некоторый запас карандашей
HPEN hpen1=CreatePen(PS_SOLID,5,RGB(255,0,0)),
    hpen2=CreatePen(PS_SOLID,5,RGB(25,130,100));
    hdc=GetDC(hwnd);
getCoord(xy);
//Закрасим окно цветом фона - т е очистим
RECT rect;rect.top=0;rect.left=0;rect.right=cxClient;rect.bottom=cyClient;
FillRect(hdc,&rect,wc.hbrBackground);
SelectObject(hdc,hpen2); //Выберем перо
//Указатель - в  низ экрана в гориз позицию оси ординат
MoveToEx(hdc,off.y,cyClient,NULL);
LineTo(hdc,off.y,0);        //Вертикальная ось
//Стрелка на вертикальной оси
LineTo(hdc,off.y-3,4);LineTo(hdc,off.y+3,4); LineTo(hdc,off.y,0);
//Указатель в 0 по оси абсцисс и в поз гориз оси по вертикали
MoveToEx(hdc,0,off.x,NULL);
LineTo(hdc,cxClient,off.x); //Горизонтальная ось
//Стрелка
LineTo(hdc,cxClient-4,off.x-3);LineTo(hdc,cxClient-4,off.x+3);
LineTo(hdc,cxClient,off.x);
//Теперь заполним массив структур pt
 if(ymax!=ymin)
 {for(i=0;i<sc;i++) pt[i].x=off.y+(int)(xy[0][i]*mstbx);

for(i=0;i<sc;i++) pt[i].y=off.x+(long)(xy[1][i]*mstby);
pt[sc].x=pt[0].x;pt[sc].y=pt[0].y;
//Рисование графика можно выполнить с помощью различных функций
SelectObject(hdc,hpen1);
//MoveToEx(hdc,pt[0].x,pt[0].y,NULL);
//for(i=1;i<sc+1;i++)LineTo(hdc,pt[i].x,pt[i].y);
Polyline(hdc,pt,sc+1);
//PolyBezier(hdc,pt,3*(((sc-4)/3)+1)+1);
ReleaseDC(hwnd,hdc);
DeleteObject(hpen1);
delete pt;
}}
```

Файл sintax.cpp

```
//В этои файле мы определим все необходимые утилиты для синтаксического
//анализа и интерпретации (вычисления) выражений (формул)
#include <windows.h>
#include <stdio.h>
#include <math.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
```

```
#pragma hdrstop
#include "interpr.hpp"

extern HWND hwnd;
//НЕКОТОРЫЕ СЛУЖЕБНЫЕ ПОДПРОГРАММЫ, ПРЕДВАРЯЮЩИЕ
//СИНТАКСИЧЕСКИЙ АНАЛИЗ
//Функция приведения латинских букв к верхнему регистру
void touppercase(char *f)
{char *temp;
for(temp=f;*temp!='\0';temp++)
if(islower(*temp)) *temp=(char)toupper(*temp);
}
//Функция очистки формульной строки от излишков и неточностей
void clearformula(char *f)
{char *temp;  temp=f;
for(;*temp!='\0';)
if(isalnum(*temp) || strchr(OP,*temp)
|| *temp=='.' || *temp=='(' || *temp==')') temp++;
//Вначале удалим все недопустимые в формуле символы
else memmove(temp,temp+1,strlen(temp+1)+1);
//Затем несколько раз на наличие сочетаний типа "+*" или "-*"
temp=f+1;
for(;*temp!='\0';temp++)
if((*temp=='*' || *temp=='/' || *temp=='^') &&
(*(temp-1)=='+' || *(temp-1)=='-'))
{memmove(temp,temp+1,strlen(temp+1)+1);temp=f;}
}

/*Функция поиска по таблицам для определения типа
строковых лексем параметры функции - искомая строка и ад-
рес, по которому положить значение обнаруженной именован-
ной константы */
void look_up(char * s,double* cv)
{int i;
//Просмотрим таблицу операций
for(i=0;strlen(op[i].on);i++)
//Если нашли совпадение с допустимым именем операции
//возвращаем конкретный тип операции
if(!strcmp(op[i].on,s))
{token_type=OPERATION;tok= op[i].ov;return;}
//Поиск в таблице констант
for(i=0;strlen(tc[i].cn);i++)
/*Если нашли совпадение с допустимым именем константы
возвращаем индекс константы в массиве структур и ее значе-
ние записываем по заданному адресу */
if(!strcmp(tc[i].cn,s))
```

```
{token_type=OPERAND;tok=CONSTANT;*cv=tc[i].cv;return;}
//Если ничего не нашли
token_type=0;tok=-1;
}
//Функция выделения и классификации очередной лексемы возвращает класс
лексемы. //Ее параметры - указатель на указатель текущего текста формулы и адрес
для значения //операнда-константы - он транзитом передается в look_up(). Первый
параметр мы //вынуждены получать "по ссылке", чтобы значение адреса текущей
лексемы //изменялось в вызывающей программе, а не в нашей подпрограмме
get_token(char **cf,double*cv)throw(InputErrors)
{char *temp; //Временный указатель на лексему
token_type=0;tok=0;
char*opr;
for(int i=0;i<1;i++)
{//Если конец формулы
if(**cf=='\0') {*token=0; token_type=EOL;break;}
//Если это открытая круглая скобка
if(**cf=='(')
{temp=token;
*temp=**cf;                                  //перепишем лексему в token
(*cf)+=1;                                     // переход на следующую позицию
temp++; *temp=0; //token закроем нулем
tok=EXPRESS;
token_type=OPERAND;break;}

/*Если это закрытая круглая скобка
if(**cf==')')
{//(*cf)+=1;
temp=token;
*temp=**cf;                  //перепишем его в token
(*cf)+=1;              // переход на следующую позицию
temp++; *temp=0; //token закроем нулем
tok=EXPRESS;
return(token_type=SKOBKA);}
*/
//Если это символ арифметической операции
if((opr=strchr(OP,**cf))!=NULL)
 { temp=token;
*temp=**cf;         //перепишем его в token
(*cf)+=1;              // переход на следующую позицию
temp++; *temp=0; //token закроем нулем
switch(*opr)
{case '+': tok=PLUS;break;
case '-': tok=MINUS;break;
case '*': tok=MUL;break;
case '/': tok=DIV;break;
```

```
case '^': tok=POW;break;
}
//сообщим что класс лексемы - операция
token_type=OPERATION;break;
}
//Если встретили цифру
if(isdigit(**cf)) {
//то запишем всю числовую подстроку в token
temp=token;
while(isdigit(**cf) || **cf=='.')
{*(temp++)=**cf;(*cf)+=1;}
*temp = '\0';
tok=NUMBER;
token_type = OPERAND;break;
}
//Если встретили букву
if(isalpha(**cf)) {
/*то это переменная или операция - функция или именованная константа; пока все буквы - цифры перепишем и временно зарегистрируем просто строкой */
temp=token;
while(isalnum(**cf)) {*temp++=**cf;(*cf)+=1;}
token_type=STRING;}
*temp = '\0';

//Не отходя далеко проанализируем полученную строку
//Если это 1-буквенное имя независимой переменной
if(token_type==STRING && !strcmp(token,"T"))
{tok=VARIABLE;token_type=OPERAND;break;}
//В противном случае поищем среди унарных операций и констант
look_up(token,cv);
//Если не нашли - выбрасываем исключение
if(tok<0) throw(InvalidConstName);
}
return token_type;
}

//ТЕПЕРЬ УТИЛИТЫ ДЛЯ ИНТЕРПРЕТАЦИИ ВЫРАЖЕНИЙ
/* Нам понадобятся некоторые вспомогательные функции, в частности функция выполнения заданных двухместных математических операций - ее параметры при вызове - это код требуемой двухместной операции и значения левого и правого операндов.Все приводимые ниже функции целесообразно поместить в заголовочный файл с названием например interpr.h для последующего использования */
double oper(int co,double lo, double ro)throw(InputErrors)
{double res=0.0;
switch(co)
{case PLUS :res=lo+ro;break;
```

```
case MINUS:res=lo-ro;break;
case MUL :res=lo*ro;break;
case DIV :
{if(ro==0) throw(DivisionByZero);
else res=lo/ro;
break;}
case POW :res=pow(lo,ro);break;
default: throw(SintaxError);
}
return res;
}
/*Эта функция осуществляет унарные операции, к классу которых мы отнесли
помимо унарных "плюс" и "минус" все математические функции, полагая
находящееся в скобках после имени функции выражение единственным их
операндом.
Для сокращения дефицитного места в книге мы привели реализации
всего нескольких математических функций, а вы сможете рас-
ширить эти возможности по аналогии */
double unary(int co,double ro)throw(InputErrors)
{
double res;
switch(co)
{case PLUS :res=ro;break;
case MINUS:res=-ro;break;
case SIN :res=sin(ro);break;
case COS :res= cos(ro);break;
case TAN :res= tan(ro);break;
case SQRT :res= sqrt(ro);break;
default: throw(SintaxError);
}
return res;
}


/*Это уже основная подпрограмма вычисления значений по тексту формулы, она
возвращает вычисленное значение, а ее параметры - указатель на текст формулы
и значение независимой переменной */

/* Формула состоит из операндов и операций. Типы операндов - число, именованная
константа, переменная, выражение в скобках. */
struct FRM{                  //Для структуры формулы
unsigned dtok;               //Код типа операнда
unsigned co;                 //Код операции
double z;                    //значение операнда
};
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
double get_exp(char* f,double t)throw(InputErrors)
{FRM frm[32];
double result; //для промежуточных результатов вычислений
char privat_form[80]; //для текущего фрагмента формулы
int brack,                    //Счетчик скобок
i,j;
//рабочие переменные для циклов
double cv;                    //для значения именованной константы

for(i=0;i<32;i++)
memset(&frm[i],0,sizeof(frm[i]));//Обнулим
int cnt=0;                          //Количество элементов в формуле
char* tmp=f;                        //Исходный адрес формулы
/*Просмотрим текст формулы и заполним ее структуру*/
for(i=0,get_token(&tmp,&cv);token_type!=EOL;i++,get_token(&tmp,&cv))
{//Если получили операнд - число, константу или переменную
switch(token_type)
{case OPERAND:
{frm[i].dtok=tok;
switch(tok)
{case NUMBER: frm[i].z=atof(token);break;
case CONSTANT: frm[i].z=cv;break;
case VARIABLE: frm[i].z=t;break;
case EXPRESS: //Если операнд - выражение в скобках
{int tt=token_type;//Сохраним тип лексемы
brack=1;
/*Перепишем все после нее до парной ей закрытой в массив privat_form, создавая
таким образом другую, частную формулу, как фрагмент общей - для нее мы можем
использовать такой же механизм исследования */
 for(j=0;brack && j<78;j++,tmp++)
 {if(*tmp=='(')brack++; if(*tmp==')')brack--;
 privat_form[j]=*tmp; }
 privat_form[j-1]=0;//Закроем нулем
 if(brack) throw(UnpairedParentheses); //Если не нашли парную скобку
 result=get_exp(privat_form,t);
 token_type=tt;//Восстановим тип лексемы
 frm[i].dtok=NUMBER;
 frm[i].z=result;
 break;
 }//case EXPRESS
}//switch tok
break;
}//case OPERAND
//Если получили операцию
case OPERATION: frm[i].co=tok;break;
}//switch
```

```
} //for
cnt=i;                  //Количество заполненных элементов
//Теперь можем обрабатывать. Вначале необходимо выполнить все унарные
операции
//- у них самый высокий приоритет. Признаком унарной операции является либо
//если она первая в структуре формулы, либо предыдущий в формуле элемент -
//операция, а последующий - операнд
int flag=1;
for(;flag;){
flag=0;
for(i=0;i<cnt;i++)
if((!i && frm[i].co && frm[i+1].dtok) ||
(i && frm[i-1].co && frm[i].co && frm[i+1].dtok))
{frm[i+1].z=unary(frm[i].co,frm[i+1].z);
//Сомкнем ряды на выполненной операции
memmove(&frm[i],&frm[i+1],(cnt-i-1)*sizeof(frm[i]));cnt--;flag++;
}}

/*Теперь необходимо выполнить все возведения в степень */
for(i=1;i<cnt;i++)
if(frm[i].co==POW && frm[i-1].dtok && frm[i+1].dtok)
{frm[i-1].z=oper(POW,frm[i-1].z,frm[i+1].z);
//Сомкнем ряды на выполненной операции
memmove(&frm[i],&frm[i+2],(cnt-i-2)*sizeof(frm[i]));i--;cnt-=2;
}

/*Теперь выполним все деления и умножения */
for(i=1;i<cnt;i++)
if((frm[i].co==MUL || frm[i].co==DIV) && frm[i-1].dtok &&
frm[i+1].dtok)
{frm[i-1].z=oper(frm[i].co,frm[i-1].z,frm[i+1].z);
//Сомкнем ряды на выполненной операции
memmove(&frm[i],&frm[i+2],(cnt-i-2)*sizeof(frm[i]));i--;cnt-=2;
}
/*Теперь выполним все сложения и вычитания */
for(i=1;i<cnt;i++)
if((frm[i].co==PLUS || frm[i].co==MINUS) &&
frm[i-1].dtok && frm[i+1].dtok)
{frm[i-1].z=oper(frm[i].co,frm[i-1].z,frm[i+1].z);
//Сомкнем ряды на выполненной операции
memmove(&frm[i],&frm[i+2],(cnt-i-2)*sizeof(frm[i]));i--;cnt-=2;
}
//Теперь в 0-м элементе лежит результат
return frm[0].z;
}
```

Файл interpr.hpp

```
#ifndef __INTERPR_H
#define __INTERPR_H
//ВНАЧАЛЕ ОПРЕДЕЛИМ ВСЕ ДАННЫЕ И УТИЛИТЫ, НЕОБХОДИМЫЕ ДЛЯ СИНТАКСИЧЕСКОГО АНАЛИЗА
int token_type,     //Тип лексемы - см ниже enum LexemClass
tok;//Подкласс лексемы - см ниже enum OperandCodes, enum OperationCodes
char token[10]; //для хранения выделенной лексемы
int FormulaOk;        //Флаг завершения ввода формулы
char formula[5][80]; //Для текста формул
double tmax,tmin,tstep;//Для максим, миним значений параметра и шага
int index;
int sc;              //Для количества точек расчета и построения графиков
double* t;           //Указатель на массив значений параметра
//Массив из 2-х указателей на массивы значений x,y в натуральных единицах
double* xy[2];
//Коды классов лексем
  enum LexemClass{OPERAND=1,OPERATION,SKOBKA,EOL};
//Подклассы (коды) операндов
 enum OperandCodes{STRING=1,  //Неклассифицированная пока строка
                              NUMBER,    //Число
                              VARIABLE,  //Переменная
                               CONSTANT,  // Именованная константа
                                EXPRESS};  // Выражение в скобках
//Коды операций
 enum OperationCodes
   {MUL=1,DIV,POW,PLUS,MINUS,ABS, ACOS,ASIN, ATAN, COS,
   COSH, EXP,LOG,LOG10,SIN,SINH,SQRT,TAN,TANH};
//Массив структур для именованных констант
struct cnst {
 char cn[6];//Имя константы
double cv; //Значение константы
}tc[2]={{"PI",M_PI},{"E",M_E}};
//Все обозначения (имена) операций и их коды сведем в таблицу
//(массив структур tf[]) по шаблону
struct {
char on[10];//Имя операции
int ov;     //Числовой код операции
}op[]=
{
{"ABS",ABS},{"ACOS",ACOS},{"ASIN",ASIN},{"ATAN",ATAN},
{"COS",COS},{"COSH",COSH},{"EXP",EXP},{"LN",LOG},
{"LOG",LOG10},{"SIN",SIN},{"SINH",SINH},{"SQRT",SQRT},
{"TAN",TAN},{"TANH",TANH},{"+",PLUS},{"-",MINUS},
{"*",MUL},{"/",DIV},{"^",POW},{"",0}};
char *OP="+-*/^"; //Перечень арифметических операций
```

```
// Коды ощибок ввода пользователя
enum InputErrors{SintaxError,UnpairedParentheses,NotExpression,NotVariable,
        InvalidOperation,InvalidConstName,DivisionByZero};
//Сообщения об ошибках ввода пользователя - их индексы должны соответствовать
//кодам ошибок
char *err[]= {
"Синтаксическая ошибка",
"Непарные круглые скобки",
"Где-то не выражение",
"Где-то не переменная",
"Есть нераспознанная операция",
"Нераспознанное имя константы",
"Получается деление на нуль"
};
//Прототипы Функций
get_token(char **cf,double*cv)throw(InputErrors);
//Приведения к верхнему регистру и очистки от излишеств
void touppercase(char*), clearformula(char *f);
//Вычисления значения выражения
double get_exp(char* f, double t)throw(InputErrors);
#endif
```

Файл dialog.hpp

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by DObject.rc
//
#define DIALOG              101
#define IDC_EDIT1           1008
#define IDC_EDIT2           1009
#define IDC_EDIT3           1010
#define IDC_EDIT4           1011
#define IDC_EDIT5           1012
```

Файл menu.hpp

```
#define CM_INPUTFORMULA  24300
#define CM_ARRAYEVALUTOR  24301
#define CM_GRAPHBUILDER  24302
#define CM_EXIT  24303
#define CM_HELP  24304
#define CM_HELPHELP 24305
#define CM_HELPINTERPR 24306
#define CM_FONT  24307
```

Файл dialog.rc

```
101 DIALOG 0, 0, 318, 110
```

```
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION |
WS_SYSMENU
CAPTION "Ввод формулы в параметрическом виде"
FONT 8, "MS Sans Serif"
LANGUAGE LANG_RUSSIAN, 1
{
 CONTROL "OK", 1, "BUTTON", BS_DEFPUSHBUTTON | WS_CHILD |
WS_VISIBLE | WS_TABSTOP, 32, 88, 50, 14
 CONTROL "Cancel", 2, "BUTTON", BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE
| WS_TABSTOP, 200, 88, 50, 14
 CONTROL "Параметрическое выражение для  X(t) =", 1001, "STATIC", SS_LEFT |
SS_CENTERIMAGE | SS_SUNKEN | WS_CHILD | WS_VISIBLE | WS_GROUP, 7, 4,
145, 14
 CONTROL "Параметрическое выражение для  Y(t) =", 1000, "STATIC", SS_LEFT |
SS_CENTERIMAGE | SS_SUNKEN | WS_CHILD | WS_VISIBLE | WS_GROUP, 6, 21,
146, 13
 CONTROL "Максимальное знач  параметра  tmax =", 1002, "STATIC", SS_LEFT |
SS_CENTERIMAGE | SS_SUNKEN | WS_CHILD | WS_VISIBLE | WS_GROUP, 5, 36,
147, 12
 CONTROL "Минмальное знач  параметра  tmin =", 1018, "STATIC", SS_LEFT |
SS_CENTERIMAGE | SS_SUNKEN | WS_CHILD | WS_VISIBLE | WS_GROUP, 6, 51,
146, 13
 CONTROL "Шаг по параметру при расчетах значений", 1003, "STATIC", SS_LEFT |
SS_CENTERIMAGE | SS_SUNKEN | WS_CHILD | WS_VISIBLE | WS_GROUP, 7, 66,
145, 15
 CONTROL "5.0*sin(t)*cos(t)", 1008, "EDIT", ES_LEFT | ES_AUTOHSCROLL |
WS_CHILD | WS_VISIBLE | WS_BORDER | WS_TABSTOP, 156, 4, 156, 12
 CONTROL "15.0*cos(t)", 1009, "EDIT", ES_LEFT | ES_AUTOHSCROLL |
WS_CHILD | WS_VISIBLE | WS_BORDER | WS_TABSTOP, 155, 19, 156, 14
 CONTROL "2.0*PI", 1010, "EDIT", ES_LEFT | ES_AUTOHSCROLL | WS_CHILD |
WS_VISIBLE | WS_BORDER | WS_TABSTOP, 156, 36, 156, 12
 CONTROL "0.0", 1011, "EDIT", ES_LEFT | ES_AUTOHSCROLL | WS_CHILD |
WS_VISIBLE | WS_BORDER | WS_TABSTOP, 156, 50, 155, 14
 CONTROL "0.04*PI", 1012, "EDIT", ES_LEFT | ES_AUTOHSCROLL | WS_CHILD |
WS_VISIBLE | WS_BORDER | WS_TABSTOP, 156, 66, 155, 14
}
```

ПРОГРАММА № 10. BMP- файлы, битовые изображения, спрайты, элементы игры в Windows.

ЗАДАНИЕ:

В SDK MS Visual C++ приведен пример компьютерной игры FoxBeer (лиса и медведь).

Все визуальные компоненты этой игры (123 фрагмента сборного фона в виде лесной опушки, различные фазы спрайтов медведя и лисы) были созданы в виде отдельных bmp-файлов, а затем записаны в общий файл foxbeer.art.

Структура этого файла такова: вначале лежит количество bmp-файлов в этом файле,

затем идут парами 13-байтовые имена bmp-файлов и их смещения в объединяющем файле, а после этих справочных данных - сами bmp-файлы (288 штук).
Вам предоставляется этот файл и отдельный bmp-файл с лесным пейзажем и предлагается составить программу, которая выводит на экран пейзаж как фон, на котором будут прогуливаться перед сном медведь и лиса.
Дополнительно к приведенному эскизу программы редлагается отпрограммировать следующее:
-управление движением спрайтов с кдавиатуры;
-смену спрайтов в зависимости от действий пользователя и ситуации на экране (в качестве примера используйте игру Microsoft).
-извлекаемые из сборного файла фрагменты сохраните каждый в отдельном файле и просмотрите например в Paint .
-попытайтесь собрать фон из фрагментов, созданных Micrsoft (вместо движения спрайтов по горизонтали лучше бы двигать фон, имитируя параллельное движение движение наблюдателя).

МЕТОДИЧЕСКОЕ ПОСОБИЕ.
1)Файл фонового пейзажа можно сделать ресурсом и затем при необходимости загружать из ресурса - bmpBack=LoadBitmap(hInst,"BackBitmap") например при обработке WM_CREATE - он будет всегда под рукой при необходимости его отобразить.
2)Выбор и чтение сборного файла можно сделать по команде пункта файлового меню.
3)Так как предполагаетя движение спрайтов - нужен дирижер в виде таймера; начало движения можно осуществит по выбору пункта меню или нажатию кнопки в ToolBar - приобработке этой команды желательно создать таймер, а при обработке таймерного сообщения WM_TIMER потребовать перерисовки окна - по этому сигналу спрайты видимо будут менять позу (фазу) и позицию в окне.
4)Таким образом вся фактическая работа будет выплняться при обработке сообщения WM_PAINT:
-рисуется фон из ресурсов или произвольного файла - этот выбор можно предусмотреть через меню;
-анализируем текущую ситуацию на экране (позицию спрайта - чтобы не убежал за пределы окна и его фазу) и в зависимости от этого выбираем фазу спрайта и рисуем его.
Для выполнения каждой работы придется составить подпрогрмму - набор таких подпрограмм предлагается вам для старта в файлах dib.cpp, bear.cpp а далее все зависит от вашей фантазии и квалификации.
*/

Файл bear.cpp

```
#include "bear.hpp"
#include "dib.hpp"
// Прототипы функций
BOOL InitApp(HINSTANCE);
LRESULT CALLBACK  WndProc(HWND, UINT, WPARAM, LPARAM);
// Имя класса окна
```

```
char const szClassName[]   = "BmpinfoClass";
// Заголовок окна
char const szWindowTitle[] = "Спрайты и движение";

////////////////////// Функция WinMain///////////////////////
#pragma argsused
HINSTANCE hInst;
int APIENTRY
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
  LPSTR lpszCmdLine, int nCmdShow)
{  MSG  msg; // структура для работы с сообщениями
  HWND hwnd; // идентификатор главного окна приложения
  hInst=hInstance;
  // Инициализируем приложение
  if(!InitApp(hInstance)) return FALSE;
// После успешной инициализации создаем главное окно
  hwnd = CreateWindow(
    szClassName,         // имя класса окна
    szWindowTitle,       // заголовок окна
    WS_OVERLAPPEDWINDOW, // стиль окна
    CW_USEDEFAULT,       // задаем размеры и расположение
    CW_USEDEFAULT,       // окна, принятые по умолчанию
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    0,0,hInstance, NULL);
  // Если создать окно не удалось, завершаем приложение
  if(!hwnd)  return FALSE;

  // Рисуем главное окно
  ShowWindow(hwnd, nCmdShow  );  UpdateWindow(hwnd);
  /*
  // Запускаем цикл обработки сообщений
  while(GetMessage(&msg, 0, 0, 0))
  {    DispatchMessage(&msg);  }
  */
  for(;;)//Бесконечный цикл до получения WM_QUIT
{
if(PeekMessage(&msg, NULL, 0, 0,PM_REMOVE))//Если в очереди есть сообщения
{
if(msg.message==WM_QUIT) break;//Если это конец работы
TranslateMessage(&msg);
DispatchMessage(&msg);
}
else{}
}
return msg.wParam;}
```

```
/////////////// Инициализация и регистрация класса окна/////////////
BOOL  InitApp(HINSTANCE hInstance)
{  WNDCLASS wc;    // структура для регистрации
  memset(&wc, 0, sizeof(wc));
  wc.lpszMenuName  = "APP_MENU";
  wc.style         = CS_HREDRAW | CS_VREDRAW;
  wc.lpfnWndProc   = (WNDPROC) WndProc;
  wc.cbClsExtra    = 0;
  wc.cbWndExtra    = 0;
  wc.hInstance     = hInstance;
  wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
  wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground =(HBRUSH)GetStockObject(WHITE_BRUSH);
  wc.lpszClassName = (LPSTR)szClassName;
  return RegisterClass(&wc);
}

/////////////// Функция WndProc - обработчик сообщений//////////////
LRESULT CALLBACK
WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{ PAINTSTRUCT ps;      // Структура, описывающая прямоуголник рисовагия
  static HPALETTE hPal;//Для идентификаторов палитр
  switch (msg)         //В зависимости от кода сообщения msg
  {////////////////Выход из программы////////////////////////
case WM_DESTROY:
{//Освобождение памяти
for(USHORT i=0;i<cnt;i++) if(lpfe[i].hDib!=NULL)
GlobalFree(lpfe[i].hDib);
if(lpfe!=NULL)GlobalFree(lpfe);
KillTimer(hwnd,1);//Убийство таймера
PostQuitMessage(0);
return 0;
}
/////////////////При создании главного окна/////////////////
case WM_CREATE:
{//Определяем размеры окна при его создании
LPCREATESTRUCT lpcs=(LPCREATESTRUCT)lParam;
cxClient = lpcs->cx; ; cyClient = lpcs->cy;
//Загружаем  Bitmap  из ресурсов
bmpBack=LoadBitmap(hInst,"BackBitmap");
return 0;}
//////////////////////По тикам таймера//////////////////
case WM_TIMER:
   {//По сигналу таймера требуем перерисовки окна
InvalidateRect(hwnd, NULL,FALSE);
```

```
return 0;
   }
//при изменении размеров окна сохраняем новые  ширину и высоту
    case WM_SIZE:
   {  cxClient = LOWORD(lParam) ; cyClient = HIWORD(lParam) ;
    return 0;
  }
//////////////////Если надо рисовать/////////////////////////
case WM_PAINT:
{//получаем контекст отображения для рисования во внутренней области окна
HDC hdc =BeginPaint(hwnd, &ps );
if(fromBack=='R')
{//Загружаем  Bitmap  из ресурсов
bmpBack=LoadBitmap(hInst,"BackBitmap");
DrawBackR(hdc,bmpBack);
}
if(fromBack=='F')
{//Вызываем специалиста по рисованию файлового фона
if(hBackDib!=NULL) BackPaint(hdc,0,0,hBackDib);
}
//Если отрисована последняя фаза спрайта - начинаем снова с 1-й
if(ibmpBW>ibmpBWEnd) ibmpBW=ibmpBWBegin;
//Нечетный вариант и промежуточная фаза и позиция не крайняя
if((ibmpBW&1)&&(ibmpBWBegin>122) && (ibmpBW<=ibmpBWEnd) && (xBW>0))
{hDib=lpfe[ibmpBW].hDib;//Берем из массива нужную фазу нужного спрайта
DIBPaint(hdc,xBW,cyClient-cyClient/5,hDib);//Рисуем
//DIBPaintBitBlt(hdc, xBW,cyClient-cyClient/5 , hDib);
ibmpBW+=(USHORT)2;                  //Меняем фазу
xBW-=cxClient/70;                   //Меняем позицию
}
//Если наехали на левую границу окна - меняем вариант на четный
if((xBW<=0) && (ibmpBWBegin&1))
{ibmpBWBegin+=1;  ibmpBWEnd+=1;ibmpBW=ibmpBWBegin;}
//Все вышесказанное повторяем для четного варианта и движения вправо
if(!(ibmpBWBegin&1) && (ibmpBWBegin>122) && (ibmpBW<=ibmpBWEnd) &&
(xBW<cxClient))
{hDib=lpfe[ibmpBW].hDib;
DIBPaint(hdc,xBW,cyClient-cyClient/5,hDib);
//DIBPaintBitBlt(hdc, xBW,cyClient-cyClient/5 , hDib);
ibmpBW+=(USHORT)2;xBW+=cxClient/70;
}
if((xBW>=cxClient) && !(ibmpBWBegin&1))
{ibmpBWBegin-=1;ibmpBWEnd-=1;ibmpBW=ibmpBWBegin;}
//Идентичные процедуры для второго спрайта
if(ibmpFW>ibmpFWEnd) ibmpFW=ibmpFWBegin;
if((ibmpFW&1)&&(ibmpFWBegin>122) && (ibmpFW<=ibmpFWEnd) && (xFW>0))
```

```
{hDib=lpfe[ibmpFW].hDib;
DIBPaint(hdc,xFW,cyClient-cyClient/3,hDib);
ibmpFW+=(USHORT)2;
xFW-=cxClient/70;
EndPaint(hwnd, &ps);//освобождаем контекст отображения
return 0;
}
if((xFW<=0) && (ibmpFWBegin&1))
{ibmpFWBegin+=1;ibmpFWEnd+=1;ibmpFW=ibmpFWBegin;}
if(!(ibmpFWBegin&1) && (ibmpFWBegin>122) &&
(ibmpFW<=ibmpFWEnd) && (xFW<cxClient))
{hDib=lpfe[ibmpFW].hDib;
DIBPaint(hdc,xFW,cyClient-cyClient/3,hDib);
ibmpFW+=(USHORT)2;
xFW+=cxClient/70;
EndPaint(hwnd, &ps);//освобождаем контекст отображения
return 0;
}
if((xFW>=cxClient) && !(ibmpFWBegin&1))
{ibmpFWBegin-=1;ibmpFWEnd-=1;ibmpFW=ibmpFWBegin;}
EndPaint(hwnd, &ps);//освобождаем контекст отображения
return 0;
}
//////////////////обработка сообщений от меню//////////
  case WM_COMMAND:
{switch(wParam)
    {
case CM_HELPABOUT:
     {
MessageBox(hwnd,    "Bitmap Working, v.1.0\n"
     "Учебная программа работы с BMP-файлами",
     "About BMPWORK", MB_OK | MB_ICONINFORMATION);
     return 0;
   }
//Выбор файла по стандартному файловому навигатору
case CM_OPENSPRITEFILE:
   {f=OpenSelectFile();                          //выбираем файл
if(f!=NULL)   lpfe= ReadSpriteFile(f);       //читаем файл в память
//Если файл спрайтов успешно прочитан - разблокируем меню выбора спрайтов
if(lpfe!=NULL) EnableMenu-
Item(GetMenu(hwnd),2,MF_ENABLED|MF_BYPOSITION);
return 0;
}
//////////////////////bmp фона из файла//////////////////////////
case CM_FROMFILE:
{fromBack='F'; //Формируем признак источника загрузки - файл
```

```
//Загружаем bitmap из дискового файла
hBackDib=LoadBitmapFromFile();
if(hBackDib!=NULL) InvalidateRect(hwnd, NULL,FALSE);
return 0;
}
////////////////////bmp фона из ресурсов//////////////
case CM_FROMRES:
{fromBack='R'; InvalidateRect(hwnd, NULL,FALSE); return 0; }
////////////////Для движения  создаем таймер/////////
case CM_MOBIL:
{SetTimer(hwnd,1,150,NULL); return 0; }
////////////////////завершаем работу приложения/////////
case CM_FILEEXIT:
{for(USHORT i=0;i<cnt;i++)        //Освобождаем память
if(lpfe[i].hDib!=NULL) GlobalFree(lpfe[i].hDib);
if(lpfe!=NULL)GlobalFree(lpfe);
KillTimer(hwnd,1);            //Таймер
DestroyWindow(hwnd);          //Окно
return 0;
}
////////////////////////////////////////////////Выбор спрайтов////////////////////
//case CM_BM:{ibmpBegin=ibmp=123;ibmpEnd=126;x=0;return 0;}
//case CM_BS:{ibmpBegin=ibmp=127;ibmpEnd=150;x=0;return 0;}
case CM_BW:
{ibmpBW=ibmpBWBegin=151;//Начальная фаза
ibmpBWEnd=174;         //Конечная фаза
xBW=cxClient;
//Разблокируем меню начала движения
EnableMenuItem(GetMenu(hwnd),3,MF_ENABLED|MF_BYPOSITION);
return 0;
}
//Все то же для другого спрайта
case CM_FW:
{ibmpFW=ibmpFWBegin=239; ibmpFWEnd=262;
EnableMenuItem(GetMenu(hwnd),3,MF_ENABLED|MF_BYPOSITION);
return 0;
}
/*
case CM_FB:{ibmpBegin=ibmp=175;ibmpEnd=178;x=0;return 0;}
case CM_FC:{ibmpBegin=ibmp=179;ibmpEnd=180;x=0;return 0;}
case CM_FCD:{ibmpBegin=ibmp=181;ibmpEnd=182;x=0;return 0;}
case CM_FCW:{ibmpBegin=ibmp=183;ibmpEnd=206;x=0;return 0;}
case CM_FJ:{ibmpBegin=ibmp=207;ibmpEnd=210;x=0;return 0;}
case CM_FJT:{ibmpBegin=ibmp=211;ibmpEnd=214;x=0;return 0;}
case CM_FK:{ibmpBegin=ibmp=215;ibmpEnd=216;x=0;return 0;}
case CM_FR:{ibmpBegin=ibmp=217;ibmpEnd=232;x=0;return 0;}
```

```
case CM_FS:{ibmpBegin=ibmp=233;ibmpEnd=234;x=0;return 0;}
case CM_FT:{ibmpBegin=ibmp=235;ibmpEnd=238;x=0;return 0;}
*/
default : return 0;
  }}
//это сообщение приходит при изменении системной палитры . Наше приложение
//в ответ на это сообщение программа заново реализует свою логическую
//палитру и при необходимости перерисовывает окно
 case WM_PALETTECHANGED:
 {//если это ваше окно , передаем управление обработчику
WM_QUERYNEWPALETTE
 if(hwnd==(HWND)wParam)   break;
 }
//в ответ на это сообщение приложение должно
//реализовать свою логическую палитру и обновить окно
 case WM_QUERYNEWPALETTE:
{HDC hdc=GetDC(hwnd);;
HPALETTE hOldPal;
int nChanged;
//выбираем логическую  палитру в контекст отображения. При обработке
//WM_QUERYNEWPALETTE палитра выбирается для активного окна , а при
//обработке сообщения  WM_PALETTECHANGED - для фонового
hOldPal=SelectPalette(hdc, hPal,(msg==WM_QUERYNEWPALETTE)?FALSE:TRUE);
//реализуем логическую палитру и выбираем ее в контекст отображения
nChanged=RealizePalette(hdc);
SelectPalette(hdc, hOldPal, TRUE);
// освобождаем контекст отображения
ReleaseDC(hwnd, hdc); //если были изменения палитры  //перерисовываем
if(nChanged)  InvalidateRect(hwnd, NULL, TRUE);
return nChanged;
 }
default:
break;
}
return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

Файл dib.cpp

```
//функции для работы с файлами в формате DIB
#include "dib.hpp"
//////////////////////B_бор и открытие файла//////////////////
FILE* OpenSelectFile(void)
{
OPENFILENAME ofn;         //структура для выбора файла
char szFile[256];         //буфер для записи пути к выбранному файлу
char szFileTitle[256];    //буфер для записи имени  выбранного файла
```

```
//фильтр расширений имени файлов
char szFilter[256]="ArtFiles\0*.art;\0BitmapFiles\0*.bmp;*.dib;*.rle;\0Any
fILES\0*.*\0\0";
//идентификатор открываемого файла
FILE* f=NULL;
//инициализация имени выбираемого файла не нужна - создаем пустую строку
szFile[0]='\0';
//записываем нулевые значения во все поля структуры для выбора файла
memset(&ofn, 0, sizeof(OPENFILENAME));
//инициализируем нужные нам поля
ofn.lStructSize= sizeof(OPENFILENAME);//размер структуры
ofn.hwndOwner=NULL;                  //идентификатор родительского окна
ofn.lpstrFilter=szFilter;            //адрес строки фильтра
ofn.nFilterIndex=1;                  //номер позиции выбора в начале
//адрес буфера для записи пути выбранного файла
ofn.lpstrFile=szFile;
//размер буфера для записи пути выбранного файла
ofn.nMaxFile=sizeof(szFile);
//адрес буфера для записи имени выбранного файла
ofn.lpstrFileTitle=szFileTitle;
//размер буфера для записи имени выбранного файла
ofn.nMaxFileTitle=sizeof(szFileTitle);
//в качестве начального каталога для поиска выбираем текущий каталог
ofn.lpstrInitialDir=NULL;
//определяем режимы выбора файла
ofn.Flags=OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST|
OFN_HIDEREADONLY;
if(GetOpenFileName(&ofn))            //выбираем входной файл
{f=fopen(ofn.lpstrFile,"rb");        //открываем выбранный файл
if(f==NULL) MessageBox(NULL, "Ошибка открытия файла","Bitmap Info",
                  MB_OK |  MB_ICONHAND);}
//возвращаем идентификатор файла
return f;}
////Чтение файла спрайтов и формирование массива справочных структур
USHORT i;
LPFILEENTRY ReadSpriteFile(FILE* f)
{ULONG pos;
LPFILEENTRY lpFileEntry;
//курсор в виде песочных часов
HCURSOR hCursor=SetCursor(LoadCursor(NULL, IDC_WAIT));
if(f!=NULL)
{//Читаем первое число из файла - количество картинок ?
fread(&cnt,sizeof(ULONG),1,f);
//Выделим память под массив справочных структур (поля - смещения и имена
файлов)
lpFileEntry=(LPFILEENTRY)GlobalAlloc(GMEM_FIXED,cnt*sizeof(FILEENTRY));
```

```
//если мало свободной памяти возвращаем признак ошибки
if(lpFileEntry==NULL)  return NULL;
//Заполняем массив справочных структур
for(i=0;i<cnt;i++)
{fread(&lpFileEntry[i].ofsbit,sizeof(ULONG),1,f);
fread(lpFileEntry[i].fname,13,1,f);
LPSTR uch = strstr((char*)lpFileEntry[i].fname,"BMP");
if((i<287) && (uch==NULL))
{MessageBox(NULL, "Попался вовсе не BMP - файл",
"Bitmap Info", MB_OK | MB_ICONHAND);
return NULL;
}
//Запоминаем тек позицию и становимся на нужную для чтения
if(i>0)
{pos=ftell(f);
//Вычисление размеров
lpFileEntry[i-1].size=lpFileEntry[i].ofsbit- lpFileEntry[i-1].ofsbit;
//Выделяем память для bmp-файла
lpFileEntry[i-1].hDib=(HDIB)GlobalAlloc(GMEM_FIXED, lpFileEntry[i-1].size );
lpFileEntry[i-1].lpBuf=(unsigned char _huge *)GlobalLock(lpFileEntry[i-1].hDib);
//читаем файл в полученный блок памяти
fseek(f,lpFileEntry[i-1].ofsbit,0);
fread(lpFileEntry[i-1].lpBuf,lpFileEntry[i-1].size,1,f);
fseek(f,pos,0); //Возвращаем указатель в зону имен и смещений
GlobalUnlock(lpFileEntry[i-1].hDib);//расфиксируем память
}}
fclose(f); //закрываем файл
SetCursor(hCursor);//востанавливаем курсор
return lpFileEntry;
}  }


///////////////Чтение файла фона//////////////////
HDIB LoadBitmapFromFile()
{HDIB hDib;      //идентификатор глобальной области памяти для размещения файла
DWORD FileSize; //Для размера файла
FILE* fBack=OpenSelectFile();
if(fBack!=NULL)
{//определяем размер файла .
fseek(fBack,0l, 2); FileSize= ftell(fBack);
rewind(fBack);//возвращаем текущую позицию на начало файла
//заказываем глобальный блок памяти размер которого равен длине файла
hDib=(HDIB) GlobalAlloc(GMEM_FIXED, FileSize );
lpBackBuf=(UCHAR*)GlobalLock(hDib);
//если мало свободной памяти возвращаем признак ошибки
if(lpBackBuf==NULL) return NULL;
```

```
fread(lpBackBuf,1,FileSize,fBack);//читаем файл в полученный блок памяти
fclose(fBack); //закрываем файл
GlobalUnlock(hDib);//расфиксируем память
// Определяем тип битового изображения
int nDIBType = DIBType(hDib);
if(!nDIBType) // если ошибка, выдаем сообщение
{MessageBox(NULL, "Ошибка в формате BMP - файла",
"Bitmap Info", MB_OK | MB_ICONHAND);
return NULL; }
return hDib;
}
return NULL;
}
///////////////Определение и проверка формата DIB/////////////
int DIBType(HDIB hDib)
{  LPBITMAPFILEHEADER lpDIBFileHeader;
   LPBITMAPINFOHEADER lpih;
    LPBITMAPCOREHEADER lpch;
    DWORD biSize;
    LPDIB  hDIBPtr;
    int nDIBType;
    if(hDib==NULL)return(-2); //неправильный идентификатор DIB
   //фиксируем память в которой находится DIB
   hDIBPtr=(LPDIB)GlobalLock(hDib);
   if( hDIBPtr==NULL) return (-1);
   lpDIBFileHeader=( LPBITMAPFILEHEADER)hDIBPtr;
   //проверяем тип файла
   if(lpDIBFileHeader->bfType!=0x4d42)
   { GlobalUnlock(hDib);      return 0;     }
    //проверяем размер заголовка
     biSize=(DWORD)(hDIBPtr[sizeof(BITMAPFILEHEADER)]);
    if(biSize==sizeof(BITMAPINFOHEADER)) //40 байт
    {  //это заголовок DIB в формате Windows
     lpih=(LPBITMAPINFOHEADER) (hDIBPtr+ sizeof(BITMAPFILEHEADER) );
       //проверяем основные полязаголовка DIB
       if((lpih->biPlanes==1) && ((lpih->biBitCount==1) ||  (lpih->biBitCount==4)||
        (lpih->biBitCount==8) || (lpih->biBitCount==24)) &&
        ((lpih->biCompression==BI_RGB) || (lpih->biCompression==BI_RLE4 &&
        lpih->biBitCount==4)|| (lpih->biCompression==BI_RLE8 &&
        lpih->biBitCount==8)))
    {      //определяем метод компрессии файла
     if(lpih->biCompression==BI_RGB)     nDIBType=WINRGB_DIB;
   else  if(lpih->biCompression==BI_RLE4)   nDIBType=WINRLE4_DIB;
   else if(lpih->biCompression==BI_RLE8)    nDIBType=WINRLE8_DIB;
   else      nDIBType=0;
  }
```

```
   else      nDIBType=0;
   }
   else if(biSize==sizeof(BITMAPCOREHEADER))  //12 байт
   {    //это заголовок DIB в формате Presentation Manager
     lpch=( LPBITMAPCOREHEADER) (hDIBPtr+ sizeof(BITMAPFILEHEADER) );
        //проверяем основные полязаголовка DIB
        if((lpch->bcPlanes==1)&&  (lpch->bcBitCount==1 ||  lpch->bcBitCount==4 ||
         lpch->bcBitCount==8 ||  lpch->bcBitCount==24))
      {  nDIBType=PM_DIB;      }
    else       nDIBType=0;
   }
    else    nDIBType=0;
   GlobalUnlock(hDib);
   //возвращаем тип файла или признак ошибки
   return  nDIBType;
   }
////////////////определение размера палитры//////////////////
WORD  DIBNumColors(LPDIB lpDib)
{ DWORD dwColorUsed;
 LPBITMAPINFOHEADER  lpih;
 lpih=(LPBITMAPINFOHEADER)(lpDib+sizeof( LPBITMAPFILEHEADER) );
  //количество цветов
  dwColorUsed=(lpih->biClrUsed);
  //если используется палитра уменьшенного размера
  //возвращаем нужный размер
  if(dwColorUsed)return((WORD)dwColorUsed);
  //если количество используемых цветов не указано
  //вычисляем стандартный размер палитры , исходя
  //из количества бит определяющих цвет пиксела
  switch(lpih->biBitCount)
  { case 1:     return 2;
   case 4:     return 16;
   case 8:     return 256;
   default:    return 0; //палитра не используется
   } }
/////////////определение высоты DIB  в пикселах///////////////
WORD  DIBHeight(LPDIB lpDib)
{ LPBITMAPINFOHEADER  lpih;
 lpih=  ( LPBITMAPINFOHEADER)(lpDib+sizeof( LPBITMAPFILEHEADER) );
return lpih->biHeight;
}
///////////////определение ширины DIB  в пикселах//////////////
//---------------------------------------
WORD    DIBWidth (LPDIB lpDib)
{ LPBITMAPINFOHEADER  lpih;
 lpih= ( LPBITMAPINFOHEADER)(lpDib+sizeof(BITMAPFILEHEADER) );
```

```
return lpih->biWidth;
}
//функция DIBFindBits - определение адреса массива бит изображения
LPSTR DIBFindBits(LPDIB lpDib)
{LPBITMAPFILEHEADER lpfh;
LPBITMAPINFOHEADER lpih;
lpfh=(LPBITMAPFILEHEADER)lpDib;
//используем значение указанное в заголовке файла(если оно не равно нулю)
if(lpfh->bfOffBits) return((LPSTR)lpfh+lpfh->bfOffBits);
//вычисляем адрес исходя из размеров заголовков и таблицы цветов
lpih=( LPBITMAPINFOHEADER)(lpDib+sizeof(BITMAPFILEHEADER) );
return ((LPSTR)lpih+lpih-
>biSize+(DWORD)(DIBNumColors(lpDib)*sizeof(RGBQUAD)));
}

//////////////рисование DIB при помощи функции StretchDIBits//////////
BOOL DIBPaint(HDC hdc, int x, int y, HDIB hDib)
{ WORD wHeight, wWidth,w,h;
 LPDIB lpDib;
 LPBITMAPINFOHEADER lpih;
 lpDib=(LPDIB)GlobalLock(hDib);
 if(lpDib==NULL) return (-1);
 lpih=( LPBITMAPINFOHEADER)(lpDib+sizeof(BITMAPFILEHEADER) );
  //определяем размеры DIB
  wHeight=lpih->biHeight;  wWidth=lpih->biWidth;
  w=wWidth/sizeof(COLORREF);h=wHeight/sizeof(COLORREF);
  //Определяем адрес DIB
  COLORREF* adib=(COLORREF*)DIBFindBits(lpDib);
  //рисуем DIB
  StretchDIBits(hdc,  x, y,wWidth/1.5, wHeight/1.5, 0, 0,  wWidth,wHeight,
  adib /*DIBFindBits(lpDib)*/,  (LPBITMAPINFO)lpih,
DIB_RGB_COLORS,MERGEPAINT);
  GlobalUnlock(hDib);
  return TRUE;
}

//////////////////Рисование фона/////////////////////////////////////
BOOL BackPaint(HDC hdc, int x, int y, HDIB hDib)
{LPDIB lpDib;
LPBITMAPINFOHEADER lpih;
lpDib=(LPDIB)GlobalLock(hDib);
if(lpDib==NULL) return (-1);
lpih=( LPBITMAPINFOHEADER)(lpDib+sizeof(BITMAPFILEHEADER) );
//определяем размеры DIB
hBack=lpih->biHeight;  wBack=lpih->biWidth;
//рисуем DIB
```

```
CONST VOID *lpBits=DIBFindBits(lpDib);
StretchDIBits(hdc,  x, y,wBack,hBack,0, 0, wBack,hBack,lpBits,
(LPBITMAPINFO)lpih, DIB_RGB_COLORS,SRCCOPY);
GlobalUnlock(hDib);
return TRUE;
}
////////////////Рисование DIB при помощи функции BitBlt/////////
BOOL DIBPaintBitBlt(HDC hdc, int x, int y, HDIB hDib)
{HBITMAP hbmp;
HDC hMemDC;
WORD wHeight, wWidth;
LPDIB lpDib;
LPBITMAPINFOHEADER lpih;
lpDib=(LPDIB)GlobalLock(hDib);
if(lpDib==NULL) return (-1);
lpih=( LPBITMAPINFOHEADER)(lpDib+sizeof(BITMAPFILEHEADER) );
//определяем размеры DIB
wHeight=lpih->biHeight; wWidth=lpih->biWidth;
//создаем совместимое битовое изображение
hbmp=CreateCompatibleBitmap(hdc, wWidth,wHeight) ;
//создаем совместимый контекст памяти
hMemDC=CreateCompatibleDC(hdc);
//преобразуем DIB в DDB
SetDIBits(hdc, hbmp, 0, wHeight, DIBFindBits(lpDib),(LPBITMAPINFO)lpih,
DIB_RGB_COLORS);
//выбираем DDB в контекст отображения
hbmp=(HBITMAP)SelectObject(hMemDC, hbmp);
//рисуем DIB
BitBlt(hdc, x, y,wWidth,wHeight,hMemDC,0,0,0x008800c6);
//удаляем контексты
DeleteObject(SelectObject(hMemDC, hbmp));DeleteDC(hMemDC);
GlobalUnlock(hDib);
return TRUE;
}
//функция DIBCreatePalette - создаем палитру на базе таблицы цветов  DDB
HPALETTE  DIBCreatePalette(HDIB hDib)
{ LPLOGPALETTE lpPal;
 HPALETTE hPal=NULL;
 HANDLE hLogPal;
 int i, wNumColors;
 LPSTR lpbi;
 LPBITMAPINFO lpbmi;
 if(!hDib) return NULL;
 lpbi=(LPSTR)GlobalLock(hDib);
 lpbmi= ( LPBITMAPINFO)(lpbi+sizeof(BITMAPFILEHEADER) );
 //определяем размер таблицы цветов
```

```
 wNumColors=DIBNumColors(lpbi);
 //если в DIB есть таблица цветов создаем палитру
 if( wNumColors)
{ //заказываем память для палитры
     hLogPal=GlobalAlloc(GHND, sizeof(LOGPALETTE)
          + sizeof(PALETTEENTRY)* wNumColors);
       if(!hLogPal){ GlobalUnlock(hDib); return NULL; }
       //получаем указатель на палитру
       lpPal=(LPLOGPALETTE)GlobalLock(hLogPal);
       lpPal->palVersion=0x300; //заполняем заголовок
       lpPal->palNumEntries= wNumColors;
       for(i=0;i< wNumColors;i++){        //заполняем палитру
         lpPal->palPalEntry[i].peRed= lpbmi->bmiColors[i].rgbRed;
         lpPal->palPalEntry[i].peGreen= lpbmi->bmiColors[i].rgbGreen;
         lpPal->palPalEntry[i].peBlue= lpbmi->bmiColors[i].rgbBlue;
       }
       hPal=CreatePalette(lpPal); //создаем палитру
       if(!hPal){
         GlobalUnlock(hLogPal);  GlobalFree(hLogPal);
          return NULL;
        }
      GlobalUnlock(hLogPal);  GlobalFree(hLogPal);
     }
    GlobalUnlock(hDib);
    //возвращаем идентификатор созданной палитры
    return hPal;
  }

//////////Рисование фона, загруженного из ресурсов//////////////////
void DrawBackR(HDC hdcPaint,HBITMAP bmpBack)
{//Совместимый контекст памяти
hMemDC=CreateCompatibleDC(hdcPaint);
//Выбираем изображение фона в контекст
bmpBackOld=(HBITMAP)SelectObject(hMemDC,bmpBack);
if(bmpBackOld)
{SetMapMode(hMemDC,GetMapMode(hdcPaint));//Совмещаем режимы отображения
//Определяем размеры изображения
POINT pSize,pOrg;
BITMAP bm;
GetObject(bmpBack,sizeof(BITMAP),(LPSTR)&bm);
pSize.x=bm.bmWidth;pSize.y=bm.bmHeight;
//Преобразов коорд для устройства
DPtoLP(hdcPaint,&pSize,1);
pOrg.x=0;pOrg.y=0;
//Преобразов коорд для контекста памяти
DPtoLP(hMemDC,&pOrg,1);
```

```
BitBlt(hdcPaint,0,0,pSize.x,pSize.y,hMemDC,0,0,SRCCOPY); //Рисуем
DeleteObject(SelectObject(hMemDC,bmpBack));
DeleteDC(hMemDC);
}}
```

Файл dib.hpp

```
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include <mem.h>
#include <stdio.h>
#pragma hdrstop
  #define WINRGB_DIB 1
  #define WINRLE4_DIB 2
  #define WINRLE8_DIB 3
  #define PM_DIB 10

  typedef HGLOBAL HDIB;
  typedef  UCHAR*/*unsigned char _huge* */ LPDIB;
//Структура со сведениями о bmp-файлах
  typedef struct
   { UCHAR* lpBuf;    //Адрес в памяти
   UCHAR fname[13];//ИМЯ
   ULONG ofsbit;   //Смещение в art-файле
   DWORD size;     //размер
   HDIB hDib;      //Идентификатор изображения
   }FILEENTRY, *LPFILEENTRY;

int ibmpBW,ibmpFW;          //Текущие фазы спрайтов
int ibmpBWBegin,ibmpFWBegin; //Начальные фазы спрайтов
int ibmpBWEnd,ibmpFWEnd;     //Конечные фазы спрайтов

int xBW,xFW;     //Горизонтальные координаты левых верхних углов спрайтов

int fromBack;           //Источник фона: R - из ресурсов, F - из файла
LPFILEENTRY lpfe;
ULONG cnt;
static HDIB  hDib;
HDC hdcWMPaint;          //Для идентификатора контекста
HDC hMemDCBack;
HDIB hBackDib;
WORD hBack,wBack;        //Размеры изображения фона
LPDIB lpBackBuf;         //указатель на глобальный блок памяти
LPBITMAPFILEHEADER lpfh;  //Указатель на структуру BITMAPFILEHEADER
LPBITMAPINFOHEADER lpih;  //Указатель на структуру BITMAPINFOHEADER
DWORD  dwColorUsed;
LPSTR lpstr;
```

```
FILE* f;                //Идентификатор файла
short cxClient , cyClient;// Размеры внутренней области окна

HDC hMemDC;             //Совместимый контекст памяти
HBITMAP bmpBackOld;
HBITMAP bmpBack;        //Идентификатор фонового рисунка в памяти

//Прототипы функций
  FILE* OpenSelectFile(void) ;      //Выбор и открытие *.bmp  файла
  LPFILEENTRY  ReadSpriteFile(FILE*);//Разборка файла спрайтов
  HDIB  LoadBitmapFromFile();//Загрузка фона из файла
  int   DIBType(HDIB hDib);
  WORD  DIBNumColors(LPDIB lpDib);
  WORD  DIBHeight   (LPDIB lpDib);
  WORD  DIBWidth (LPDIB lpDib);
  HPALETTE  DIBCreatePalette(HDIB hDib);
  BOOL  DIBPaint(HDC hdc, int x, int y, HDIB hDib);
  LPSTR DIBFindBits(LPDIB lpDib);
  BOOL BackPaint(HDC hdc, int x, int y, HDIB hDib);
  void DrawBackR(HDC,HBITMAP);        //Рисование фона
  BOOL DIBPaintBitBlt(HDC hdc, int x, int y, HDIB hDib);
```

Файл bear.hpp

```
#define CM_HELPABOUT  301
#define CM_OPENSPRITEFILE   302
#define CM_FILEEXIT   304
#define CM_FW                   356
#define CM_FT                   355
#define CM_FS                   354
#define CM_FR                   353
#define CM_FK                   352
#define CM_FJT                  351
#define CM_FJ                   350
#define CM_FCW                  349
#define CM_FCD                  348
#define CM_FC                   347
#define CM_FB                   346
#define CM_BW                   345
#define CM_BM                   344
#define CM_BS                   342
#define CM_FROMFILE            11843
#define CM_FROMRES             11842
#define CM_MOBIL               11844
```

Файл ресурсов bear.RC

```
#include "bear.hpp"
```

```
APP_MENU MENU
{
 POPUP "&File"
 {
  MENUITEM "&OpenSpriteFile", 302
  MENUITEM "&SaveAS...", 303
  MENUITEM SEPARATOR
  MENUITEM "E&xit", 304
 }
 POPUP "&Загрузка фона"
 {
  MENUITEM "Из Файла", CM_FROMFILE
  MENUITEM "Из ресурсов", CM_FROMRES
 }
 POPUP "&Выбор спрайта", GRAYED
 {
  MENUITEM "&BSView", CM_BS
  MENUITEM "&BWView", CM_BW
  MENUITEM "&FBView", CM_FB
  MENUITEM "&FCView", CM_FC
  MENUITEM "FCDView", CM_FCD
  MENUITEM "&FCWView", CM_FCW
  MENUITEM "FJView", CM_FJ
  MENUITEM "FJTView", CM_FJT
  MENUITEM "FKView", CM_FK
  MENUITEM "FRView", CM_FR
  MENUITEM "FSView", CM_FS
  MENUITEM "FTView", CM_FT
  MENUITEM "FWView", CM_FW
 }
 MENUITEM "&Движение спрайтов", CM_MOBIL, GRAYED
 POPUP "&Help"
 {
  MENUITEM "&About...", 301
 }
}
BackBitmap BITMAP  back.bmp
```

ПРИЛОЖЕНИЕ. Перечень индивидуальных заданий для докладов на семинарских занятиях.

№ студента в группе	Тема доклада на практическом занятии
1	Отличительные особенности Windows.Компоненты и подсистемы Windows. Интерфейс пользователя. Многозадачность. Управление памятью. Независимость графического интерфейса от оборудова-

	ния. Вызовы функций. Архитектура программ, управляемая событиями. Новые типы данных. Венгерская нотация.
2	Структура главной подпрограммы.Регистрация класса окна. Создание и отображение окна. Цикл обработки сообщений и очередь сообщений. Окна Windows. Оконная функция и обработка сообщений. Основные виды системных сообщений WM_. Завершение работы программы.
3	Контекст устройства. Методы получения описателя (идентификатора) контекста . Сообщение WM_PAINT. Метрические параметры текста. Структура TEXTMETRIC . Функции для рисования текста.
4	Система координат и режимы отображения.
5	Стили окна. Стиль класса окна и стиль окна. Перекрывающиеся, временные и дочерние окна.
6	Клавиатурные сообщения , их параметры и обработка. Виртуальные коды клавиш. Символьные клавиатурные сообщения. Стандарты сиволов ANSI, OEM.
7	Таймер – его назначение, создание, уничтожение, подключение таймера к окну, сообщение WM_TIMER и его обработка. Использование специальной функции таймера с обратным вызовом.
8	Сообщения от мыши и их обработка. Курсор мыши и управление курсором.
9.	Все о ресурсах – что это такое, методы создания и включения в программу, табдицы текстовых строк, пиктограммы, bitmap.
10.	Органы управления – кнопки. Создание кнопок программно и с помощью редактора ресурсов. Стили кнопок . Сообщение WM_COMMAND и его обработка.
11	Полоса просмотра Scrollbar, создание, стили, сообщения от полосы просмотра, инициализация полосы и управление полосой.
12	Редактор текста как стандартный орган управления, создание редактора, стили, посылаемые редактором извещения родительскому окну через WM_COMMAND, типы программных сообщений для редактора текста
13	Список LISTBOX, создание, стили списков, коды извещений, сообщения для списка.
14	Окна (панели) диалога модальные и немодальные, методы создания шаблона диалога с помощью текстового редактора и редактора ресурсов, органы управления на поверхности панели диалога, функция диалога и функции для создания диалога. Сообщения для органов управления – использование функции SendMessage и специальных функций SendDlgItemMessage, SetDlgItemText SetDlgItemInt, GetDlgItemText, GetDlgItemInt и других.
15	Меню, классификация типов меню, методы создания шаблонов меню, сообщения, поступающие от меню, функции для работы с меню – создания, добавления строк, удаления строк, удаления меню, акти-

	визации и блокирования строк меню, получения информации о меню
16	Орган управления общего пользования –Toolbar – подготовка изображений для кнопок, описание кнопок, вызов функции создания окна Toolbar, обработка извещений от Toolbar, сообщения для Toolbar.
17	Особенности работы с файлами в многозадачной среде. Функции windows для работы с файлами. Стандартные панели диалога для выбора и открытия файлов.Функции GetOpenFileName, GetSaveFileName, структура OPENFILENAME
18	Графический интерфейс Windows – GDI – основные понятия, структура GDI, примитивы GDI, контекст и атрибуты контекста – цвет фона, режим фона, режим рисования, цвет текста, шрифт, режимы отображения, начало координат окна и области вывода, функции для изменения и получения атрибутов контекста.
19	Инструменты для рисования -перья – использование стандартных перьев, создание, выбор в контекст и удаление. Режимы рисования. Рисование отрезков и многоугольников.
20.	Рисование геометрических фигур – точки, отрезки, ломаные линии, дуги , сегмента и сектора эллипса, прямоугольников и многоугольников.
21	Цвет и цветовые палитры. Системная цветовая палитра. Выбор цвета без использования палитры.Функция ChooseColo и структура CHOOSECOLOR.
22	Битовые изображения. Битовое изображение в формате DDB. Загрузка изображений из ресурсов приложений. Рисование изображения DDB. Функциz Bitblt и растровые операции. Копирование массива битов в DDB – SetBitmapBits
23	Битовые изображения в формате DIB. Форматы файлов и структур данных. Структуры BITMAPFILEHEADER, BITMAPINFO, BITMAPINFOHEADER, ,иты изображения.
24	Рисование изображений DIB. Загрузка bmp-файла и проверка заголовков. Создание цветовой палитры .Рисование DIB Преобразование DIB в DDB.
25	Шрифты. Классификация шрифтов. Выбор шрифтов в контексте отображения. Определение логического шрифта и структура LOGFONT. Функция ChooseFont и структура CHOOSEFONT.

Литература, использованная при составлении конспекта и подготовке примеров программ.

В. Фролов, Г. В. Фролов. Библиотека системного программиста. Тома 11-14, 22,. Москва, "Диалог-МИФИ", 1993.
Ч. Петзольд. Программирование для Windows 95. Тома 1 и 2.BHV – Санкт – Петербург, 1997.