

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ  
РОССИЙСКОЙ ФЕДЕРАЦИИ

**Комсомольский-на-Амуре государственный университет**

# **ОСНОВЫ WINDOWS API ПРОГРАММИРОВАНИЯ**

Учебно-методическое пособие  
по курсу «Компоненты операционных систем»

Комсомольск-на-Амуре

## Оглавление

1.	Начало работы.....	4
1.1.	Содержимое папки проекта.....	5
1.2.	Код простейшего приложения .....	6
1.3.	Программный интерфейс .....	9
2.	Типы данных Windows.....	11
3.	Механизм сообщений.....	13
3.1.	Формат сообщений.....	13
3.2.	Обработка сообщений.....	13
3.3.	Цикл обработки сообщений .....	15
3.4.	Функции работы с сообщениями.....	16
3.5.	Классификация сообщений по функциональным признакам ....	16
4.	Простейшая Windows программа .....	18
4.1.	Функция WinMain().....	18
4.2.	Функция окна.....	18
5.	Работа с окнами .....	20
5.1.	Стили класса окна .....	20
5.2.	Стили окна, определяющие поведение .....	21
5.3.	Стили окна, определяющие внешний вид .....	24
5.4.	Сообщения для окон .....	25
5.5.	Функции для работы с окнами.....	26
5.6.	MessageBox .....	27
6.	Основы вывода.....	29
7.	Контекст устройства.....	31
7.1.	Цвет фона .....	31
7.2.	Режим фона .....	31
7.3.	Режим рисования.....	32
7.4.	Цвет текста.....	32
7.5.	Шрифт.....	32
7.6.	Перо .....	33
7.7.	Кисть.....	34
7.8.	Режим отображения .....	34
7.9.	Начало системы координат для окна .....	35
7.10.	Начало системы физических координат.....	35
7.11.	Масштаб осей для окна.....	36
7.12.	Масштаб осей физических координат .....	36
8.	Работ с общим контекстом отображения .....	37
8.1.	Общий контекст отображения .....	37
8.2.	Рисование точки .....	38
8.3.	Рисование линий.....	38
8.4.	Рисование дуги эллипса.....	39
8.5.	Рисование геометрических фигур .....	39
8.6.	Области.....	40
9.	Органы управления .....	<b>Ошибка! Закладка не определена.</b>

9.1.	Статический текст .....	42
9.2.	Кнопки .....	42
9.3.	Элемент редактирования текста .....	46
9.4.	Списки .....	48
10.	Работа с мышью.....	50
11.	Таймер.....	52
	Список литературы.....	53

## 1. Начало работы

Для создания приложений можно использовать среду Microsoft Visual Studio. В этой среде информация, необходимая для сохранения, отладки и выполнения приложения хранится в виде проекта. В проекте содержатся имена исходных файлов программы, необходимые библиотечные файлы, список всех опций для компилятора и компоновщика, а также другие средства, используемые для построения программы.

Для того чтобы создать проект надо выполнить следующие действия:

1. В меню **Файл** выберите пункт **Создать** и затем пункт **Проект...**
2. В области **Типы проектов** выберите пункт **Win32**. Затем в области пункт **Проект Win32**.
3. Введите имя проекта. В этом примере будет использоваться имя **Пример**.

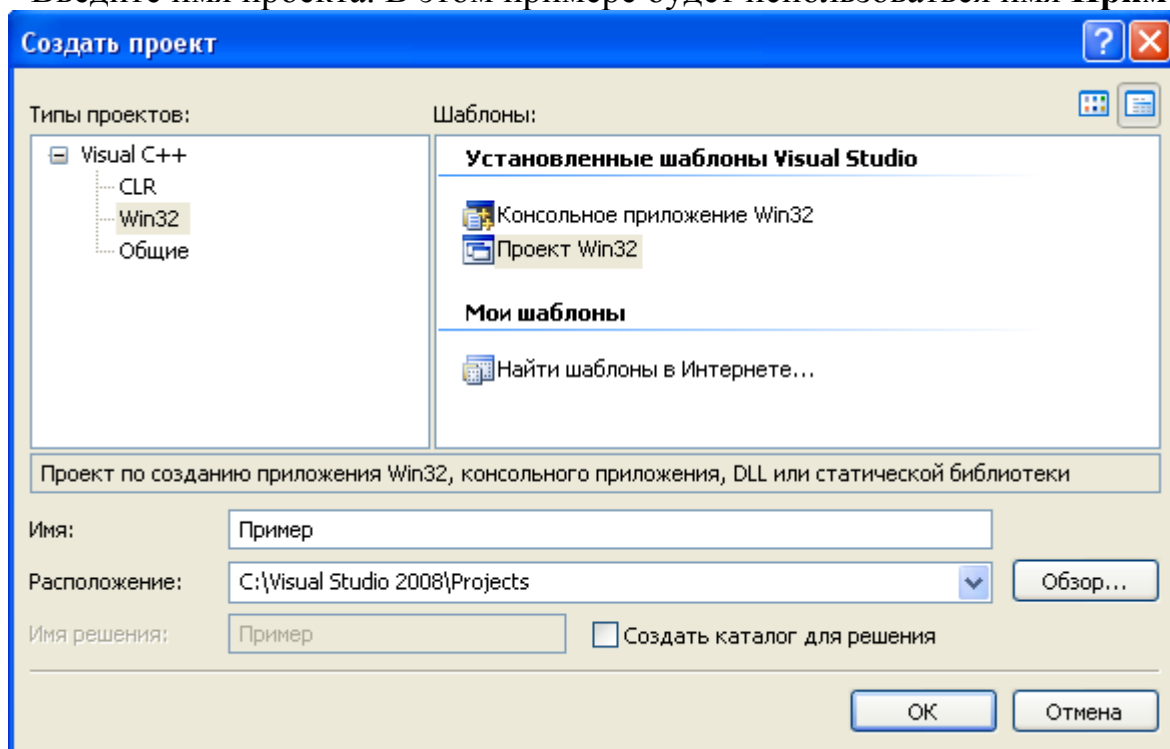


Рис. 1 Вид диалогового окна для создания проекта

4. При создании нового проекта Visual Studio помещает проект в решение. Оставьте имя решения по умолчанию, которое совпадает с именем проекта. Можно принять место размещения по умолчанию, ввести иное место размещения или перейти к каталогу, в который требуется сохранить проект.
5. Для запуска **мастера приложений Win32** нажмите кнопку **OK**.
6. На странице **Общие сведения** диалогового окна **Мастер приложений Win32** нажмите кнопку **Далее**.
7. На странице **Параметры приложения** в поле **Тип приложения** выберите пункт **Приложение Windows**.

После выполнения этих шагов будет создан проект с файлами исходного и вспомогательного кода. Эти файлы отображаются в окне **Обозреватель**

решений. Если окно **Обозреватель решений** не отображается, в меню **Вид** выберите команду **Обозреватель решений**.

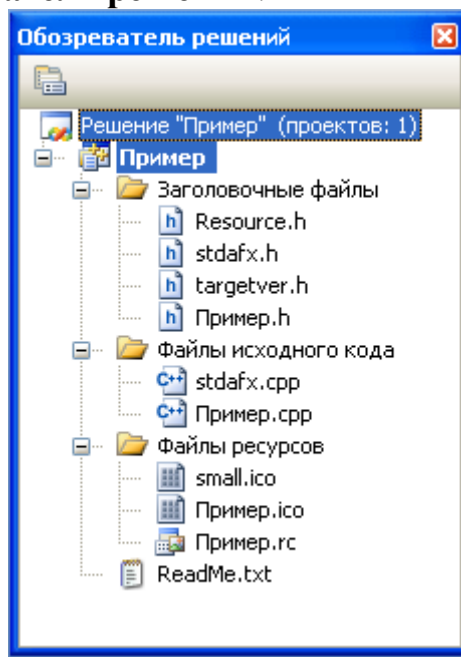


Рис. 2 Вид окна «Обозреватель решений»

Если планируется работать с текстовыми строками, следует изменить свойства проекта, которые были установлены по умолчанию.

В меню **Проект** выберите пункт **Свойства**, далее **Свойства конфигурации/ Общие Набор знаков – Использовать многобайтную кодировку**.

### 1.1. Содержимое папки проекта

Заголовочные файлы		
1	Resource.h	Стандартный файл заголовков, в котором определяются новые идентификаторы ресурсов. В Microsoft Visual C++ выполняется чтение и обновление содержимого этого файла.
2	stdafx.h	Эти файлы используются для построения файла предкомпилируемых заголовков (PCH) Пример.pch и файла предкомпилируемых типов StdAfx.obj.
3	targetver.h	
4	Пример.h	Файл заголовков.
Файлы исходного кода		
	Пример.cpp	Основной исходный файл приложения.
6	stdafx.cpp	Исходный файл, содержащий только стандартные включаемые модули
Файлы ресурсов		
7	Пример.rc	Перечень всех ресурсов Microsoft Windows,

		используемых в программе. К ним относятся значки, растровые изображения и курсоры, хранящиеся в подкаталоге RES. Этот файл можно редактировать непосредственно в Microsoft Visual C++.
8	Пример.ico	Файл значка для приложения (32x32). Этот значок включается в основной файл ресурсов Пример.rc.
9	small.ico	Файл, содержащий уменьшенную версию значка приложения (16x16). Этот значок включается в основной файл ресурсов Пример.rc.
<b>Вспомогательные файлы</b>		
10	ReadMe.txt	В этом файле представлена сводка содержимого всех файлов, входящих в состав приложения Пример.
11	Пример.vcproj	Основной файл проекта VC++, автоматически создаваемый с помощью мастера приложений. В файле представлены сведения о версии Visual C++, использованной при создании файла, а также о параметрах платформы, конфигурации и проекта, заданных с помощью мастера приложений.
12	Пример.sln	Основной файл решения. В решении может быть несколько проектов.
13	Пример.ncb	Файл IntelliSense Database - подсистемы, которая делает подсказки по языку.

## 1.2. Код простейшего приложения

Это приложение создано автоматически с помощью мастера приложений. В него добавлены только комментарии.

```
// Пример.cpp: определяет точку входа для приложения.
//
#include "stdafx.h"
#include "Пример.h"

#define MAX_LOADSTRING 100 // определяем константу MAX_LOADSTRING

// Глобальные переменные:
HINSTANCE hInst; // текущий экземпляр
TCHAR szTitle[MAX_LOADSTRING]; // Текст строки заголовка
TCHAR szWindowClass[MAX_LOADSTRING]; // имя класса главного окна
// Отправить объявления функций, включенных в этот модуль кода:
ATOM MyRegisterClass(HINSTANCE hInstance);
BOOL InitInstance(HINSTANCE, int);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK About(HWND, UINT, WPARAM, LPARAM);

//функция _tWinMain - точка входа в приложение
int APIENTRY _tWinMain(HINSTANCE hInstance,
                      HINSTANCE hPrevInstance,
                      LPTSTR lpCmdLine,
                      int nCmdShow)
{
    //Этот макрос задает значения параметрам, которые не используются
    UNREFERENCED_PARAMETER(hPrevInstance);
```

```

UNREFERENCED_PARAMETER(lpCmdLine);

// TODO: разместите код здесь.
MSG msg; // сообщение
HACCEL hAccelTable; //таблица акселераторов

// Инициализация глобальных строк
LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
LoadString(hInstance, IDC_MY, szWindowClass, MAX_LOADSTRING);
// Вызывать функцию для регистрации класса приложения
MyRegisterClass(hInstance);
// Выполнить инициализацию приложения:
if (!InitInstance (hInstance, nCmdShow))
{
    return FALSE;
}
// Загрузить ресурс - таблицу акселераторов
hAccelTable = LoadAccelerators(hInstance, MAKEINTRESOURCE(IDC_MY));

// Цикл основного сообщения:
while (GetMessage(&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}

return (int) msg.wParam;
}

//
// ФУНКЦИЯ: MyRegisterClass()
//
// НАЗНАЧЕНИЕ: регистрирует класс окна.
//
// КОММЕНТАРИИ:
//
// Эта функция и ее использование необходимы только в случае, если
нужно, чтобы данный код был совместим с системами Win32, не имеющими функции
RegisterClassEx' которая была добавлена в Windows 95. Вызов этой функции важен
для того, чтобы приложение получило "качественные" мелкие значки и установило
связь с ними.
//
ATOM MyRegisterClass(HINSTANCE hInstance)
{
    WNDCLASSEX wcex;

    wcex.cbSize = sizeof(WNDCLASSEX);

    wcex.style          = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc    = WndProc;
    wcex.cbClsExtra     = 0;
    wcex.cbWndExtra     = 0;
    wcex.hInstance     = hInstance;
    wcex.hIcon          = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_MY));
    wcex.hCursor        = LoadCursor(NULL, IDC_ARROW);
    wcex.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
    wcex.lpszMenuName   = MAKEINTRESOURCE(IDC_MY);
    wcex.lpszClassName  = szWindowClass;
    wcex.hIconSm        = LoadIcon(wcex.hInstance,
MAKEINTRESOURCE(IDI_SMALL));

```

```

    return RegisterClassEx(&wcex);
}

//
// ФУНКЦИЯ: InitInstance(HINSTANCE, int)
// НАЗНАЧЕНИЕ: сохраняет обработку экземпляра и создает главное окно.
// КОММЕНТАРИИ:
// В данной функции дескриптор экземпляра сохраняется в глобальной
переменной, а также создается и выводится на экран главное окно программы.
//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd;
    hInst = hInstance; // Сохранить дескриптор экземпляра в глобальной
переменной
    // Создать окно:
    hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
    // Если окно не создалось, то завершить работу
    if (!hWnd)
    {
        return FALSE;
    }
    ShowWindow(hWnd, nCmdShow); //отобразить окно
    UpdateWindow(hWnd); //обновить содержимое окна (сообщение WM_PAINT)
    return TRUE;
}

//
// ФУНКЦИЯ: WndProc(HWND, UINT, WPARAM, LPARAM)
// НАЗНАЧЕНИЕ: обрабатывает сообщения в главном окне.
// WM_COMMAND - обработка меню приложения
// WM_PAINT -Закрасить главное окно
// WM_DESTROY - ввести сообщение о выходе и вернуться.
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message)
    {
        case WM_COMMAND:
            wmId = LOWORD(wParam); // Идентификатор пункта меню или
//элемента управления
            wmEvent = HIWORD(wParam); // Код извещения от органа управления
            // Разобрать выбор в меню
            switch (wmId)
            {
                case IDM_ABOUT:
                    DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
                    break;
                case IDM_EXIT:
                    DestroyWindow(hWnd);
                    break;
                default:
                    return DefWindowProc(hWnd, message, wParam, lParam);
            }
            break;
        case WM_PAINT:

```



```

        hdc = BeginPaint(hWnd, &ps);
        // TODO: добавьте любой код отрисовки...
        EndPaint(hWnd, &ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
// Обработчик сообщений для окна "О программе".
INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM
lParam)
{
    UNREFERENCED_PARAMETER(lParam);
    switch (message)
    {
    case WM_INITDIALOG:
        return (INT_PTR) TRUE;
    case WM_COMMAND:
        if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
        {
            EndDialog(hDlg, LOWORD(wParam));
            return (INT_PTR) TRUE;
        }
        break;
    }
    return (INT_PTR) FALSE;
}

```

### 1.3. Программный интерфейс

API (application programming interface)— это набор готовых констант, структур и функций, используемых при программировании пользовательских приложений и обеспечивающих правильное взаимодействие между приложением и операционной системой.

Функции и константы Win32 API содержатся в трех основных библиотеках:

1. Kernel32.dll. Эта библиотека предназначена для работы с объектами ядра операционной системы и ее функции позволяют управлять памятью, процессами и другими системными ресурсами.
2. User32.dll. Здесь сосредоточены функции для управления окнами, для обработки сообщений, для работы с меню, таймерами.
3. GDI32.dll. Эта библиотека обеспечивает графический интерфейс операционной системы (Graphics Device Interface). Здесь содержатся функции управления выводом на экран и принтер, функции для работы со шрифтами.

#### Плюсы использования API

1. Расширение функциональности программ, разработанных в RAD-средах (например, круглые окна, нестандартная реакция на сообщения).

2. Создание кода, оптимизированного для исполнения в среде Windows (аналог – html-файл, созданный средствами Word).
3. Конечно, легче писать программы, используя библиотеки VCL или MFC, но эти продукты надо купить. Используя функции API, можно создавать программы при помощи условно бесплатного компилятора.

## 2. Типы данных Windows

Windows-программах стандартные типы данных из языков C или C++, такие как `int` или `char*` применяются не часто. Вместо них используются типы данных, определенные в различных библиотечных файлах. Например, `WinDef.h` или `WinNT.h`. Все типы определены при помощи директив `#define` или оператора `typedef`. Такая замена позволяет отделить программный интерфейс Windows от самой операционной системы Windows, и от конкретных реализаций компиляторов языка Си.

Также при программировании под Windows принято использовать префиксы перед именами переменных, указывающие на принадлежность к типу данных. Например, целочисленная переменная-счетчик может быть объявлена так:

```
INT nCount;
```

Префикс «n» в имени переменной используется только для наглядности.

Таблица 1

Некоторые типы данных Windows

Тип данных	Описание	Префикс
APIENTRY	Соглашение о вызовах для системы функций. <code>#define APIENTRY WINAPI</code>	
ATOM	Атом. <code>typedef WORD ATOM;</code>	atm
BOOL	Булевский тип (переменная может иметь значения TRUE или FALSE).	b
WINAPI	Соглашение о вызовах для системы функций. <code>#define WINAPI __stdcall</code>	
WPARAM	Параметр сообщения. <code>typedef UINT_PTR WPARAM;</code>	
LPARAM	Параметр сообщения. <code>typedef LONG_PTR LPARAM;</code>	
HANDLE	Дескриптор объекта. <code>typedef PVOID HANDLE;</code>	h
PVOID	Указатель на любой тип. Определен в <code>WinNT.h</code> как	
LPSTR	Указатель на завершающуюся нулем строку 8-битных символов в кодировке ANSI.	lp
HWND	Дескриптор окна. <code>typedef HANDLE HWND;</code>	
CALLBACK	Соглашение о вызовах для функции обратного вызова.	
LRESULT	Тип результата, возвращаемого из оконной процедуры.	lResult

INT	32-битное целое. typedef int INT;	i
UINT	32-битное беззнаковое целое. typedef unsigned int UINT;	u
BYTE	8-битное беззнаковое целое typedef unsigned char BYTE;	ch
WORD	Беззнаковое целое размером 16 бит typedef unsigned short WORD;	w
DWORD	Беззнаковое целое размером 32 бита typedef unsigned long DWORD;	dw

### 3. Механизм сообщений

Windows является операционной системой, управляемой событиями. Почти все главные и второстепенные события в среде Windows принимают форму сообщений и обрабатываются ОС и приложениями.

#### 3.1. Формат сообщений

Само по себе сообщение представляет собой структуру данных, описанную в файле WinUser.h:

```
typedef struct tagMSG
{
    HWND      hwnd; //идентификатор получателя
    UINT      message; //уникальный для Windows код сообщения
    WPARAM    wParam; //содержимое, зависит от конкретного сообщения
    LPARAM    lParam; //содержимое, зависит от конкретного сообщения
    DWORD     time; // время отправления сообщения
    POINT      pt; //позиция курсора в экранных координатах, когда сообщение
    было отправлено.
} MSG;
```

#### 3.2. Обработка сообщений

Все события, происходящие в системе, обретают форму сообщений. Например, когда вы нажимаете и затем отпускаете клавишу, формируется прерывание, которое обрабатывается драйвером. Он вызывает процедуру в модуле user.exe, которая формирует сообщение, содержащее информацию о событии. Аналогично сообщения создаются при перемещении мыши или в том случае, когда вы нажимаете кнопки на корпусе мыши. Можно сказать, что драйверы устройств ввода/вывода транслируют аппаратные прерывания в сообщения.

Следует отметить, что в Windows используется **многоуровневая система** сообщений.

Сообщения **низкого** уровня вырабатываются, когда вы перемещаете мышь или нажимаете клавиши на корпусе мыши или на клавиатуре. В эти сообщения входит информация **о текущих координатах курсора** мыши или кодах нажатых клавиш. Обычно приложения редко анализируют сообщения низкого уровня. Все эти сообщения передаются операционной системе Windows, которая на их основе формирует сообщения более **высокого** уровня. Когда вы нажимаете кнопку в диалоговом окне приложения Windows, приложение получает сообщение о том, что **нажата кнопка**. Вам не надо постоянно анализировать координаты курсора мыши – Windows сама вырабатывает для вас соответствующее сообщение высокого уровня.

Куда направляются сообщения, созданные драйверами?

Прежде всего, сообщения попадают в **системную очередь** сообщений Windows, реализованную в модуле user.exe. 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 сообщению, 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 из программного интерфейса Windows распределяет его нужной функции окна, которая и выполняет обработку сообщения.

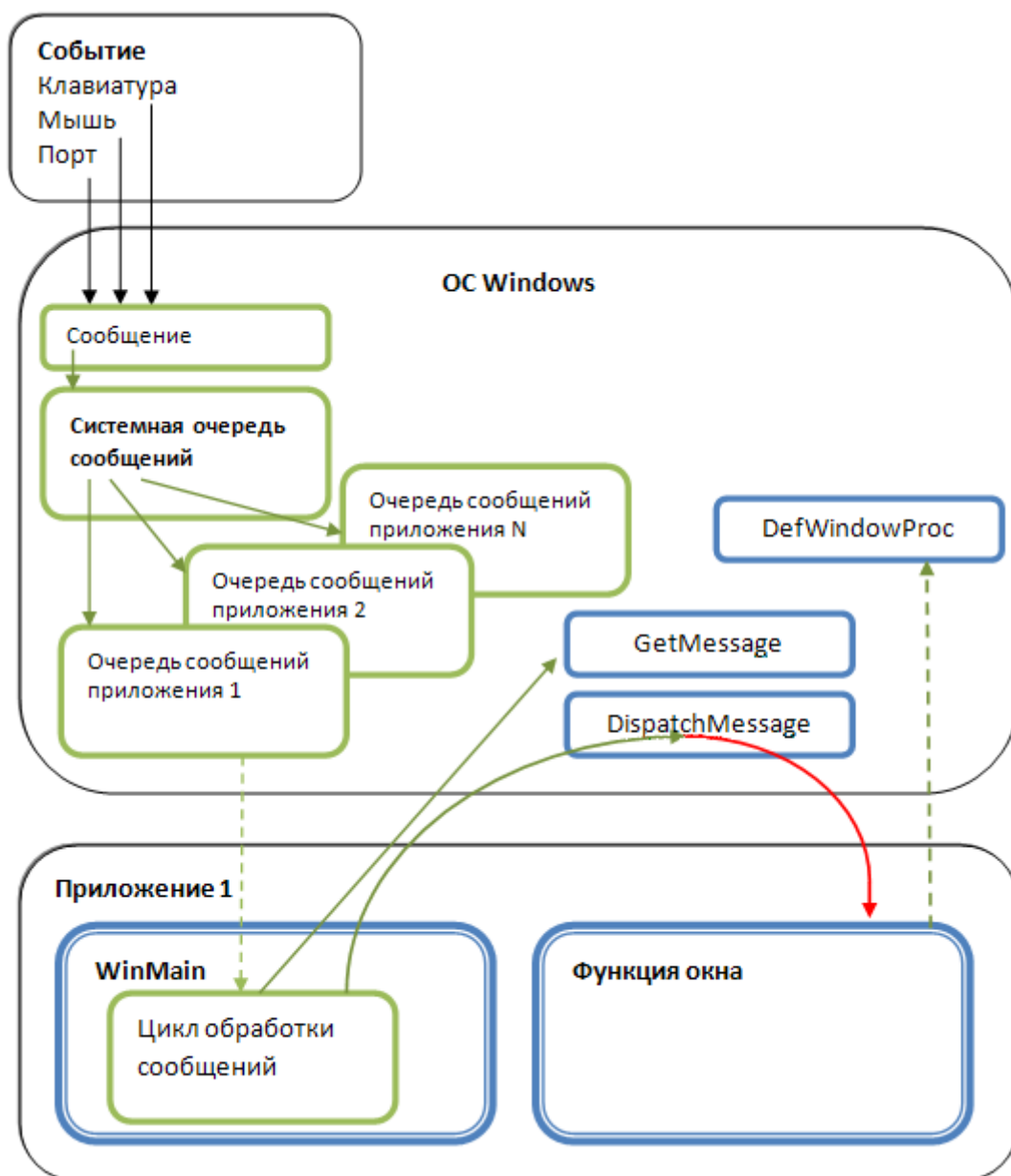


Рис.3 Обработка сообщений

Так как Windows многозадачная операционная система в ней разработан механизм использования несколькими параллельно работающими

приложениями таких ресурсов, как мышь и клавиатура. Так как все сообщения, создаваемые драйверами мыши и клавиатуры, попадают в одну системную очередь сообщений, должен существовать способ распределения этих сообщений между различными приложениями.

Хотя одновременно может быть запущено много приложений, активным является **только одно**. Принято говорить, что это приложение имеет **фокуса ввода**. Все сообщения от клавиатуры и мыши идут в очередь приложения, имеющего фокус ввода.

Перемещать фокус ввода от одного окна к другому можно, нажимая определенные клавиши, или мышью.

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 поступать в очередь приложения, захватившего мышь, вне зависимости от положения курсора мыши.

### 3.3. Цикл обработки сообщений

Простейший цикл обработки сообщений состоит из вызовов двух функций – GetMessage и DispatchMessage.

```
BOOL WINAPI GetMessage(  
    LPMSG lpMsg, // Указатель на структуру MSG, которая получает информацию об  
очередном сообщении из очереди сообщений.  
    HWND hWnd, // дескриптор окна, чьи сообщения должны быть извлечены.  
    UINT wMsgFilterMin, //наименьший номер сообщения  
    UINT wMsgFilterMax //наибольший номер сообщения  
);
```

Если wMsgFilterMin и wMsgFilterMax равны нулю, GetMessage возвращает все доступные сообщения (то есть, фильтрация сообщений не выполняется).

Функция GetMessage предназначена для **выборки сообщения из очереди** приложения. Сообщение выбирается из очереди и записывается в область данных, принадлежащую приложению. Функция возвращает ненулевое значение, если очередное сообщение не WM\_QUIT, и ноль в случае WM\_QUIT.

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 – для распределения приложению достаточно вызвать функцию DispatchMessage.

Вот как выглядит простейший вариант цикла обработки сообщений:

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

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

Коды сообщений определены в файле WinUser.h, включаемом в исходные тексты любых приложений Windows.

### 3.4. Функции работы с сообщениями

Таблица 2

**Функции для работы с сообщениями**

	Имя функции	Назначение
1	GetMessage	Извлекает сообщение из очереди сообщений приложения.
2	DispatchMessage	Передаёт сообщение функции окна.
3	PostMessage	Посылает сообщение в очередь.
4	SendMessage	Посылает сообщение в функцию окна, возврат из функции происходит после обработки этого сообщения.
5	PostQuitMessage	Посылает сообщение о завершении (WM_QUIT) в очередь сообщений приложения.
6	TranslateMessage	Переводит сообщения виртуальных клавиш в символные.

### 3.5. Классификация сообщений по функциональным признакам

Таблица 3

**Системные сообщения**

1	WM_SYSCOMAND	Выбран пункт меню System
2	WM_SYSKEYDOWN	Нажата системная клавиша
	и др.	

Таблица 4

**Сообщения от мыши**

1	WM_NCHITTEST	Это сообщение генерируется драйвером мыши при любых перемещениях мыши.
2	WM_MOUSEMOVE	Перемещение курсора мыши во внутренней области окна
3	WM_NCMOUSEMOVE	Перемещение курсора мыши во внешней области окна
4	WM_MOUSEACTIVATE	Активизация неактивного окна при помощи мыши.
От левой клавиши мыши		
1	WM_LBUTTONDOWNBLCLK	Двойной щелчок левой клавишей мыши во внутренней области окна



2	WM_LBUTTONDOWN	Нажата левая клавиша мыши во внутренней области окна
3	WM_LBUTTONUP	Отпущена левая клавиша мыши во внутренней области окна
4	WM_NCLBUTTONDOWNBLCLK	Двойной щелчок левой клавишей мыши во внешней области окна
5	WM_NCLBUTTONDOWN	Нажата левая клавиша мыши во внешней области окна
6	WM_NCLBUTTONUP	Отпущена левая клавиша мыши во внешней области окна
Аналогичные сообщения приходят от средней и правой клавиш мыши с префиксами WM_RBUTTON и WM_MBUTTON		

Таблица 5

**Сообщения от клавиатуры**

1	WM_KEYDOWN	Нажата клавиша клавиатуры
2	WM_KEYUP	Отжата клавиша
3	WM_SYSKEYDOWN	Нажата системная клавиша
4	WM_SYSKEYUP	Отжата системная клавиша
5	WM_CHAR	Символьное сообщение

Таблица 6

**Сообщения для управление окнами**

1	WM_CREATE	Посылается после создания окна
2	WM_DESTROY	Окно будет уничтожено
3	WM_PAINT	Требуется перерисовка окна
4	WM_CLOSE	Окно будет закрыто
5	WM_MOVE	Посылается после перемещения окна
6	WM_SIZE	После изменения размеров окна
	И др.	

## 4. Простейшая Windows программа

Приложение, которое реагирует на сообщения должно содержать, по крайней мере, 2 функции: WinMain() и WndProc().

### 4.1. Функция WinMain()

Функции WinMain должна выполнить следующие действия:

- зарегистрировать класс окна приложения (возможно и другие классы), здесь же указывается, какая функция будет обрабатывать сообщения этого окна;
- создать главное окно, на основе созданного класса и отобразить его на экране (и другие, подчиненные окна);
- запустить цикл обработки сообщений;
- пока не получено сообщение WM\_QUIT, отправлять сообщения на обработку в функцию окна;
- по сообщению WM\_QUIT завершить работу приложения.

### 4.2. Функция окна

Вторая часть Windows программы – оконная процедура (функция окна). ОС Windows САМА вызывает ее при обработке сообщений, предназначенных для данного окна.

ВСЕ сообщения передаются в функцию окна, но не на все сообщения мы обязаны реагировать. Если в приложении планируется обрабатывать конкретное сообщение, то оно включается в оператор switch. Остальные сообщения обрабатываются по умолчанию. Для этого их пересылают на обработку в функцию DefWindowProc.

Только одно сообщение мы всегда обязаны перехватывать. Это WM\_DESTROY, посылаемое самой Windows в тот момент, когда пользователь закрывает окно (нажимая кнопку закрытия в заголовке окна). Стандартный ответ на WM\_DESTROY заключается в вызове функции PostQuitMessage(0). Это соответствует послке сообщения о выходе из программы со значением нуля в качестве кода возврата.

В качестве параметров функции окна будут передаваться дескриптор текущего окна, код сообщения и его параметры (wParam и lParam). Мы эту функцию НЕ ВЫЗЫВАЕМ.

Вся функция окна в простейшем случае это оператор switch, который анализирует идентификатор сообщений (переменная message), например:

```
switch (message)
{
case WM_PAINT:
    hdc = BeginPaint(hWnd, &ps);
    // TODO: добавьте любой код отрисовки...
    EndPaint(hWnd, &ps);
```

```
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
```

Если тип сообщения не встретился, то такое сообщение отправляется на обработку по умолчанию в функцию DefWindowProc

## 5. Работа с окнами

Чтобы создать окно надо:

1. Зарегистрировать класс окна, указав там стиль (собственный или зарегистрированный в Windows) – стиль класса (поле `ws.style`). Так мы зададим общие свойства окон данного класса.
2. На базе этого класса создать окно (использовать функцию `CreateWindow`), указав при этом стиль окна. Это уточнение вида и поведения окон.
3. Отобразить окно (использовать функцию `ShowWindow`).

### 5.1. Стили класса окна

Всего существует 13 констант, задающих стиль окна. Они начинаются с перффикса `CS_`. Стиль задается константами через битовую операцию «или» (`()`). Используется в `CreateWindow` **первым** параметром.

Таблица 7

**Наиболее употребимые константы**

	Стиль	Описание
1	<code>CS_HREDRAW</code>	Внутренняя область окна должна быть перерисована при изменении ширины окна.
2	<code>CS_VREDRAW</code>	Внутренняя область окна должна быть перерисована при изменении высоты окна.
3	<code>CS_DBLCLKS</code>	Функция окна будет получать сообщения при двойном щелчке клавишей мыши ( <code>double click</code> ).
4	<code>CS_CLASSDC</code>	Необходимо создать единый контекст отображения, который будет использоваться всеми окнами, создаваемыми на базе данного класса.
5	<code>CS_OWNDC</code>	Для каждого окна, определяемого на базе данного класса, будет создаваться отдельный контекст отображения.
6	<code>CS_PARENTDC</code>	Окно будет пользоваться родительским контекстом отображения, а не своим собственным. Родительский контекст - это контекст окна, создавшего другое окно (см. дальше).
7	<code>CS_NOCLOSE</code>	В системном меню окна необходимо запретить выбор функции закрытия окна (строка <code>Close</code> будет отображаться серым цветом, и ее нельзя выбрать).
8	<code>CS_GLOBALCLASS</code>	Данный класс является глобальным и доступным другим приложениям. Другие приложения могут создавать окна на базе этого класса.

9	CS_SAVEBITS	Для данного окна ОС 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.
---	-------------	---

## 5.2. Стили окна, определяющие поведение

Стиль окна задается комбинацией констант с префиксом `WS_`, используется в **третьем** параметре функции `CreateWindow`.

Определено 3 стиля окон, определяющих их поведение – перекрывающиеся окна (overlapped window), временные окна (pop-up window) и дочерние окна (child window).

### 1. Перекрывающиеся окна. Стиль `WS_OVERLAPPED`

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. Для создания перекрывающихся окон определен стиль `WS_OVERLAPPEDWINDOW`, который включает в себя перечисленные выше свойства.

```
#define WS_OVERLAPPEDWINDOW (WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU |  
WS_THICKFRAME | WS_MINIMIZEBOX | WS_MAXIMIZEBOX)
```

Координаты создаваемых функцией `CreateWindow` перекрывающихся окон указываются по отношению ко всему экрану.

### 2. Всплывающие окна. Стиль `WS_POPUP`

Другим базовым стилем является стиль временных окон, которые обычно используются для вывода информационных сообщений и остаются на экране непродолжительное время.

Временные окна, в отличие от перекрывающихся, могут не иметь заголовка. Если для временного окна определен заголовок, оно может иметь и системное меню. Часто для создания временных окон, имеющих рамку, используется стиль `WS_POPUPWINDOW`, определенный следующим образом:

```
#define WS_POPUPWINDOW (WS_POPUP | WS_BORDER | WS_SYSMENU)
```

Временные окна могут иметь окно владельца и могут сами владеть другими окнами.

Приложение Windows может создавать несколько окон, связанных между собой **"узами родства"** и **"отношениями собственности"**.

В функции `CreateWindow` в качестве **8 (восьмого)** параметра функции можно указать так называемый **идентификатор окна-хозяина**. Окно-хозяин уже должно существовать на момент создания второго окна, имеющего владельца.

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, загромождая его.

Начало системы координат, используемой при создании временных окон, находится в левом верхнем углу **экрана**.

Если окно имеет хозяина, это означает, что оно всегда на **поверхности** хозяина, сворачивается, закрывается вместе с ним, но **не перемещается** вместе с ним.

### 3. Дочерние окна. Стил **WS\_CHILDWINDOW**

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 органы управления представляют собой дочерние окна.**

```
#define WS_CHILDWINDOW (WS_CHILD)
```

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 (получают сообщение WM\_PAINT).

Перечислим особенности дочерних окон.

Само собой разумеется, что дочерние окна должны иметь **окно-родителя**. Только дочерние окна могут иметь родителей, перекрывающие и временные окна могут иметь окно-хозяина, но не 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. Само же дочернее окно pošлет родителю сообщение WM\_COMMAND, чтобы сообщить, что с ним происходит.

Приложения Win32 должны разбирать сообщение WM\_COMMAND на составные части следующим образом:

```
case WM_COMMAND:
{
    wId = LOWORD(wParam); // идентификатор элемента управления
```

```

nCmd = HIWORD(wParam); // код извещения
hWnd = (HWND) (UINT) lParam; // идентификатор дочернего окна

// код
}

```

Таблица 8

Параметры сообщения

Источник сообщения	HIWORD (wParam)	LOWORD (wParam)	lParam
Меню	0	Идентификатор пункта меню (IDM_*)	0
Акселератор	1	Идентификатор акселератора (IDM_*)	0
Дочернее окно	Код извещения от органа управления	Идентификатор органа управления (ID_*)	Идентификатор дочернего 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** (тип int). Для удобства восприятия это числовое значение переопределяют символьным выражением. Например,

```
#define IDB_New 11
```

Таким образом, если для какого-либо окна приложения вы создаете несколько дочерних окон, необходимо для каждого окна указать свой идентификатор. 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.

### 5.3. Стили окна, определяющие внешний вид

Комбинация стилей окна, определяющих внешний вид, задается в **третьем** параметре функции CreateWindow при помощи битовой операции | . Не все стили совместимы друг с другом.

Таблица 9

**Константы для задания стилей окна**

	<b>Имя константы</b>	<b>Описание стиля</b>
1	WS_BORDER	Окно с тонкой рамкой.
2	WS_THICKFRAME	Окно будет иметь толстую рамку для изменения размера окна.
3	WS_CAPTION	Окно будет иметь заголовок. Этот стиль несовместим со стилем WS_DLGFRAME.
4	WS_DISABLED	Вновь созданное окно сразу становится заблокированным (не получает сообщения от мыши и клавиатуры).
5	WS_VISIBLE	Создается окно, которое сразу становится видимым. По умолчанию окна создаются невидимыми, и для их отображения требуется вызывать функцию ShowWindow.
6	WS_DLGFRAME	Окно с двойной рамкой без заголовка. Несовместим со стилем WS_CAPTION
7	WS_GROUP	Определяет первый орган управления в группе органов управления. Используется только в диалоговых окнах.
8	WS_MAXIMIZE	Создается окно максимально возможного размера.
9	WS_MINIMIZE	Создается свернутое окно. Этот стиль необходимо использовать вместе со стилем WS_OVERLAPPED
10	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
11	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



12	WS_SYSMENU	Окно будет иметь системное меню и кнопку закрытия окна.
13	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
14	WS_VSCROLL	В окне создается вертикальная полоса просмотра
15	WS_HSCROLL	В окне создается горизонтальная полоса просмотра

#### 5.4. Сообщения для окон

Таблица10

Некоторые сообщения для окон

	Сообщение	Описание
1	WM_ACTIVATE	Посылается как активному, так и неактивному окну. Окно получает фокус ввода, если окно активизировано по нажатию кнопки мыши, то оно получит сообщение WM_MOUSEACTIVATE.
2	WM_CREATE	Посылается после создания, но перед отображением
3	WM_CLOSE	Это сигнал, что программа должна завершаться. При обработке этого сообщения можно спрашивать о желании завершить работу. Если да, то окно уничтожается функцией DestroyWindow.
4	WM_DESTROY	Посылается после удаления окна с экрана, затем это сообщение пошлется всем дочерним окнам.
5	WM_MOVE	Посылается после перемещения окна, в lParam содержатся новые координаты левого верхнего угла клиентской области окна.
6	WM_MOVING	Во время перемещения, lParam – указатель на структуру RECT с экранными координатами перемещаемого прямоугольника.
7	WM_SIZE	Посылается окну после того, как его размер изменился.
8	WM_SHOWWINDOW	Посылается для изменения состояния отображения.

9	WM_COMMAND	Посылается в функцию родительского окна, если орган управления изменяет свое состояние (например, нажали на кнопку).
10	WM_ENABLE	Посылается, когда окно изменяет свое состояние активное/неактивное.
11	WM_QUIT	Посылается функцией PostQuitMessage и означает, что приложение завершает работу. Извлечение этого сообщения из очереди завершает работу цикла обработки сообщений.
12	WM_PAINT	Уведомляет окно о том, что требуется перерисовка всей или части рабочей области окна
13	WM_SETTEXT	Изменить заголовок окна
14	WM_COMMAND	Окну передано сообщение от органа управления или от меню.

### 5.5. Функции для работы с окнами

Таблица 11

**Некоторые функции управления окнами**

	<b>Функция</b>	<b>Описание</b>
1	MoveWindow(HWND hWnd, int nLeft, int nTop, int nWidth, int nHeight, BOOL fRepaint)	Перемещение и изменение размеров окна. Если последний параметр TRUE, то посылается WM_PAINT.
2	EnableWindow(HWND hWnd, BOOL fEnable)	Разрешить/запретить ввод в окно.
3	IsWindowEnable (HWND hWnd)	Проверить доступно ли окно.
4	SetWindowText(HWND hWnd, LPCSTR lpString)	Сменить заголовок окна.
5	ShowWindow(HWND hWnd, int nCmdShow)	Установить состояние отображения. Задается константой SW_*
6	CloseWindow(HWND hWnd)	Свернуть окно.
7	IsZoomed(HWND hWnd)	Проверить свернуто ли окно.
8	Update Window(HWND hWnd)	Послать WM_PAINT в обход очереди сообщений для обновления клиентской области.
9	CreateWindow(LPCSTR lpClassName, LPCSTR lpWindowName, DWORD dwStyle, int x, int y, int nWidth, int	Создать окно.

	nHeight, HWND hWndParent, HMENU hMenu, HINSTANCE hInstance, void FAR* lpParam)	
10	DestroyWindow(HWND hWnd)	Уничтожить окно.
11	BringWindowToTop(HWND hWnd)	Переместить окно наверх.
12	IsWindow(HWND hWnd)	Проверить, есть ли окно с указанным дескриптором.

Некоторые режимы отображения окна для функции ShowWindow:

	Константа, задающая состояние отображения	Описание
1	SW_HIDE	Скрывает окно и активизирует другое окно.
2	SW_SHOWNORMAL	Активизирует и отображает окно.
3	SW_SHOWMAXIMIZED	Активизирует и отображает окно в развернутом виде.
4	SW_SHOWMINIMIZED	Активизирует и отображает окно в свернутом виде.
5	SW_MAXIMIZE	Разворачивает указанное окно.
6	SW_MINIMIZE	Сворачивает указанное окно и активизирует следующее окно верхнего уровня.
6	SW_SHOWNOACTIVATE	Отображает окно как свернутое.

## 5.6. MessageBox

Очень полезная функция MessageBox позволяет выводить на экран небольшое диалоговое окно, в котором можно выводить сообщения для пользователя.

В этой функции 4 параметра:

1. Дескриптор окна-владельца или NULL.
2. Текст сообщения.
3. Заголовок окна.
4. Стилль окна сообщений.

MessageBox(hWnd, " не выбран файл ", "Внимание" , MB\_OK) ;

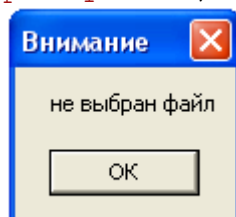


Рис. 4 Вид диалогового окна, созданного функцией MessageBox  
с параметром MB\_OK

```
MessageBox (hWnd, "Вы действительно хотите выйти?", " ", MB_OKCANCEL
```

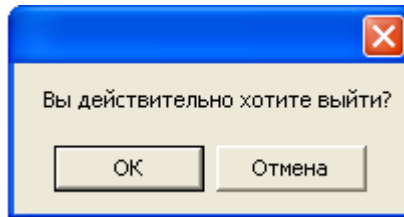


Рис. 5 Вид диалогового окна, созданного функцией MessageBox  
с параметром MB\_OKCANCEL

Итак, вид диалогового окна определяется последним параметром – стилями.

- MB\_OK – это значение по умолчанию, соответствует одной кнопке ОК.
- MB\_OKCANCEL – создается окно с двумя кнопками
- MB\_YESNO – соответствует двум кнопкам: Yes и No.
- MB\_YESNOCANCEL – соответствует трем кнопкам: Yes, No, Cancel.
- И др

Эта функция возвращает одну из констант, соответствующую нажатой кнопке. Значения этих констант следующие:

- IDOK – была нажата кнопка ОК
- IDCANCEL – была нажата кнопка Cancel
- IDYES – была нажата кнопка Yes
- IDNO – была нажата кнопка No
- И др.

### Пример использования

```
int msgboxID = MessageBox (NULL, "Вы действительно хотите выйти?", " ", MB_OKCANCEL);
```

```
switch (msgboxID) // анализируем, какая кнопка нажата
{case IDCANCEL: // нажата кнопка Отмена
    // TODO: add code
    break;
case IDOK: // нажата кнопка ОК
    PostMessage (hWnd, WM_CLOSE, NULL, NULL);
    break;
}
```

## 6. Основы вывода

После того, как окно создано, приложение может использовать его рабочую область как угодно. Если там необходимо что-либо нарисовать или вывести текст, то возникают следующие проблемы:

- неизвестны границы рабочей области,
- неизвестно когда будет происходить вывод,
- кроме непосредственно вывода существуют ситуации, когда содержимое рабочей области надо восстановить (например, окна могут перекрывать друг друга).

Когда приложению требуется обновить рабочую область, то ему посылается сообщение WM\_PAINT. Но в системе слишком много событий, портящих рабочую область (например, курсор мыши).

Часть проблем решает сама ОС Windows. Это перерисовка окна при перемещении и восстановление фона окна.

Область, которую надо перерисовать, определяется в следующей структуре:

```
typedef struct tagPAINTSTRUCT
{
    HDC    hdc; //идентификатор контекста устройства
    BOOL   fErase; //!=TRUE посылается сообщение WM_ERASEBKGRN и фон
обновляется
    RECT   rcPaint; // область отрисовки
    BOOL   fRestore;
    BOOL   fIncUpdate;
    BYTE   rgbReserved[16];
} PAINTSTRUCT;
```

Поле rcPaint, которое представляет собой структуру типа RECT, содержит координаты верхнего левого и правого нижнего угла прямоугольника, внутри которого нужно что-то рисовать.

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

Windows отслеживает координаты той области (invalid rectangle), которой нужно обновление, чтобы не перерисовывать все окно.

Сообщение WM\_PAINT имеет самый низкий приоритет. Оно помещается в очередь приложения, когда область invalid rectangle не пуста и нет других сообщений.

Перед сообщением WM\_PAINT посылается сообщение WM\_ERASEBKGRN. Если это сообщение не обрабатывать, то по умолчанию рабочая область будет закрашиваться кистью, определенной при регистрации окна. Если рисовать в окне во время обработки других сообщений, то перед приходом WM\_PAINT все будет закрашено фоном.

Если перерисовать надо немедленно, то для послыки WM\_PAINT надо послать не в очередь, а напрямую в функцию окна функциями

- пометить область как требующую обновления – `InvalidateRect (NULL, NULL, TRUE)`. Значения параметров означают: главное окно, вся область, фон будет стерт.
- вызвать функцию `UpdateWindow (hWnd)` ;

## 7. Контекст устройства

Контекст устройства выступает в роли связующего звена между приложением и драйвером устройства (дисплей, принтер, плоттер, память) и представляет собой структуру данных размером около 800 байт. Эта структура данных содержит информацию о том, как нужно выполнять операции вывода на данном устройстве (например, цвет и толщину линии, тип системы координат и т. д.)

Контекст отображения можно сравнить с листом бумаги, на котором приложение рисует, а также пишет текст. Инструменты для рисования - это перья, кисти (а также шрифты и даже целые графические изображения), с помощью которых создается изображение. Функции рисования не имеют параметров, указывающих цвет или толщину линии. Такие параметры хранятся в контексте отображения.

Чтобы начать рисовать, контекст отображения надо получить (об этом следующая глава). При этом все атрибуты контекста будут иметь значения, установленные по умолчанию. Рассмотрим, какие параметры есть у контекста и как их изменять.

### 7.1. Цвет фона

Цвет фона (background color ) в контексте отображения соответствует цвету бумаги. Приложение может изменить цвет фона, воспользовавшись функцией `SetBkColor`.

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

Для создания цвета в формате `COLORREF` определен макрос `RGB(r,g,b)`, в котором надо задать значение каждой компоненты цвета (от 0 до 255). Например, красный цвет задается так `RGB(255,0,0)`.

### 7.2. Режим фона

Вы можете установить два режима фона (background mode) – непрозрачный (`OPAQUE`) и прозрачный (`TRANSPARENT`), вызвав функцию `SetBkMode`, указав нужный режим.

```
int WINAPI SetBkMode(HDC hdc, int fnBkMode);
```

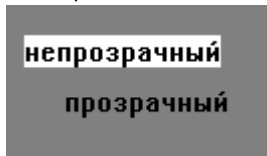


Рис. 6 Режимы фона

По умолчанию выбран режим непрозрачного отображения, при котором в процессе вывода цвет фона удаляется. Например, приложение создало окно с серым фоном и выводит в нем строку текста черного цвета. В этом случае в режиме `OPAQUE` вы увидите черные буквы внутри горизонтальной полосы белого цвета, имеющей высоту, равную высоте букв. В прозрачном режиме

TRANSPARENT аналогия с листом бумаги серого цвета и черным карандашом будет полная

### 7.3. Режим рисования

Когда вы рисуете что-нибудь на бумаге обычным карандашом или фломастером, цвет получившегося изображения соответствует цвету выбранного вами карандаша или фломастера. Иными словами, цвет копируется из инструмента, выбранного для рисования. Именно такой режим рисования (drawing mode) выбран по умолчанию в контекст отображения. При этом новое изображение полностью замещает (закрашивает) то, что находится под ним.

Приложение Windows может выбрать и другие режимы рисования, например, рисование инвертированием цвета фона, рисование черным или белым цветом

Для того чтобы выбрать режим рисования, приложение должно использовать функцию `int WINAPI SetROP2(HDC hdc, int fnDrawMode);`

Таблица 12

**Константы для задания режима рисования**

Значения параметра <b>fnDrawMode</b>	Результат
R2_COPYPEN (по умолчанию)	Цвет нарисованной линии будет такой же, как и цвет пера.
R2_BLACK	Цвет линии черный.
R2_WHITE	Цвет линии белый.
R2_NOP	Бесцветная линия.
R2_NOT	На черном фоне будет нарисована белая линия, а на белом фоне – черная

### 7.4. Цвет текста

По умолчанию в контексте отображения для вывода текста выбран черный цвет. Поэтому, если вы не изменили ни одного атрибута контекста отображения, связанного с текстом и цветом, такие функции, как TextOut и DrawText будут выводить черный текст на белом фоне в непрозрачном режиме.

Для выбора цвета текста приложение должно использовать функцию SetTextColor .

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

### 7.5. Шрифт

Контекст отображения содержит информацию о том, какой шрифт (font) используется для вывода текста. По умолчанию текст выводится системным шрифтом с переменной шириной букв в кодировке ANSI.



С помощью функций `CreateFont`, `CreateFontIndirect` и `SelectObject` приложение может выбрать для вывода текста любой другой шрифт, установленный в операционной системе.

## 7.6. Перо

Для того чтобы нарисовать линию или геометрическую фигуру, приложение Windows должно создать собственное перо (pen) или воспользоваться пером, выбранным в контекст отображения по умолчанию (черное перо шириной в один пиксел).

### Выбор пера

Для выбора встроенного пера лучше всего воспользоваться макрокомандами `GetStockPen` и `SelectPen`, определенными в файле `windowsx.h` так:

```
#define GetStockPen(i) ((HPEN)GetStockObject(i))
#define SelectPen(hdc, hpen) ((HPEN)SelectObject((hdc), (HGDIOBJ)(HPEN)(hpen)))
```

Макрокоманда `GetStockPen` возвращает идентификатор встроенного пера, заданного параметром `i`. Вы можете выбрать для этого параметра одно из следующих значений:

- `BLACK_PEN` – черное перо в один пиксел (для любого режима отображения).
- `WHITE_PEN` – белое перо.
- `NULL_PEN` – невидимое (для рисования границ закрашенных фигур).






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.

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 `fnPenStyle` определяет стиль линии и может принимать одно из следующих значений

<code>PS_SOLID</code>		сплошное	} ширина только 1 пиксел
<code>PS_DASH</code>		штриховое	
<code>PS_DOT</code>		пунктир	
<code>PS_DASHDOT</code>		штрих-пунктир	
<code>PS_NULL</code>		невидимая	
<code>PS_INSIDEFRAME</code>		для обводки фигур	

Параметр `nWidth` определяет ширину пера. Используемая при этом единица длины зависит от режима отображения, поэтому вы можете задавать ширину пера не только в пикселях, но и в долях миллиметра или дюйма.

Таблица 13

#### Алгоритм выбора нового пера

1	Создать переменные типа перо для нового и старого перьев.	<code>HPEN hpen_new, hpen_old;</code>
2	Создать новое перо.	<code>hpen_new = CreatePen(PS_SOLID, 4, RGB(5,12,15));</code>
3	Выбрать новое перо, сохранив при этом старое.	<code>hpen_old = (HPEN) SelectObject(hdc, hpen_new);</code>
4	Использовать новое перо.	
5	Вернуть в контекст старое перо.	<code>SelectObject(hdc, hpen_old);</code>
6	Освободить память, занимаемую новым пером.	<code>DeleteObject(hpen_new);</code>

### 7.7. Кисть

Для закрашивания внутренней области окна приложения или замкнутой геометрической фигуры можно использовать не только различные цвета, но и графические изображения небольшого (8x8 пикселей) размера - кисти (brush).

Для выбора одной из встроенной кисти `GetStockBrush`.

```
#define GetStockBrush(i) ((HBRUSH)GetStockObject(i))
```

В качестве параметра для этой макрокоманды можно использовать следующие значения: `BLACK_BRUSH`, `WHITE_BRUSH`, `GRAY_BRUSH`, `LTGRAY_BRUSH`

Функции для создания новых кистей:

- `CreateSolidBrush(COLORREF)` - сплошная.
- `CreateHatchBrush(int, COLORREF)` - штриховая. Стили для штриховой кисти: `HS_HORIZONTAL`, `HS_VERTICAL`, `HS_FDIAGONAL`, `HS_CROSS`, `HS_DIACROSS`
- `CreatePatternBrush(HBITMAP)` - по образцу.

### 7.8. Режим отображения

Режим отображения, установленный в контексте отображения, влияет на систему координат. Устанавливая различные режимы отображения, приложение может изменять направление и масштаб координатных осей.

По умолчанию в контексте отображения установлен режим отображения `MM_TEXT`. Для этого режима начало системы координат находится в верхнем левом углу внутренней области окна. Ось `x` направлена вправо, ось `y` - вниз. В качестве единицы измерения используется пиксель.

Иногда удобнее использовать обычную систему координат, в которой ось x направлена слева направо, а ось y - снизу вверх. Вы можете выбрать один из нескольких режимов отображения с таким направлением осей. В качестве единицы измерения можно использовать сотые и тысячные доли дюйма, сотые и десятые доли миллиметра и другие величины.

С помощью функции `SetMapMode` приложение может установить в контексте режим отображения, удобный для решения той или иной задачи.

- `MM_LOMETRIC` ось y – вверх, ось x – вправо, одинаковый масштаб по осям 0,1мм.
- `MM_HIMETRIC` ось y – вверх, ось x – вправо, одинаковый масштаб по осям 0,01мм.
- `MM_ISOTROPIC` направления осей можно выбирать, одинаковый масштаб по осям.
- `MM_ANISOTROPIC` все параметры произвольны.

**Физические координаты**, как это следует из названия, имеют непосредственное отношение к физическому устройству вывода. В качестве единицы измерения длины в системе физических координат всегда используется пиксель. 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.

Перевод координат логических координат в физические:

```
LPtoDP(HDC hdc, POINT FAR* lppt, int cPoint);
```

Здесь второй параметр – указатель на массив структур, третий – размер массива.

## **7.9. Начало системы координат для окна**

По умолчанию начало системы координат для окна (window origin) установлено в точку (0,0). Для перемещения начала системы координат окна можно использовать функцию `SetWindowOrg (hDc, nXOrigin, nYOrigin)`.

## **7.10. Начало системы физических координат**

По умолчанию начало системы физических координат установлено в точку (0,0) рабочей области окна. Для перемещения начала системы координат окна можно использовать функцию `SetViewportOrg` или `SetViewportOrgEx`.

```
POINT p; // координаты точки  
SetViewportOrgEx(hdc, 100, 200, &p);
```

### **7.11. Масштаб осей для окна**

Для некоторых режимов отображения приложение может изменять масштаб осей в окне (window extent), устанавливая для него новое значение в контексте отображения.

По умолчанию используется значение (1,1), т. е. используется масштаб 1:1. Приложение может изменить масштаб осей для окна, вызвав функцию SetWindowExt.

### **7.12. Масштаб осей физических координат**

Контекст отображения содержит масштаб осей для физического устройства (viewport extent), который вместе с масштабом осей в окне используется в процессе преобразования координат.

По умолчанию для масштаба осей физических координат используется значение (1,1), т. е. масштаб 1:1. Приложение может изменить масштаб осей физических координат, вызвав функцию SetViewportExt.

## 8. Работ с общим контекстом отображения

Существуют следующие типы контекста отображения:

- общий контекст отображения (common display context);
- контекст отображения для класса окна (class display context);
- личный контекст отображения (private display context);
- родительский контекст отображения (parent display context);
- контекст отображения для окна (window display context);
- контекст физического устройства (device context);
- информационный контекст (information context);
- контекст для памяти (memory device context);
- контекст для метафайла (metafile context).

### 8.1. Общий контекст отображения

Как правило, приложения выполняют всю работу по рисованию во время обработки сообщения WM\_PAINT, хотя часто требуется рисовать и во время обработки других сообщений. Приложение должно придерживаться следующей последовательности действий:

- получение или создание контекста отображения;
- установка необходимых атрибутов в контексте отображения;
- выполнение операций рисования;
- освобождение или удаление контекста отображения.

Для получения навыков работы с контекстом мы рассмотрим общий контекст отображения.

Для получения общего контекста отображения приложение должно вызвать функцию **BeginPaint** (при обработке сообщения WM\_PAINT) или **GetDC** (при обработке других сообщений). При этом перед регистрацией класса окна в поле стиля класса окна в структуре WNDCLASS не должны использоваться значения CS\_OWNDC, CS\_PARENTDC или CS\_CLASSDC : например, **wc.style = 0;**

Функции **BeginPaint** и **GetDC** возвращают контекст отображения для окна **hwnd**:

```
HDC WINAPI BeginPaint(HWND hwnd, PAINTSTRUCT FAR* lpps);  
HDC WINAPI GetDC(HWND hwnd);
```

При этом функция **BeginPaint** подготавливает указанное окно для рисования, заполняя структуру типа PAINTSTRUCT (адрес которой передается через параметр **lpps**) информацией, которую можно использовать в процессе рисования.

После использования контекста отображения, надо освобождать. Если контекст получали, используя функцию **BeginPaint**, то для освобождения надо использовать функцию **void WINAPI EndPaint(HWND hwnd, const PAINTSTRUCT FAR\* lpps)**. Если контекст получали, используя функцию **GetDC**, то используем **int WINAPI ReleaseDC(HWND hwnd)**.

**Плюсы и минусы общего контекста**

**Плюсы:** этот контекст используется чаще всего и поэтому для ускорения доступа к нему Windows использует кеширование (размер кеша достаточен для хранения только пяти контекстов отображения).

**Минусы:** каждый раз, когда приложение получает общий контекст отображения, его атрибуты принимают значения по умолчанию. Если перед выполнением рисования приложение изменит атрибуты контекста отображения, вызвав соответствующие функции GDI, в следующий раз при получении общего контекста отображения эти атрибуты снова примут значения по умолчанию. Поэтому установка атрибутов должна выполняться каждый раз после получения общего контекста отображения.

## 8.2. Рисование точки

Функция рисования точки SetPixel устанавливает цвет точки с заданными координатами:

```
COLORREF WINAPI SetPixel(  
    HDC hdc,           // контекст отображения  
    int nXPos,         // x-координата точки (логическая, а в  
сообщениях           // передаются           физические  
координаты!)  
    int nYPos,         // y-координата точки  
    COLORREF clrref); // цвет точки
```

## 8.3. Рисование линий

Для рисования прямых линий в контексте отображения хранятся координаты текущей позиции пера. Для изменения текущей позиции пера предназначена функция MoveToEx.

```
BOOL WINAPI MoveToEx(  
    HDC hdc,           // идентификатор контекста отображения  
    int x,             // x-координата  
    int y,             // y-координата  
    POINT FAR* lppt); // указатель на структуру POINT, старые  
координаты пера.
```

Чтобы узнать текущую позицию пера, приложение может использовать функцию GetCurrentPositionEx:

```
BOOL WINAPI GetCurrentPositionEx(HDC hdc, POINT FAR* lppt);
```

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 MoveToEx, в точку с координатами (xEnd,yEnd). После того как линия будет нарисована, текущая позиция пера станет равной (xEnd,yEnd).

Особенностью функции LineTo является то, что она немного не дорисовывает линию: эта функция рисует всю линию, не включая точку (xEnd,yEnd).

Можно создать свою собственную функцию рисования линии, например такую:

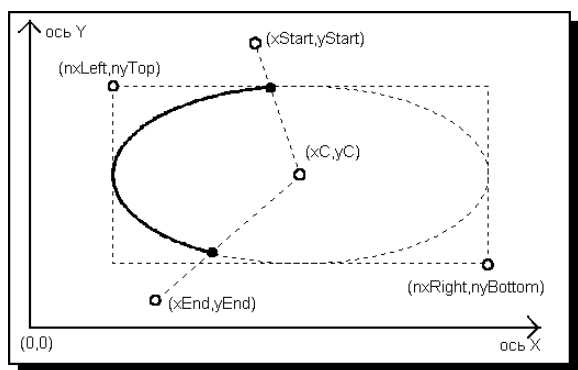
```
BOOL DrawLine(HDC hdc, int x1, int y1, int x2, int y1)
{ POINT pt;
  MoveToEx(hdc, x1, y1, &pt);
  return LineTo(hdc, x2, y2);
}
```

### Рисование ломаной линии

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

## 8.4. Рисование дуги эллипса

К сожалению, возможности рисования кривых линий при помощи функций GDI ограничены - единственная функция Arc позволяет нарисовать дугу



эллипса или окружности:

```
BOOL WINAPI Arc(
    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
```

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;
```

## 8.5. Рисование геометрических фигур

Прямоугольник:

```
BOOL WINAPI Rectangle(HDC hdc, int nLeftRect, int nTopRect, int nRightRect,
nBottomRect);
```

Эллипс:

```
BOOL WINAPI Ellipse(HDC hdc, int nLeftRect, int nTopRect, int nRightRect, int
nBottomRect);
```

Многоугольник:

```
BOOL Polygon(HDC hdc; // идентификатор контекста
const POINT FAR* lppt; //адрес массива, содержащего координаты вершин
int cPoints; //число точек
```

Закрашенный прямоугольник без окрашенной границы:

```
int FillRect(
    HDC hdc;
    const RECT FAR* lprc; //указатель на структуру RECT
    HBRUSH hbr; // дескриптор кисти
)
```

Покраска границы прямоугольника:

```
int FrameRect(HDC hdc; const RECT FAR* lprc; HBRUSH hbr);
```

## 8.6. Области

В интерфейсе GDI есть средства, позволяющие приложениям создавать не отдельные геометрические фигуры, а области. Такие области можно закрашивать или использовать в качестве маски при выводе графического изображения. В последнем случае область называется **областью ограничения**. Она должна быть **выбрана** в контекст отображения.

Для создания прямоугольной области предназначены функции CreateRectRgn и CreateRectRgnIndirect :

```
HRGN WINAPI CreateRectRgn(4 координаты для задания прямоугольника);  
HRGN WINAPI CreateRectRgnIndirect(const RECT FAR* lprc);
```

Можно создать область в виде эллипса (эллипс вписан в заданный прямоугольник):

```
HRGN WINAPI CreateEllipticRgn(int nLeftRect, int nTopRect, int nRightRect,  
int nBottomRect);
```

### Попадание в область

Функция BOOL WINAPI PtInRegion(HRGN hrgn, int nX, int nY) позволяет определить попадает ли точка с координатами (nX, nY) в область hrgn (при попадании функция возвращает TRUE).

```
HRGN hrgn;  
BOOL in;  
//определим текущую координату курсора(передается в lParam)  
x=LOWORD(lParam);  
y=HIWORD(lParam);  
if(PtInRegion(hrgn,x,y)) {...} //курсор внутри области  
else {...} //курсор вне области
```

### Комбинирование областей

Функция CombineRegion позволяет вам изменить существующую область, скомбинировав ее из двух других:

```
int WINAPI CombineRgn(  
    HRGN hrgnDest, // новая область  
    HRGN hrgn1, // первая исходная область  
    HRGN hrgn2, // вторая исходная область  
    int fnCombineMode); // режим комбинирования, задается константой
```

Таблица 14

Значения констант для режима комбинирования

Режим комбинирования	Описание
RGN_AND	Пересечение областей
RGN_OR	Объединение областей
RGN_XOR	Объединение областей с исключением перекрывающихся областей
RGN_DIFF	Область hrgn1, которая не входит в область hrgn2
RGN_COPY	Область hrgn1

В зависимости от результата выполнения операции функция CombineRegion может вернуть одно из следующих значений:

Таблица 15

Значения констант, возвращаемые функцией CombineRegion

Значение	Описание
----------	----------



ERROR	Ошибка
NULLREGION	Новая область пустая
SIMPLEREGION	Новая область не является самопересекающейся (т. е. граница созданной области не пересекает саму себя)
COMPLEXREGION	Создана самопересекающаяся область

### Закрашивание области

```

BOOL WINAPI PaintRgn(HDC hdc, HRGN hrgn); // кисть берется из контекста
BOOL WINAPI FillRgn(HDC hdc, HRGN hrgn, HBRUSH hbrush); // кисть указана
явно

```

### Окраска границы области

```

BOOL WINAPI FrameRgn(HDC hdc, HRGN hrgn, HBRUSH hbrush, int nWidth, int
nHeight);

```

Параметры `nWidth` и `nHeight` определяют, соответственно, ширину и высоту кисти `hrgn` в пикселах, используемой для рисования границы.

### Область ограничения

По умолчанию в контексте отображения задана область ограничения вывода, совпадающая со всей областью вывода. Например, если приложение получило контекст отображения для окна, область ограничения совпадает с внутренней областью (client region) этого окна.

Приложение может создавать область ограничения вывода сложной, практически произвольной, формы, исключая или включая в нее области в виде многоугольников или эллипсов. Это позволяет получить при отображении интересные эффекты, труднодостижимые без использования областей ограничения (правда, ценой снижения скорости вывода изображения).

Приложение может использовать созданную область для маски, **ограничивающей вывод**. Для этого область следует выбрать ее в контекст отображения функцией `SelectClipRgn` :

```

int WINAPI SelectClipRgn(HDC hdc, HRGN hrgn);

```

В качестве значения параметра `hrgn` вы можете использовать значение `NULL`. В этом случае для ограничения вывода будет использована внутренняя область окна.

Отобразить окно в очертаниях региона можно, вызвав функцию `int SetWindowRgn(HWND hWnd, HRGN hRgn, BOOL bRedraw)`. Так можно создавать окна нестандартной формы.

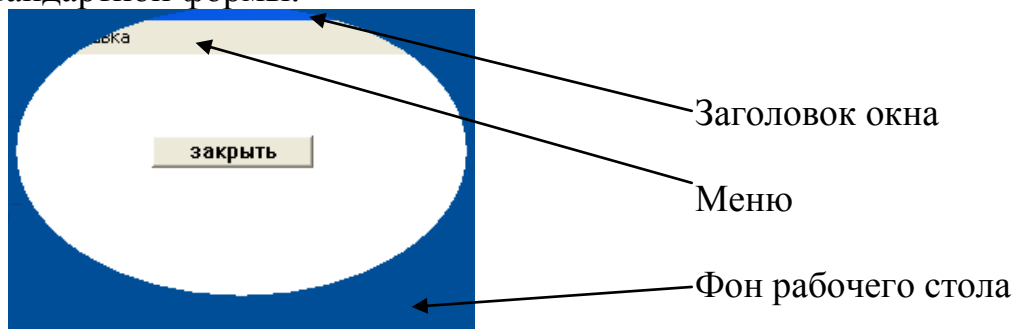


Рис.7 Отображение окна в эллиптической области

## 9. Органы управления

Орган управления – это дочернее окно, которое приложение использует вместе с главным окном, чтобы выполнять обычные задачи ввода и вывода данных. Это кнопки, полосы просмотра, редакторы текстов, меню и т. д. Для создания таких дочерних окон существуют предопределенные классы. Например, дочернее окно-кнопка создается на базе класса "button", редактор текста – "edit" и т.п.

### 9.1. Статический текст

Статические элементы управления это надписи для других органов управления. Создается на основе класса "static".

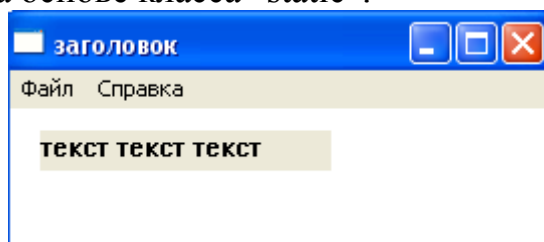


Рис.8 Окно со статическим текстом

Для создания такого элемента надо объявить переменную типа `HWND` (идентификатор окна) и после создания главного окна приложения `hWnd` вызывать функцию `CreateWindow`:

```
HWND hStatic= CreateWindow("static", // класс
    "текст текст текст текст", // текст
    WS_CHILD | WS_VISIBLE ,
    15,10, //x и y,
    145, 20, //ширина и высота
    hWnd, // родитель
    NULL,
    hInstance, NULL);
```

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** указываются третьим параметром при создании элемента.

`SS_LEFT`, `SS_RIGHT`, `SS_CENTER` – выравнивание текста.

`SS_BLACKRECT`, `SS_GRAYRECT` и `SS_WHITERECT` – различные рамки.

Изменить текст в элементе `hStatic` – вызвать `SetWindowText( hStatic, lpszString)`, где вторым параметром указать новый текст.

### 9.2. Кнопки

Кнопки – это органы управления, которые уведомляют родительское окно о том, что пользователь выбрал этот орган управления. Для создания кнопки, приложение должно создать дочернее окно на базе класса "button". Для этого надо объявить переменную для сохранения идентификатора окна и определить идентификатор органа управления.

```
#define IDB_Help 1// идентификатор элемента управления (кнопки)

HWND hWnd, hHelpButton; // идентификаторы окон

hHelpButton = CreateWindow("button", // класс окна
    "Help", // надпись на кнопке
    WS_CHILD | WS_VISIBLE | BS_DEFPUSHBUTTON, // стиль кнопки
    10, 30, // координаты левого верхнего угла кнопки
    40, 20, // ширина и высота
    hWnd, // родительское окно, которое будет получать сообщения от кнопки
    (HMENU)IDB_Help, // идентификатор кнопки
    hInstance, NULL);
```

После этого родительское окно будет получать от кнопки сообщение WM\_COMMAND. Этим сообщением кнопка информирует родительское окно о том, что с ней что-то сделали, например, нажали.

В младшем слове wParam передается идентификатор органа управления (IDB\_Help).

Старшее слово содержит код извещения от органа управления (notification code), по которому можно судить о том, какое действие было выполнено над органом управления.

Для кнопки это – BN\_CLICKED – «нажали».

lParam содержит идентификатор дочернего окна (HWND hHelpButton).


В функции окна в операторе switch добавляется обработка сообщения WM\_COMMAND.








```
case WM_COMMAND:
    {if (LOWORD(wParam) == IDB_Help)
        { // обработчик нажатия кнопки
        }
    }
```

Стиль кнопки влияет на ее внешний вид и поведение.

Таблица 16

**Константы для создания кнопок**

	Стиль кнопки	Описание
1	BS_PUSHBUTTON	Стандартная кнопка. 
2	BS_DEFPUSHBUTTON	Стандартная кнопка, срабатывает по нажатию клавиши Enter. 
3	BS_AUTO3STATE	3 состояния (выключено, включено, не активно), квадратная форма, текст размещается справа, перерисовываются автоматически. 
4	BS_CHECKBOX	2 состояния (выключено/включено),

		квадратная форма, НЕ перерисовываются автоматически.  кнопка  кнопка
5	BS_AUTORADIOBUTTON	2 состояния (выключено/включено), круглая форма, перерисовываются автоматически.  кнопка  кнопка
6	BS_RADIOBUTTON	2 состояния (выключено/включено), круглая форма, НЕ перерисовываются автоматически.  кнопка  кнопка
7	BS_OWNERDRAW	Внешний вид определяется родителем.
9	BS_GROUPBOX	Рамка, не получает сообщений. 

Управление кнопкой из приложения возможно через функции или через посылку соответствующих сообщений.

### Управление кнопками через вызовы функций

#### 1. Переместить – MoveWindow.

Например, при изменении размера окна кнопка будет находиться всегда в его середине. Для этого обрабатываем сообщение WM\_SIZE для главного окна и перемещаем кнопку.

```
case WM_SIZE:
    MoveWindow(hButton, LOWORD(lParam)/2, HIWORD(lParam)/2, ширина, высота,
TRUE);
```

#### 2. Сделать недоступным – EnableWindow (hButton, FALSE)

#### 3. Разблокировать – EnableWindow (hButton, TRUE)

#### 4. Определить, заблокировано ли окно – IsWindowEnabled(hButton)

#### 5. Скрыть – ShowWindow(hButton, SW\_HIDE )

#### 6. Показать – ShowWindow(hButton, SW\_SHOWNORMAL)

#### 7. Изменить заголовок – SetWindowText(hButton, lpszString)

#### 8. Удалить – DestroyWindow(hButton)

### Передача сообщений кнопке

PostMessage – сообщение посылается в очередь и управление сразу передается обратно.

SendMessage – управление вернется только после возврата из функции окна.

### Нажать кнопку

PostMessage(hButton, BM\_SETSTATE, TRUE, 0L) – это «отображение» нажатия.

 - кнопка утоплена.

SendMessage(hButton, BM\_SETSTATE, FALSE, 0L) – это «отображение» отжатия.

**кнопка**

- кнопка выпуклая.

При этом мы увидим, что кнопка нажимается, но действие, за которое она отвечает, не выполнится. Чтобы выполнить действие, за которое отвечает кнопка, надо послать сообщение WM\_COMMAND.

#### «Генерация» нажатия

```
SendMessage(hWnd, WM_COMMAND, IDB_Button1, MAKELPARAM(hButton1, BM_CLICKED));
```

#### Установить состояние переключателя

Установить выключенное состояние: ☐ **кнопка**

```
SendMessage(hCheck1, BM_SETCHECK, 0, 0L);
```

Установить включенное состояние: ☒ **кнопка**

```
SendMessage(hCheck2, BM_SETCHECK, 1, 0L);
```

Сделать не активным: ☒ **кнопка**

```
SendMessage(hCheck3, BM_SETCHECK, 2, 0L);
```

#### Получить состояние переключателя

```
int nCheck=SendMessage(hCheckButton, BM_GETCHECK, 0, 0L);
```

Результат:

0 – выключен (прямоугольник не перечеркнут, в кружке нет точки)

1 – включен

2 – не активен

Переключатели, созданные с типами BS\_CHECKBOX и BS\_RADIOBUTTON автоматически не перерисовываются. Их надо перерисовывать самим при переключении:

```
SendMessage(hCheckButton, BN_SETCHECK, 1, 0L);
```

#### Группировка переключателей

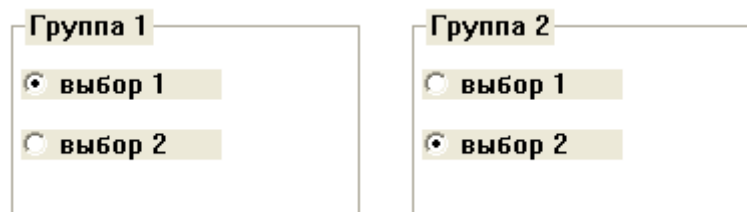


Рис.9 Две группы переключателей

Чтобы создать 2 независимых группы переключателей, надо создавать рамки BS\_GROUPBOX добавив стиль окна WS\_GROUP. Иначе все переключатели будут составлять одну группу.

```
HWND hGroup1 = CreateWindow("button", "Группа 1", WS_CHILD | WS_VISIBLE | BS_GROUPBOX | WS_GROUP, 220, 130, 175, 105, hWnd, NULL, hInstance, NULL);
HWND hWnd1 = CreateWindow("button", "выбор 1", WS_CHILD | WS_VISIBLE | BS_AUTORADIOBUTTON, 225, 160, 100, 15, hWnd, (HMENU) IDB_Button1, hInstance, NULL);
PostMessage(hWnd1, BM_SETSTATE, 1, 0L);
```

```
HWND hWnd2= CreateWindow("button", "выбор 2", WS_CHILD |
WS_VISIBLE | BS_AUTORADIOBUTTON, 225, 190, 100, 15, hWnd,
(HMENU) IDB_Button2, hInstance, NULL);
```

После этого создавать члены этой группы. Объявляем новый элемент Group2 и перечисляем членов новой группы.

### 9.3. Элемент редактирования текста

Поля редактирования текста дают возможность пользователю просматривать и редактировать текст.

$$y = 2 \times + 6$$

Рис.10 Два редактора текста

Создается на основе класса "edit".

```
HWND hEdit;
hEdit = CreateWindow("edit", //класс окна
"2", //текст в окне
WS_CHILD | WS_VISIBLE | WS_BORDER | ES_LEFT, //стили окна
10, 15, //x и y
25, 20, //ширина и высота
hWnd1, // родитель
(HMENU) 4, // идентификатор, задать самим
hInstance, NULL);
```

Таблица 17

Значения констант, для задания стиля

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

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`.

### Получение текста из редактора текста

Для получения текста из простейшего текстового редактора необходимо послать для него сообщение с кодом `EM_GETLINE`:

```
char chText[70]; // массив символов для хранения строки
WORD cbCount=0; // число символов в строке
cbCount = SendMessage(hEdit, EM_GETLINE, 0, (LPARAM)(LPSTR)chText);
```

Текст длиной `cbCount` будет содержаться в строке `chText`.

## 9.4. Списки

Эти органы управления показывают на экране список, из которого пользователь может выбрать один или несколько пунктов.

Создаются на основе класса “listbox” или класса “combobox”, который объединяет “edit” и “listbox”.

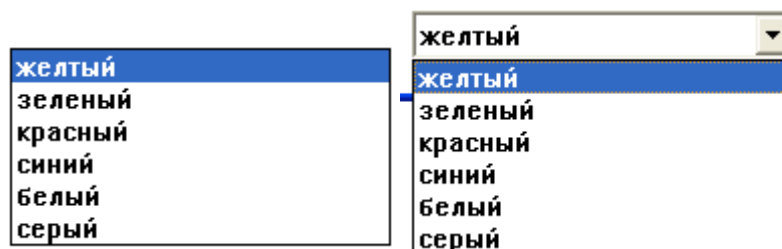


Рис.11 Два списка

Особенности:

1. Эти списки создаются пустыми и их надо заполнять.
2. combobox создается открытым.
3. Для listbox существует стиль `LBS_STANDARD` – это список сортированный по алфавиту `LBS_SORT`, с рамкой `WS_BORDER`, с вертикальной полосой прокрутки `WS_VSCROLL` и ПОСЫЛАЮЩИЙ родительскому окну сообщения `LBS_NOTIFY`. По умолчанию сообщения не посылаются.
4. Вариации несколько колонок `LBS_MULTICOLUMN`, можно выбрать несколько элементов `LBS_MULTIPLESEL` (не подряд) или `LBS_EXTENDEDSEL`(поряд).

Коды извещения передаются в `WM_COMMAND` и показывают, какое действие было совершено над списком.

wParam – идентификатор органа управления

мл. lParam – идентификатор окна

ст. lParam - коды извещения:

`LBN_DBLCLK` – двойной щелчок

`LBN_SETFOCUS` – получен фокус

`LBN_KILLFOCUS` – потерян фокус



LBN\_SELCANCEL – отмена выбора

LBN\_SELCHANGE – выбор другой строки

Работа со списками осуществляется через сообщения, с использованием функции SendMessage.

Таблица 18

**Управление списком**

Действие	
Добавить строку	<code>SendMessage(hList, LB_ADDSTRING, 0, (LPARAM) szString);</code>
Удалить строку	<code>SendMessage(hList, LB_DELETETEXT, Num, 0L);</code>
Определить число элементов	<code>nCol= SendMessage(hList, LB_GETCOUNT, 0, 0L);</code>
Определить номер выбранной строки	<code>nNum= SendMessage(hList, LB_GETCURSEL, 0, 0L);</code>
Копирование строки в буфер	<code>nCol=SendMessage(hList, LB_GETTEXT, (WPARAM) Num, (LPARAM) szBuf);</code>
Запрет перерисовки	<code>SendMessage(hList, LB_SETREDRAW, FALSE, 0L);</code>

## 10. Работа с мышью

Мышь может порождать много сообщений, всего их 22! Однако большинство из них вы можете благополучно проигнорировать, передав эти сообщения функции DefWindowProc.

Таблица 19

**Сообщения, поступающие от мыши**

	Сообщение	Причина возникновения сообщения
1	WM_LBUTTONDOWNBLCLK	Двойной щелчок левой клавишей мыши во внутренней области окна
2	WM_LBUTTONDOWN	Нажата левая клавиша мыши во внутренней области окна
3	WM_LBUTTONUP	Отпущена левая клавиша мыши во внутренней области окна
4	WM_NCLBUTTONDOWNBLCLK	Двойной щелчок левой клавишей мыши во внешней области окна
5	WM_NCLBUTTONDOWN	Нажата левая клавиша мыши во внешней области окна
6	WM_NCLBUTTONUP	Отпущена левая клавиша мыши во внешней области окна
7	WM_MOUSEMOVE	Перемещение курсора мыши во внутренней области окна
8	WM_NCMOUSEMOVE	Перемещение курсора мыши во внешней области окна

Аналогичные 12 сообщения идут от правой и средней кнопок с префиксами WM\_RBUTTON и WM\_MBUTTON.

Таблица 20

**Информация, передаваемая в сообщении от мыши**

wParam	значение, с помощью которого можно определить, какие <b>клавиши</b> на мыши и клавиатуре были нажаты в тот момент, когда произошло событие, связанное с сообщением
LOWORD(lParam)	горизонтальная позиция курсора мыши ( <b>физическая, ОКОННАЯ</b> )
HWORD(lParam)	Вертикальная

Таблица 21

**Значения wParam**

Значение wParam	Причина возникновения сообщения
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>

**Двойным щелчком** (double click) называется **пара одиночных щелчков**, между которыми прошло достаточно мало времени.

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 следующие 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\_LBUTTONDOWNDBLCLK - WM\_LBUTTONUP

Сообщение **WM\_MOUSEMOVE** извещает приложение о перемещении курсора мыши (даже без нажатия). С помощью этого сообщения приложение может, например, рисовать в окне линии вслед за перемещением курсора.

Куда попадают сообщения от мыши?

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 всплывающих окон.

При этом может сложиться ситуация, когда клавиша мыши нажата в **одном окне**, а отпущена в **другом**. Чтобы этого избежать, можно захватить мышь, используя функцию

HWND WINAPI SetCapture(HWND hWnd);

В режиме захвата все сообщения от мыши идут в окно с указанным идентификатором hWnd.

Функция SetCapture возвращает идентификатор окна, которое захватывало мышь до вызова функции или NULL, если такого окна не было.

Освободить мышь – 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.

## 11. Таймер

Существует объект, который с заданной частотой посылает сообщения WM\_TIMER. Это сообщение низкоприоритетное. Оно посылается на обработку в том случае, если в очереди нет других сообщений.

Чтобы создать таймер, надо вызвать функцию SetTimer(hWnd, timer\_id, nTimerout, tmProc ).

Таблица 22

Параметры функции SetTimer

Параметр	Описание
HWND hWnd	таймер относится к этому окну
UINT timer_id	идентификатор таймера, определим сами.
UINT nTimerout	период посылки сообщений таймером, измеряется в мс
TIMERPROC tmProc	функция таймера, обычно NULL. Она нужна, если надо, чтобы в нескольких окнах работал один таймер.

Чтобы удалить таймер, надо вызвать функцию KillTimer(hWnd, timer\_id);

Для наглядности определим символьные идентификаторы таймеров.

```
#define ID_TIMER1 1
#define ID_TIMER2 2
```

Это часть функции окна, в котором будут работать таймеры.

```
case WM_CREATE: // в момент создания главного окна создаем таймеры
    SetTimer(hWnd, ID_TIMER1, 5, NULL); // создаем первый таймер
    SetTimer(hWnd, ID_TIMER2, 1000, NULL); // создаем второй таймер
    return 0;

case WM_TIMER: // сообщение от таймеров
    // по wParam различаем таймеры
    switch (wParam)
    {
        case ID_TIMER1: // выполняем действия, которые относятся к
            первому таймеру
                break;
        case ID_TIMER2: // выполняем действия, которые относятся ко
            второму таймеру
                break;
    }

case WM_DESTROY: // окончание работы
    KillTimer(hWnd, ID_TIMER1); // уничтожаем таймеры
    KillTimer(hWnd, ID_TIMER2);
    PostQuitMessage(0);
    break;
```

## Список литературы

1. Шилдт Г. Программирование на С и С++ для Windows 95, 1996.
2. Рихтер Дж. Windows для профессионалов. 4-е издание. – М.: Русская редакция Microsoft Press, 2004.
3. Румянцев П. В. Азбука программирования в Win32 API. – М.: Горячая Линия – Телеком, 2004.
4. Румянцев П. В. Работа с файлами в Win32 API. – М.: Горячая Линия – Телеком, 2002.
5. Верма Р. Справочник по функциям Win32 API. – М.: Горячая линия – Телеком, 2002.
6. Фролов А., Фролов Г. Операционная система Windows 95 для программиста  
<http://www.frolov-lib.ru/books/bsp.old/v22/index.html>
7. <http://msdn.microsoft.com/>
8. <http://www.firststeps.ru/mfc/winapi/win/apiwind1.html>