



Основы
разработки приложений
с использованием
Windows Forms

Урок №2

Обработка событий мыши и таймера

Содержание

1. Принципы обработки сообщений мыши	3
2. Принципы обработки нажатия клавиш	7
3. Функции для работы с окнами.....	14
4. Работа с таймером	17
5. Перечисление окон	24
6. Ресурсы приложения	33
6.1. Пиктограмма	35
6.2. Курсор	38
7. Обработка ошибок	45
Домашнее задание	49

Материалы урока прикреплены к данному PDF-файлу. Для доступа к материалам, урок необходимо открыть в программе Adobe Acrobat Reader.

1. Принципы обработки сообщений мыши

Как правило, Windows — приложения, обладающие окном, являются интерактивными приложениями, активно использующими пользовательский ввод. При этом в качестве устройства ввода чаще всего используется клавиатура и мышь.

Рассмотрим основные принципы обработки сообщений мыши. Отметим, что оконная процедура приложения получает сообщения мыши в случае, если мышь проходит через окно (даже если окно не активно или не имеет фокуса ввода), а также при щелчке внутри окна. Если мышь перемещается по клиентской области окна, то оконная процедура получает сообщение `WM_MOUSEMOVE`. Если кнопка мыши нажимается или отпускается внутри клиентской области, то оконная процедура получает следующие сообщения:

- `WM_LBUTTONDOWN` — нажата левая кнопка мыши;
- `WM_MBUTTONDOWN` — нажата средняя кнопка мыши;
- `WM_RBUTTONDOWN` — нажата правая кнопка мыши;
- `WM_LBUTTONUP` — отпущена левая кнопка мыши;
- `WM_MBUTTONUP` — отпущена средняя кнопка мыши;
- `WM_RBUTTONUP` — отпущена правая кнопка мыши.
- `WM_LBUTTONDBLCLK` — двойной щелчок левой кнопкой мыши;

- `WM_MBUTTONDOWNBLCLK` — двойной щелчок средней кнопкой мыши;
- `WM_RBUTTONDOWNBLCLK` — двойной щелчок правой кнопкой мыши.

Прокрутка колесика вызывает сообщение `WM_MOUSEWHEEL`.

Для всех этих сообщений значение параметра `lParam` содержит положение мыши. При этом в младшем слове (младшие 2 байта) находится значение координаты `x`, а в старшем слове (старшие 2 байта) — значение координаты `y`. Отсчет координат ведется от левого верхнего угла клиентской области окна. Эти значения можно извлечь из `lParam` при помощи макросов `LOWORD` и `HIWORD`.

```
WORD LOWORD(DWORD dwValue); //Возвращает младшее
                               //слово из указанного
                               //32-битного значения
WORD HIWORD(DWORD dwValue); //Возвращает старшее
                              //слово из указанного
                              // 32-битного значения
```

Следует особо подчеркнуть, что окно будет получать сообщения о двойном щелчке (`DBLCLK`) только в том случае, если стиль соответствующего класса окна содержит флаг `CS_DBLCLKS`. Поэтому перед регистрацией класса окна нужно присвоить полю `style` структуры `WNDCLASSEX` значение, включающее этот флаг. Если класс окна определен без флага `CS_DBLCLKS` и пользователь делает двойной щелчок левой кнопкой мыши,

то оконная процедура последовательно получает сообщения `WM_LBUTTONDOWN`, `WM_LBUTTONUP`, `WM_LBUTTONDOWN` и `WM_LBUTTONUP`. Если класс окна определен с флагом `CS_DBLCLKS`, то после двойного щелчка оконная процедура получит сообщения `WM_LBUTTONDOWN`, `WM_LBUTTONUP`, `WM_LBUTTONDOWNBLCLK` и `WM_LBUTTONUP`.

Модифицировать стиль класса окна можно также вызовом функции API `SetClassLong`, которая позволяет, в общем случае, изменить для класса указанного окна 32-битное значение, связанное со структурой `WNDCLASSEX`:

```
DWORD SetClassLong(
    HWND hWnd,      //дескриптор окна
    int nIndex,      //значение, определяющее,
                    //что нужно изменить, например:
                    //GCL_STYLE — изменить стиль окна,
                    //GCL_HICON — изменить дескриптор курсора,
                    //GCL_HCURSOR — изменить дескриптор иконки
    LONG dwNewLong //новое 32-битное значение
);
```

Функция API `GetClassLong` позволяет получить 32-битное значение из структуры `WNDCLASSEX`, связанной с классом указанного окна:

```
DWORD GetClassLong(
    HWND hWnd, //дескриптор окна
    int nIndex //значение, определяющее,
                //что нужно получить из WNDCLASSEX
);
```

Пример изменения стиля класса окна:

```
UINT style = GetClassLong(hWnd, GCL_STYLE);  
SetClassLong(hWnd, GCL_STYLE, style | CS_DBLCLKS);
```

2. Принципы обработки нажатия клавиш

Одно из широко используемых сообщений порождается при нажатии клавиши. Это сообщение называется **WM_CHAR**. Для сообщений **WM_CHAR** параметр **wParam** содержит ASCII-код нажатой клавиши. **LOWORD (lParam)** содержит количество повторов, генерируемых при удерживании клавиши в нажатом положении. **HWORD (lParam)** представляет собой битовую карту со следующими значениями битов:

- 15: равен 1, если клавиша отпущена, и 0, если она нажата.
- 14: устанавливается, если клавиша уже была нажата перед посылкой сообщения.
- 13: устанавливается в 1, если дополнительно нажата клавиша **<Alt>**.
- 12-9: используется системой.
- 8: устанавливается в 1, если нажата клавиша функциональной или дополнительной части клавиатуры.
- 7-0: код клавиши (scan-код).

Символьные сообщения **WM_CHAR** передаются в оконную процедуру в промежутке между аппаратными сообщениями клавиатуры. Например, если пользователь, удерживая клавишу **<Shift>**, нажимает клавишу **<A>**, отпускает клавишу **<A>** и затем отпускает клавишу **<Shift>**, то оконная процедура получит пять сообщений:

Сообщение	Виртуальная клавиша или ANSI-код
WM_KEYDOWN	Виртуальная клавиша VK_SHIFT
WM_KEYDOWN	Виртуальная клавиша A
WM_CHAR	ANSI-код символа A
WM_KEYUP	Виртуальная клавиша A
WM_KEYUP	Виртуальная клавиша VK_SHIFT

Отметим, что имеет смысл обрабатывать только те аппаратные сообщения `WM_KEYDOWN` и `WM_KEYUP`, которые содержат (в `wParam`) виртуальные коды для клавиш управления курсором, клавиш `<Shift>`, `<Ctrl>`, `<Alt>` функциональных клавиш (`VK_LEFT`, `VK_DOWN`, `VK_SHIFT`, `VK_CTRL`, `VK_MENU`, `VK_RETURN`, `VK_TAB` и т.д.). В то же время аппаратные сообщения для символьных клавиш могут игнорироваться. Ввод информации от символьных клавиш гораздо удобнее обрабатывать, используя символьное сообщение `WM_CHAR`.

Получить состояние указанной виртуальной клавиши можно с помощью функции API `GetKeyState`. Это состояние показывает, нажата ли клавиша, отпущена или переключена в то или иное состояние.

```
SHORT GetKeyState (int nVirtKey);
```

Параметр `nVirtKey` задаёт код виртуальной клавиши. Возвращаемое значение — состояние клавиши, которое закодировано в двух битах. Если старший бит равен 1, то клавиша нажата, в ином случае, она отпущена. Если младший бит равен 1, то клавиша переключена, т.е. переведена во включенное состояние.

Полный перечень всех макросов виртуальных клавиш представлен в файле `winuser.h`.

В качестве примера обработки событий мыши, а также обработки нажатия клавиш рассмотрим следующий код (исходный код прикреплен к pdf-файлу данного урока `SOURCE/MessagesHandler`) ☺.

```
// Файл WINDOWS.H содержит определения, макросы,
// и структуры которые используются при написании
// приложений под Windows.
#include <windows.h>
#include <tchar.h>
//прототип оконной процедуры
LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);
TCHAR szClassWindow[] = TEXT("Каркасное приложение");
/*Имя класса окна*/
INT WINAPI _tWinMain(HINSTANCE hInst,
                    HINSTANCE hPrevInst,
                    LPTSTR lpszCmdLine, int nCmdShow)
{
    HWND hWnd;
    MSG Msg;
    WNDCLASSEX wcl;

    /* 1. Определение класса окна */
    wcl.cbSize = sizeof (wcl); // размер структуры
                          // WNDCLASSEX
    wcl.style = CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS;
    // окно сможет получать сообщения о двойном
    // щелчке (DBLCLK)
    wcl.lpfnWndProc = WindowProc; // адрес оконной
                          // процедуры
    wcl.cbClsExtra = 0; // используется Windows
    wcl.cbWndExtra = 0; // используется Windows
    wcl.hInstance = hInst; // дескриптор данного
                          // приложения
```

```

// загрузка стандартной иконки
wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION);
// загрузка стандартного курсора
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);
// заполнение окна белым цветом
wcl.hbrBackground = (HBRUSH)
    GetStockObject(WHITE_BRUSH);
wcl.lpszMenuName = NULL; // приложение
                        // не содержит меню
wcl.lpszClassName = szClassWindow; // имя класса
                                // окна
wcl.hIconSm = NULL; // отсутствие маленькой иконки

/* 2. Регистрация класса окна */
if (!RegisterClassEx(&wcl)) return 0; // при
                        // неудачной регистрации - выход

/* 3. Создание окна */
// создается окно и переменной hWnd присваивается
// дескриптор окна
hWnd = CreateWindowEx(
    0, // расширенный стиль окна
    szClassWindow, // имя класса окна
    TEXT("Каркас Windows приложения"),
        //заголовок окна
    /* Заголовок, рамка, позволяющая менять размеры,
       системное меню, кнопки развёртывания
       и свёртывания окна
    */
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, // x-координата левого верхнего
                // угла окна
    CW_USEDEFAULT, // y-координата левого
                // верхнего угла окна
    CW_USEDEFAULT, // ширина окна
    CW_USEDEFAULT, // высота окна
    NULL, // дескриптор родительского окна

```

```

    NULL,        // дескриптор меню окна
    hInst,       // идентификатор приложения,
                // создавшего окно
    NULL);      // указатель на область данных
                // приложения

/* 4. Отображение окна */
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd); // перерисовка окна

/* 5. Запуск цикла обработки сообщений */
// получение очередного сообщения из очереди
// сообщений
while (GetMessage(&Msg, NULL, 0, 0))
{
    TranslateMessage(&Msg); // трансляция сообщения
    DispatchMessage(&Msg); // диспетчеризация
                        // сообщений
}
return Msg.wParam;
}

LRESULT CALLBACK WindowProc(HWND hWnd,
                            UINT uMessage,
                            WPARAM wParam,
                            LPARAM lParam)
{
    TCHAR str[50];
    switch(uMessage)
    {
        case WM_LBUTTONDOWNBLCLK:
            MessageBox(
                0,
                TEXT("Двойной щелчок левой кнопкой мыши"),
                TEXT("WM_LBUTTONDOWNBLCLK"),
                MB_OK | MB_ICONINFORMATION);
            break;
    }
}

```

```

case WM_LBUTTONDOWN:
    MessageBox(
        0,
        TEXT("Нажата левая кнопка мыши"),
        TEXT("WM_LBUTTONDOWN"),
        MB_OK | MB_ICONINFORMATION);
    break;
case WM_LBUTTONUP:
    MessageBox(
        0,
        TEXT("Отпущена левая кнопка мыши"),
        TEXT("WM_LBUTTONUP"),
        MB_OK | MB_ICONINFORMATION);
    break;
case WM_RBUTTONDOWN:
    MessageBox(
        0,
        TEXT("Нажата правая кнопка мыши"),
        TEXT("WM_RBUTTONDOWN"),
        MB_OK | MB_ICONINFORMATION);
    break;
case WM_MOUSEMOVE:
    // текущие координаты курсора мыши
    wsprintf(str, TEXT("X=%d Y=%d"),
        LOWORD(lParam), HIWORD(lParam));
    SetWindowText(hWnd, str); // строка
    // выводится в заголовок окна
    break;
case WM_CHAR:
    wsprintf(str, TEXT("Нажата клавиша %c"),
        (TCHAR) wParam); // ASCII-код
    // нажатой клавиши
    MessageBox(0, str, TEXT("WM_CHAR"),
        MB_OK | MB_ICONINFORMATION);
    break;
case WM_DESTROY: // сообщение о завершении
                  // программы

```

```

        PostQuitMessage(0); // посылка сообщения
                               // WM_QUIT

        break;
    default:
        // все сообщения, которые не обрабатываются в
        // данной оконной функции направляются обратно
        // Windows на обработку по умолчанию
        return DefWindowProc(hWnd, uMessage,
                               wParam, lParam);
    }
    return 0;
}

```

Как видно из вышеприведенного кода, при обработке сообщения `WM_MOUSEMOVE` с помощью функции `wsprintf` форматируется строка, содержащая текущие координаты мыши, для последующего вывода в заголовок окна. Вывод строки в заголовок окна осуществляется функцией API `SetWindowText`:

```

BOOL SetWindowText (
    HWND hWnd,           // дескриптор окна, в котором
                        // должен быть изменен текст
    LPCTSTR lpString     // указатель на строку,
                        // содержащую новый текст
);

```

3. Функции для работы с окнами

Win32 API предоставляет широкий набор функций, которые позволяют менять размеры, расположение и характеристики отображения окна. Рассмотрим наиболее употребительные функции.

Функция API `GetWindowRect` позволяет получить размеры прямоугольника окна:

```
BOOL GetWindowRect(  
    HWND hWnd, //дескриптор окна  
    LPRECT lpRect //указатель на структуру RECT  
);  
  
typedef struct tagRECT {  
    LONG left;  
    LONG top;  
    LONG right;  
    LONG bottom;  
} RECT;
```

Поля задают координаты левого верхнего угла (`left`, `top`) и правого нижнего угла (`right`, `bottom`) прямоугольника.

Важно отметить, что в структуре `RECT` будут указаны экранные координаты левого верхнего и правого нижнего углов окна. Отсчёт координат ведётся относительно левого верхнего угла экрана (0,0).

Функция API `GetClientRect` позволяет получить размеры прямоугольника, охватывающего клиентскую (рабочую) область окна:

```

BOOL GetClientRect(
    HWND hWnd, // дескриптор окна
    LPRECT lpRect // указатель на структуру RECT
);

```

В структуре **RECT** будут указаны координаты левого верхнего и правого нижнего углов клиентской области окна. Поскольку отсчёт координат в данном случае ведётся относительно левого верхнего угла рабочей области окна, то координаты (**left, top**) будут равны (0, 0).

Функция API **MoveWindow** позволяет переместить окно, а также изменить его размеры:

```

BOOL MoveWindow(
    HWND hWnd, // дескриптор окна
    int X, // новая координата X левого верхнего
           // угла окна
    int Y, // новая координата Y левого верхнего
           // угла окна
    int nWidth, // новая ширина окна
    int nHeight, // новая высота окна
    BOOL bRepaint // необходимость немедленной
                  // перерисовки окна
);

```

Функция API **BringWindowToTop** активизирует окно и переносит его в верхнее положение, если оно находится позади других окон:

```

BOOL BringWindowToTop(
    HWND hWnd // дескриптор окна
);

```

Для поиска окна верхнего уровня служит функция API `FindWindow`:

```
HWND FindWindow(  
    LPCTSTR lpClassName, // имя класса окна  
    LPCTSTR lpWindowName // заголовок окна  
);
```

Напомним, что имя класса окна, равно как и заголовок окна можно узнать, используя утилиту `Spy++`.

Ещё одна поисковая функция API `FindWindowEx` служит для поиска дочерних окон (например, для поиска элементов управления диалогового окна):

```
HWND FindWindowEx(  
    HWND hwndParent, // дескриптор родительского окна  
    HWND hwndChildAfter, // дескриптор дочернего  
                        // окна, после которого следует  
                        // начать поиск, либо 0 –  
                        // для поиска, начиная с первого  
                        // дочернего окна  
    LPCTSTR lpzClass, // имя класса окна  
    LPCTSTR lpzWindow // заголовок окна  
);
```


4. Работа с таймером

Win API предоставляет возможность использования таймера, что является хорошим способом время от времени «будить» приложение. Это может быть полезным в том случае, если приложение выполняется как фоновое приложение. Для установки таймера необходимо использовать функцию API `SetTimer`:

```
UINT SetTimer(  
    HWND hwnd, // дескриптор окна, которое собирается  
                // использовать таймер  
    UINT nID, // идентификатор устанавливаемого таймера  
    UINT wLength, // временной интервал для таймера  
                // в миллисекундах  
    TIMEPROC lpTFunc // указатель на функцию —  
                    // обработчик прерываний таймера  
);
```

Следует подчеркнуть, что функция, указатель на которую задается параметром `lpTFunc`, является процедурой, определенной в программе и вызываемой при обработке прерываний таймера. Эта функция должна быть определена как `VOID CALLBACK` и иметь такие же параметры, как и оконная функция окна. Однако, если значение `lpTFunc` равно `NULL`, как это чаще всего и бывает, для обработки сообщений таймера будет вызываться оконная процедура главного окна приложения. В этом случае каждый раз по истечении заданного временного интервала в очередь сообщений программы будет помещаться сообщение

`WM_TIMER`, а оконная процедура программы должна будет обрабатывать его так же, как и остальные сообщения. Функция `SetTimer` в случае успешного завершения возвращает значение идентификатора таймера, в противном случае возвращается 0.

Будучи установленным, таймер будет посылать сообщения до тех пор, пока программа не завершится или не вызовет функцию API `KillTimer`:

```
BOOL KillTimer(
    HWND hwnd, // дескриптор окна, использующего таймер
    UINT nID    // идентификатор таймера
);
```

В качестве примера обработки прерываний таймера с использованием сообщения `WM_TIMER` рассмотрим следующий код, который выводит текущие дату и время в заголовок окна. Информация обновляется с интервалом в 1 секунду. Исходный код приложения прикреплен к pdf-файлу данного урока [SOURCE/Timer_WM_TIMER](#).

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

LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);
TCHAR szClassWindow[] = TEXT("Каркасное приложение");
int WINAPI _tWinMain(HINSTANCE hInst,
                    HINSTANCE hPrevInst,
                    LPTSTR lpszCmdLine,
                    int nCmdShow)
{
```

```

HWND hWnd;
MSG Msg;
WNDCLASSEX wcl;
wcl.cbSize = sizeof(wcl);
wcl.style = CS_HREDRAW | CS_VREDRAW;
wcl.lpfnWndProc = WindowProc;
wcl.cbClsExtra = 0;
wcl.cbWndExtra = 0;
wcl.hInstance = hInst;
wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION);
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);
wcl.hbrBackground = (HBRUSH)
    GetStockObject(WHITE_BRUSH);
wcl.lpszMenuName = NULL;
wcl.lpszClassName = szClassWindow;
wcl.hIconSm = NULL;
if (!RegisterClassEx(&wcl))
    return 0;
hWnd = CreateWindowEx(0,
    szClassWindow,
    TEXT("Работа с таймером"),
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    NULL, NULL, hInst, NULL);
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
while(GetMessage(&Msg, NULL, 0, 0))
{
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}
return Msg.wParam;
}

```

```

LRESULT CALLBACK WindowProc (HWND hWnd, UINT message,
                             WPARAM wParam,
                             LPARAM lParam)
{
    static time_t t;
    static TCHAR str[100];
    switch (message)
    {
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        case WM_TIMER:
            // количество секунд, прошедших
            // с 01.01.1970
            t = time(NULL);
            // формирование строки следующего формата:
            // день месяц число часы:минуты:секунды год
            lstrcpy(str, _tctime(&t));
            str[lstrlen(str) - 1] = '\\0';
            // вывод даты и времени в заголовок окна
            SetWindowText(hWnd, str);
            break;
        case WM_KEYDOWN:
            // установка таймера по нажатию
            // клавиши <ENTER>
            if (wParam == VK_RETURN)
                SetTimer(hWnd, 1, 1000, NULL);
            // уничтожение таймера по нажатию
            // клавиши <ESC>
            else if (wParam == VK_ESCAPE)
                KillTimer(hWnd, 1);
            break;
        default:
            return DefWindowProc(hWnd, message,
                                   wParam, lParam);
    }
    return 0;
}

```

Анализируя вышеприведенный код, отметим, что при поступлении сообщения `WM_TIMER` параметр `wParam` содержит идентификатор таймера (приложение может установить несколько таймеров), а `lParam` — адрес функции таймера (если он был задан при установке таймера).

В качестве примера обработки прерываний таймера с использованием функции обратного вызова рассмотрим следующий код.

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

LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);
TCHAR szClassWindow[] = TEXT("Каркасное приложение");
int WINAPI _tWinMain(HINSTANCE hInst,
                    HINSTANCE hPrevInst,
                    LPTSTR lpszCmdLine,
                    int nCmdShow)
{
    HWND hWnd;
    MSG Msg;
    WNDCLASSEX wcl;
    wcl.cbSize = sizeof(wcl);
    wcl.style = CS_HREDRAW | CS_VREDRAW;
    wcl.lpfnWndProc = WindowProc;
    wcl.cbClsExtra = 0;
    wcl.cbWndExtra = 0;
    wcl.hInstance = hInst;
    wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wcl.hCursor = LoadCursor(NULL, IDC_ARROW);
    wcl.hbrBackground = (HBRUSH)
        GetStockObject(WHITE_BRUSH);
    wcl.lpszMenuName = NULL;
    wcl.lpszClassName = szClassWindow;
```

```

wcl.hIconSm = NULL;
if (!RegisterClassEx(&wcl))
    return 0;
hWnd = CreateWindowEx(0,
                      szClassWindow,
                      TEXT("Работа с таймером"),
                      WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                      CW_USEDEFAULT, CW_USEDEFAULT,
                      CW_USEDEFAULT, NULL,
                      NULL, hInst, NULL);
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
while(GetMessage(&Msg, NULL, 0, 0))
{
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}
return Msg.wParam;
}

VOID CALLBACK TimerProc(
    HWND hwnd, // дескриптор окна, которое
               // собирается использовать
               // таймер
    UINT uMsg, // идентификатор сообщения
               // WM_TIMER
    UINT_PTR idEvent, // идентификатор
                     // устанавливаемого таймера
    DWORD dwTime // временной интервал для
                 // таймера в миллисекундах
)
{
    static time_t t;
    static TCHAR str[100];
    t = time(NULL); // количество секунд, прошедших
                   // с 01.01.1970

    // формирование строки следующего формата:
    // день месяц число часы:минуты:секунды год

```

```

lstrcpy(str, _tctime(&t));
str[lstrlen(str) - 1] = '\\0';
SetWindowText(hwnd, str); // вывод даты и времени
                           // в заголовок окна
}

LRESULT CALLBACK WindowProc (HWND hWnd,
                             UINT message,
                             WPARAM wParam,
                             LPARAM lParam)
{
    switch (message)
    {
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        case WM_KEYDOWN:
            // установка таймера по нажатию
            // клавиши <ENTER>
            if (wParam == VK_RETURN)
                SetTimer(hWnd, 1, 1000, TimerProc);
            // уничтожение таймера по нажатию
            // клавиши <ESC>
            else if (wParam == VK_ESCAPE)
                KillTimer(hWnd, 1);
            break;
        default:
            return DefWindowProc(hWnd, message,
                                   wParam, lParam);
    }
    return 0;
}

```

Исходный код приложения прикреплен к pdf-файлу данного урока [SOURCE/Timer_CALLBACK-функция](#).

5. Перечисление окон

Как известно, Win API предоставляет средства для поиска окон, как верхнего уровня, так и дочерних, на основании класса окна и заголовка окна. Речь идёт о рассмотренных ранее функциях `FindWindow` и `FindWindowEx`, возвращающих дескриптор окна верхнего уровня (т.е. окна, не имеющего «родителя») и дескриптор дочернего окна соответственно. Однако существуют функции, которые расширяют возможности вышеуказанных, и позволяют получить дескрипторы всех окон верхнего уровня, а также дескрипторы дочерних окон для указанного top-level окна.

Функция API `EnumWindows` предназначена для перечисления окон верхнего уровня:

```
BOOL EnumWindows(  
    WNDENUMPROC lpEnumFunc, // указатель на функцию  
                           // обратного вызова  
    LPARAM lParam // аргумент, передаваемый в  
                  // функцию обратного вызова  
);
```

Функция `EnumWindows` работает совместно с CALLBACK-функцией, указанной в первом параметре, и передаёт ей дескриптор каждого перечисленного окна верхнего уровня. `EnumWindows` продолжает свою работу до тех пор, пока не будут перечислены все окна верхнего уровня или пока CALLBACK-функция не вернёт нуль. Таким образом, с помощью функции `EnumWindows` можно определить,

какие приложения, обладающие окном, выполняются в данное время.

Прототип функции обратного вызова имеет следующий вид:

```
BOOL CALLBACK EnumWindowsProc(
    HWND hwnd, // дескриптор очередного перечисленного
               // окна верхнего уровня
    LPARAM lParam // аргумент, переданный
               // в CALLBACK – функцию
);
```

Следует отметить, что для продолжения перечисления окон, CALLBACK-функция должна возвращать **TRUE**, в противном случае перечисление прекратится.

В качестве примера, демонстрирующего работу функции API **EnumWindows**, рассмотрим следующий код, в котором по нажатию клавиши <CTRL> начинается перечисление окон верхнего уровня (исходный код приложения прикреплен к pdf-файлу данного урока [SOURCE/EnumerateTopLevelWindows](#)).

```
#include <windows.h>
#include <tchar.h>
LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);
TCHAR szClassWindow[] = TEXT("Каркасное приложение");
int WINAPI _tWinMain(HINSTANCE hInst,
                    HINSTANCE hPrevInst,
                    LPTSTR lpszCmdLine,
                    int nCmdShow)
{
    HWND hWnd;
    MSG msg;
```

```

WNDCLASSEX wcl;
wcl.cbSize = sizeof(wcl);
wcl.style = CS_HREDRAW | CS_VREDRAW;
wcl.lpfnWndProc = WindowProc;
wcl.cbClsExtra = 0;
wcl.cbWndExtra = 0;
wcl.hInstance = hInst;
wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION);
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);
wcl.hbrBackground = (HBRUSH)
    GetStockObject(WHITE_BRUSH);
wcl.lpszMenuName = NULL;
wcl.lpszClassName = szClassWindow;
wcl.hIconSm = NULL;
if (!RegisterClassEx(&wcl))
    return 0;
hWnd = CreateWindowEx(0, szClassWindow,
    TEXT("Перечисление окон верхнего уровня"),
    WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
    CW_USEDEFAULT, CW_USEDEFAULT,
    CW_USEDEFAULT, NULL, NULL, hInst, NULL);
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
while(GetMessage(&Msg, NULL, 0, 0))
{
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}
return Msg.wParam;
}

BOOL CALLBACK EnumWindowsProc(HWND hWnd, LPARAM lParam)
{
    HWND hWindow = (HWND) lParam; // дескриптор окна
    // нашего приложения
    TCHAR caption[MAX_PATH] = {0},
        classname[100] = {0},
        text[500] = {0};

```

```

// получаем текст заголовка текущего окна
// верхнего уровня
GetWindowText(hWnd, caption, 100);
// получаем имя класса текущего окна
// верхнего уровня
GetClassName(hWnd, classname, 100);
if(lstrlen(caption))
{
    lstrcat(text, TEXT("Заголовок окна: "));
    lstrcat(text, caption);
    lstrcat(text, TEXT("\n"));
    lstrcat(text, TEXT("Класс окна: "));
    lstrcat(text, classname);
    MessageBox(hWindow, text,
        TEXT("Перечисление окон верхнего уровня"),
        MB_OK | MB_ICONINFORMATION);
}
return TRUE; // продолжаем перечисление окон
              // верхнего уровня
}

LRESULT CALLBACK WindowProc (HWND hWnd,
                              UINT message,
                              WPARAM wParam,
                              LPARAM lParam)
{
    switch (message)
    {
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        case WM_KEYDOWN:
            if (wParam == VK_CONTROL)
                // начинаем перечисление окон верхнего уровня
                EnumWindows(EnumWindowsProc, (LPARAM) hWnd);
            break;
    }
}

```

```

        default:
            return DefWindowProc(hWnd, message,
                                wParam, lParam);
    }
    return 0;
}

```

Функция API **EnumChildWindows** предназначена для перечисления дочерних окон указанного top-level окна:

```

BOOL EnumChildWindows(
    HWND hWndParent, // дескриптор родительского окна
    WNDENUMPROC lpEnumFunc, // указатель на функцию
                           // обратного вызова
    LPARAM lParam // аргумент, передаваемый
                 // в функцию обратного вызова
);

```

Функция **EnumChildWindows** работает совместно с CALLBACK-функцией, указанной во втором параметре, и передаёт ей дескриптор каждого перечисленного дочернего окна. **EnumChildWindows** продолжает свою работу до тех пор, пока не будут перечислены все дочерние окна или пока CALLBACK-функция не вернёт нуль. Таким образом, функция **EnumChildWindows** может применяться для последовательной обработки дочерних окон, если заранее неизвестно их количество.

Прототип функции обратного вызова имеет вид:

```

BOOL CALLBACK EnumChildProc(
    HWND hwnd, //дескриптор очередного
              //перечисленного окна

```

```
LPARAM lParam // аргумент, переданный
                // в CALLBACK – функцию
);
```

Следует отметить, что для продолжения перечисления окон, CALLBACK–функция должна возвращать **TRUE**, в противном случае перечисление прекратится.

В качестве примера, демонстрирующего работу функции API **EnumChildWindows**, рассмотрим следующий код, в котором по нажатию клавиши <CTRL> начинается перечисление дочерних окон главного окна приложения «Калькулятор» (исходный код приложения прикреплен к pdf-файлу данного урока **SOURCE/ EnumerateChildWindows**).

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

LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);
TCHAR szClassWindow[] = TEXT("Каркасное приложение");
int WINAPI _tWinMain(HINSTANCE hInst,
                    HINSTANCE hPrevInst,
                    LPTSTR lpszCmdLine,
                    int nCmdShow)
{
    HWND hWnd;
    MSG Msg;
    WNDCLASSEX wcl;
    wcl.cbSize = sizeof(wcl);
    wcl.style = CS_HREDRAW | CS_VREDRAW;
    wcl.lpfnWndProc = WindowProc;
    wcl.cbClsExtra = 0;
    wcl.cbWndExtra = 0;
```

```

wcl.hInstance = hInst;
wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION);
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);
wcl.hbrBackground = (HBRUSH)
    GetStockObject(WHITE_BRUSH);
wcl.lpszMenuName = NULL;
wcl.lpszClassName = szClassWindow;
wcl.hIconSm = NULL;
if (!RegisterClassEx(&wcl))
    return 0;
hWnd = CreateWindowEx(0,
    szClassWindow,
    TEXT("Перечисление дочерних окон"),
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    CW_USEDEFAULT, NULL, NULL,
    hInst, NULL);
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
MessageBox(hWnd,
    TEXT("Откройте, пожалуйста,
    \"Калькулятор\", и нажмите <CTRL>"),
    TEXT("Перечисление дочерних окон"),
    MB_OK | MB_ICONINFORMATION);
while(GetMessage(&Msg, NULL, 0, 0))
{
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}
return Msg.wParam;
}
BOOL CALLBACK EnumChildProc(HWND hWnd, LPARAM lParam)
{
    HWND hWindow = (HWND) lParam; // дескриптор окна
                                   // нашего приложения

```

```

TCHAR caption[MAX_PATH] = {0},
                                classname[100] = {0},
                                text[500] = {0};

//получаем текст заголовка текущего дочернего окна
GetWindowText(hWnd, caption, 100);
//получаем имя класса текущего дочернего окна
GetClassName(hWnd, classname, 100);
if(lstrlen(caption))
{
    lstrcat(text, TEXT("Заголовок окна: "));
    lstrcat(text, caption);
    lstrcat(text, TEXT("\n"));
}
lstrcat(text, TEXT("Класс окна: "));
lstrcat(text, classname);
MessageBox(hWindow,
            text,
            TEXT("Перечисление дочерних окон"),
            MB_OK | MB_ICONINFORMATION);
return TRUE; // продолжаем перечисление
              // дочерних окон
}

LRESULT CALLBACK WindowProc (HWND hWnd, UINT message,
                              WPARAM wParam,
                              LPARAM lParam)
{
    switch (message)
    {
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        case WM_KEYDOWN:
            if(wParam == VK_CONTROL)
            {
                //получим дескриптор "Калькулятора"
                HWND h = FindWindow(TEXT("SciCalc"),
                                    TEXT("Калькулятор"));
            }
    }
}

```

```
if (!h)
    MessageBox (hWnd,
                TEXT ("Необходимо открыть
                    \\"Калькулятор\\""),
                TEXT ("Ошибка!!!"),
                MB_OK | MB_ICONSTOP);
else
    //начинаем перечисление дочерних
    //окон "Калькулятора"
    EnumChildWindows (h, EnumChildProc,
                      (LPARAM) hWnd);
}
break;
default:
    return DefWindowProc (hWnd, message,
                          wParam, lParam);
}
return 0;
}
```


6. Ресурсы приложения

Ресурсы являются составной частью Windows — приложений. В них определяются такие объекты, как пиктограммы (иконки), курсоры, растровые образы, таблицы строк, меню, диалоговые окна и многие другие. Для некоторых видов ресурсов система содержит предопределенные объекты. Например, стандартная иконка (`IDI_APPLICATION`) или стандартный курсор (`IDC_ARROW`).

Все нестандартные ресурсы должны быть определены в файле описания ресурсов (resource script), который является ASCII-файлом с расширением .rc. Подобный файл можно подготовить в обычном текстовом редакторе. Однако такая технология использовалась в прошлом, поскольку Microsoft Visual Studio содержит удобные редакторы ресурсов, максимально упрощающие и автоматизирующие этот процесс.

Необходимо отметить, что изначально в новом проекте ресурсы отсутствуют. Для добавления ресурса необходимо активизировать вкладку **Resource View**, в которой с помощью контекстного меню вызвать диалог добавления ресурса **Add -> Resource...** (рис. 1).

В появившемся диалоговом окне необходимо выбрать один из стандартных типов ресурса, либо с помощью кнопки **Custom...** добавить нестандартный тип ресурса. При нажатии кнопки **New** вызывается редактор ресурса, в котором можно создать новый ресурс указанного типа (рис. 2). При нажатии кнопки **Import ...** вызывается диалоговое окно для выбора существующего ресурса (рис. 3).

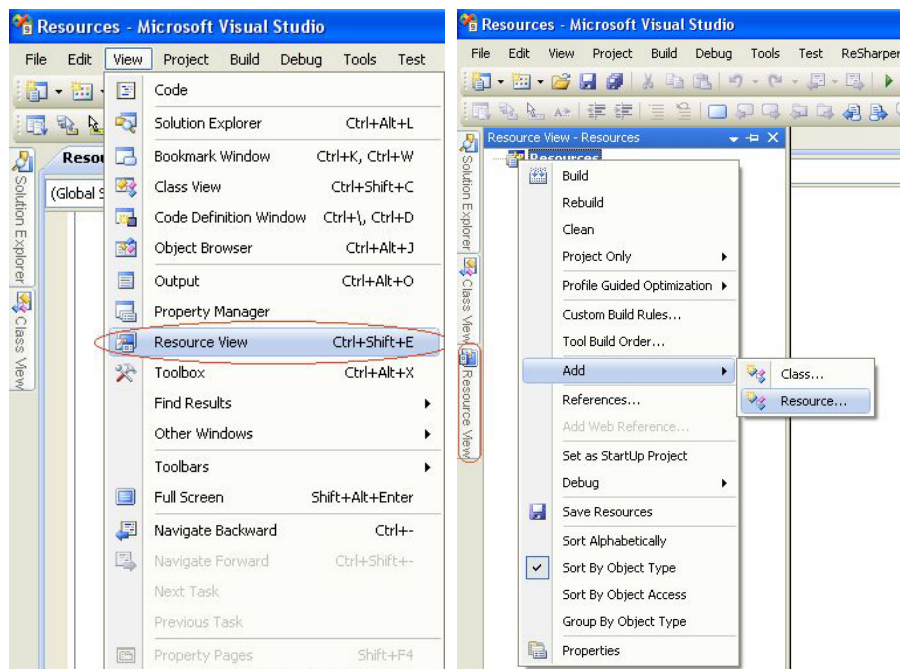


Рисунок 1

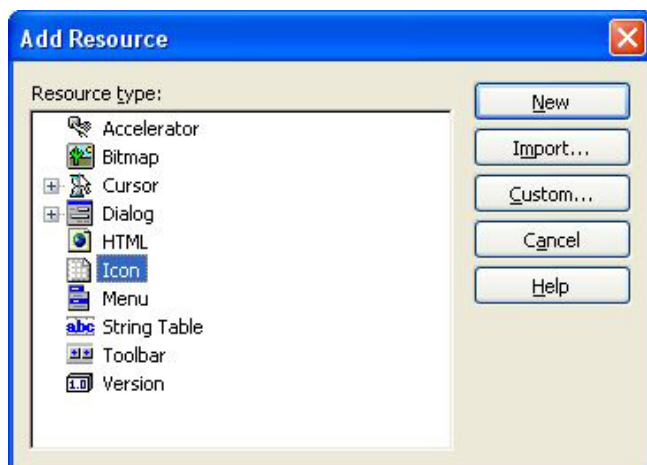


Рисунок 2

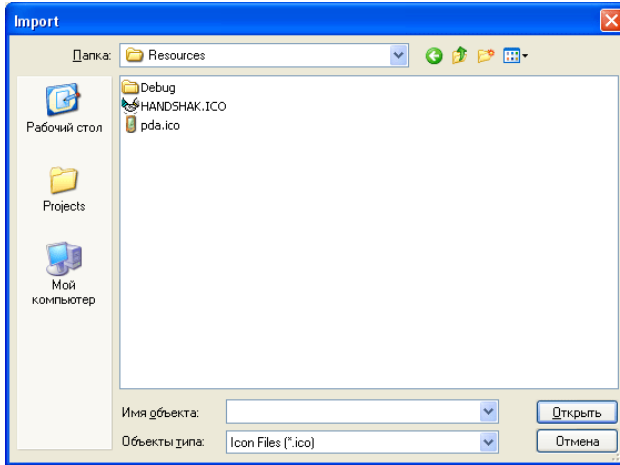


Рисунок 3

После добавления ресурса при компиляции проекта файл описания ресурсов транслируется компилятором ресурсов. В результате образуется бинарный файл с расширением **.res**. Затем компоновщик включает полученный файл в выполняемый файл программы вместе с кодом и данными программы из файлов с расширениями **.obj** и **.Lib**.

6.1. Пиктограмма

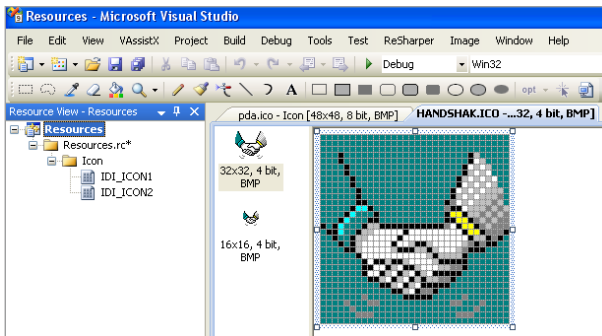


Рисунок 4

Пиктограммы (иконки) — это небольшие растровые изображения, применяемые Windows для визуального представления приложений, файлов и папок. Чаще всего для приложения создают иконки следующих типовых размеров: 16×16 пикселей для малых иконок и 32×32 пикселя для стандартных иконок. Однако существует возможность создания иконки иного размера. Для этого, вызвав контекстное меню в редакторе иконки, необходимо выбрать пункт **New Image Type...**, и в появившемся диалоговом окне выбрать нужный тип иконки (рис. 5).

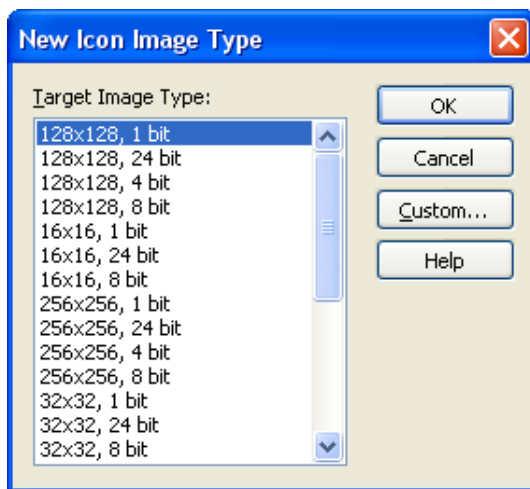


Рисунок 5

Созданная иконка сохраняется в файл с расширением **.ico**, а описание иконки добавляется в файл описания ресурсов с расширением **.rc**. При этом иконке назначается идентификатор (например, **IDI_ICON1**), который впоследствии можно изменить. Рекомендуется присваивать ресурсам идентификаторы, отражающие их семантику.

Наряду с файлом описания ресурсов, редактор ресурсов создает еще и заголовочный файл `resource.h`, содержащий определения используемых именованных констант.

Для использования в программе иконки, находящейся в ресурсах приложения, иконку следует загрузить с помощью рассмотренной ранее функции API `LoadIcon`:

```
HICON LoadIcon (
    HINSTANCE hInst, // дескриптор экземпляра
                      // приложения, содержащего иконку
    LPCSTR lpzName // строка, содержащая имя иконки
);
```

Дескриптор экземпляра приложения приходит в первом параметре функции `WinMain`. Альтернативным способом получения дескриптора экземпляра приложения является вызов функции API `GetModuleHandle`:

```
HMODULE GetModuleHandle(
    LPCTSTR lpModuleName /* имя DLL модуля */
);
```

При нулевом значении параметра функции она вернёт дескриптор экземпляра приложения.

Во втором параметре функции `LoadIcon` требуется строка с именем иконки. Поскольку имя иконки представляет собой целочисленный идентификатор (например, `IDI_ICON1`), то его можно преобразовать в строку с помощью макроса `MAKEINTRESOURCE` (make an integer into resource string):

```
#define MAKEINTRESOURCE(i) (LPTSTR) ((DWORD) ((WORD) (i)))
```

В качестве примера получения дескриптора иконки привести следующий фрагмент кода:

```
HINSTANCE hInstance = GetModuleHandle(0);
HICON hIcon = LoadIcon(hInstance,
                        MAKEINTRESOURCE(IDI_ICON1));
```

6.2. Курсор

Курсор — это изображение размером 32×32 пикселя, которое отмечает положение указателя мыши. Курсор во многом похож на иконку. Главное отличие заключается в наличии активной точки (**hotspot**). Активной точкой называется пиксель, который принадлежит изображению курсора и отмечает его точное положение на экране в любой момент времени. В стандартном курсоре, имеющем вид стрелки, активная точка расположена в левом верхнем углу курсора.

Чтобы назначить активную точку, нужно выбрать инструмент **Set Hot Spot Tool**, а затем щелкнуть мышью на том пикселе изображения, который должен стать активной точкой.

Созданный курсор сохраняется в файл с расширением **.cur**, а описание курсора добавляется в файл описания ресурсов. При этом курсору назначается идентификатор (например, **IDC_CURSOR1**), который впоследствии можно изменить на идентификатор, отражающий семантику ресурса (рис. 6).

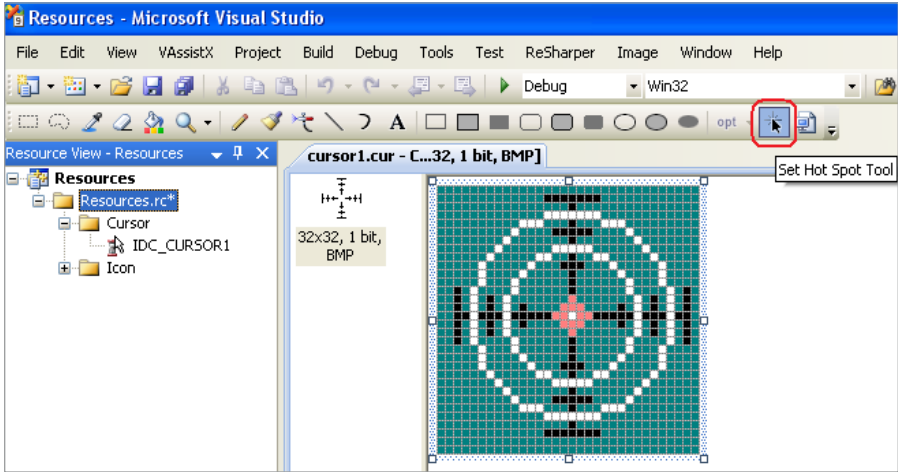


Рисунок 6

Для использования в программе курсора, находящегося в ресурсах приложения, курсор следует загрузить с помощью рассмотренной ранее функции API `LoadCursor`:

```
HCURSOR LoadCursor (
    HINSTANCE hInst, // дескриптор экземпляра
                        // приложения, содержащего курсор
    LPCSTR lpszName // строка, содержащая имя
                    // курсора
);
```

Во втором параметре функции `LoadCursor` требуется строка с именем курсора. Поскольку имя курсора представляет собой целочисленный идентификатор (например, `IDC_CURSOR1`), то его можно преобразовать в строку с помощью рассмотренного выше макроса `MAKEINTRESOURCE`.

Пример получения дескриптора курсора:

```
HINSTANCE hInstance = GetModuleHandle(0);
HCURSOR hCursor = LoadCursor(hInstance,
                               MAKEINTRESOURCE(IDC_CURSOR1));
```

Как отмечалось ранее, иконка и курсор приложения указываются на этапе определения класса окна при инициализации структуры **WNDCLASSEX**. Однако существует возможность модифицировать оконный класс, в частности, определить для приложения другую иконку или курсор. Для этой цели служит функция API **SetClassLong**:

```
DWORD SetClassLong(
    HWND hWnd,          // дескриптор окна с типом класса,
                        // который нужно модифицировать
    int nIndex,          // данный параметр указывает,
                        // что необходимо изменить
    LONG dwNewLong       // новое значение для замены
);
```

Пример модификации оконного класса:

```
HICON hIcon = LoadIcon(hInstance,
                        MAKEINTRESOURCE(IDI_ICON1));
SetClassLong(hWnd, GCL_HICON, LONG(hIcon));
HCURSOR hCursor1 = LoadCursor(hInstance,
                               MAKEINTRESOURCE(IDC_CURSOR1));
SetClassLong(hWnd, GCL_HCURSOR, LONG(hCursor1));
```

Для динамического изменения формы курсора в зависимости от его местонахождения применяется функция API **SetCursor**:


```
HCURSOR SetCursor(
    HCURSOR hCursor //дескриптор курсора
);
```

В качестве практического примера использования иконок и курсоров, определённых в ресурсах приложения, рассмотрим следующий код (исходный код приложения прикреплен к pdf-файлу данного урока [SOURCE/Resources](#)):

```
#include <windows.h>
#include "resource.h"
#include <tchar.h>
LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);
TCHAR szClassWindow[] = TEXT("Каркасное приложение");
HICON hIcon;
HCURSOR hCursor1, hCursor2;
int WINAPI _tWinMain(HINSTANCE hInst,
                    HINSTANCE hPrevInst,
                    LPTSTR lpszCmdLine,
                    int nCmdShow)
{
    HWND hWnd;
    MSG Msg;
    WNDCLASSEX wcl;
    wcl.cbSize = sizeof(wcl);
    wcl.style = CS_HREDRAW | CS_VREDRAW;
    wcl.lpfnWndProc = WindowProc;
    wcl.cbClsExtra = 0;
    wcl.cbWndExtra = 0;
    wcl.hInstance = hInst;
    // иконка загружается из ресурсов приложения
    wcl.hIcon = LoadIcon(hInst,
                        MAKEINTRESOURCE(IDI_ICON1));
```

```

// курсор загружается из ресурсов приложения
wcl.hCursor = LoadCursor(hInst,
                        MAKEINTRESOURCE(IDC_CURSOR1));
wcl.hbrBackground = (HBRUSH)
                    GetStockObject(WHITE_BRUSH);
wcl.lpszMenuName = NULL;
wcl.lpszClassName = szClassWindow;
wcl.hIconSm = NULL;

if (!RegisterClassEx(&wcl))
    return 0;

hWnd = CreateWindowEx(0, szClassWindow,
                    TEXT("Ресурсы"),
                    WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT,
                    CW_USEDEFAULT,
                    CW_USEDEFAULT,
                    CW_USEDEFAULT, NULL,
                    NULL, hInst, NULL);

ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
while(GetMessage(&Msg, NULL, 0, 0))
{
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}
return Msg.wParam;
}

LRESULT CALLBACK WindowProc (HWND hWnd,
                            UINT message,
                            WPARAM wParam,
                            LPARAM lParam)
{
    switch (message)
    {

```

```

case WM_DESTROY:
    PostQuitMessage(0);
    break;

case WM_CREATE:
{
    // получаем дескриптор приложения
    HINSTANCE hInstance =
        GetModuleHandle(0);
    // загружаем иконку из ресурсов
    // приложения
    hIcon = LoadIcon(hInstance,
        MAKEINTRESOURCE(IDI_ICON2));
    // загружаем курсоры из ресурсов
    // приложения
    hCursor1 = LoadCursor(hInstance,
        MAKEINTRESOURCE(IDC_CURSOR1));
    hCursor2 = LoadCursor(hInstance,
        MAKEINTRESOURCE(IDC_CURSOR2));
}
    break;

case WM_KEYDOWN:
    if(wParam == VK_RETURN)
        // устанавливаем иконку
        hIcon = (HICON) SetClassLong(hWnd,
            GCL_HICON, LONG(hIcon));
    break;

case WM_MOUSEMOVE:
{
    // устанавливаем тот или иной курсор
    // в зависимости от местонахождения
    // указателя мыши
    RECT rect;
    GetClientRect(hWnd, &rect);
    int x = LOWORD(lParam);
    int y = HIWORD(lParam);

```

```
        if(x >= rect.right / 4 && x <=
            rect.right * 3 / 4 && y >=
            rect.bottom / 4 && y <=
            rect.bottom * 3 / 4)
            SetCursor(hCursor1);
        else
            SetCursor(hCursor2);
    }
    break;
default:
    return DefWindowProc(hWnd, message,
                        wParam, lParam);
}
return 0;
}
```

7. Обработка ошибок

Важным критерием оценки любого приложения является надёжность его работы, поэтому достаточно актуальным является вопрос обработки ошибок.

При использовании в коде программы той или иной функции API необходимо понимать, как в этих функциях устроена обработка ошибок. Обычно, при вызове функции Windows, она проверяет переданные ей параметры, а затем пытается выполнить свою работу. Если передан недопустимый параметр или если данную операцию нельзя выполнить по какой-то другой причине, функция возвращает значение, свидетельствующее об ошибке. Например, некоторые функции имеют логический тип **BOOL** возвращаемого значения. В этом случае ложное возвращаемое значение обозначает ошибку. Другим примером является набор функций, которые возвращают дескриптор некоторого объекта **HANDLE**. Если вызов одной из таких функций заканчивается неудачно, то обычно возвращается **NULL**. При возникновении ошибки желательно разобраться, почему вызов данной функции оказался неудачен. Следует отметить, что за каждой ошибкой закреплён свой код — 32-битное число, которое можно получить, вызвав функцию API **GetLastError**. Данную функцию нужно вызывать сразу же после неудачного вызова функции Windows, иначе код ошибки может быть потерян. Если **GetLastError** возвращает нулевое значение, это означает, что предшествующий вызов функции Windows завершился успешно.

В некоторых случаях было бы удобно получить описание ошибки на основе кода ошибки. Для этого цели предусмотрена функция API `FormatMessage`:

```
DWORD FormatMessage(
    DWORD dwFlags,      // набор битовых флагов, которые
                        // определяют различные аспекты процесса
                        // форматирования, а также способ интерпретации
                        // 2-го параметра lpSource
    LPCVOID lpSource,   // указатель на строку, содержащую
                        // сообщение об ошибке

    DWORD dwMessageId,  // код ошибки
    DWORD dwLanguageId, // идентификатор языка, на
                        // котором выводится описание
                        // ошибки

    LPTSTR lpBuffer,    // выходной буфер, который
                        // выделяется системой, если
                        // в 1-м параметре указан флаг
                        // FORMAT_MESSAGE_ALLOCATE_BUFFER
    DWORD nSize,         // число символов, записываемых
                        // в выходной буфер, либо минимальный размер
                        // выделяемого буфера, если в 1-м параметре
                        // указан флаг
                        // FORMAT_MESSAGE_ALLOCATE_BUFFER
    va_list* Arguments  // список аргументов
                        // форматирования
);
```

Следующий фрагмент кода является примером использования вышеописанных функций:

```
DWORD dwError = GetLastError(); // получим код последней
                                // ошибки

LPVOID lpMsgBuf = NULL;
TCHAR szBuf[300];
```

```

// Функция FormatMessage форматирует строку сообщения
BOOL fOK = FormatMessage(
    FORMAT_MESSAGE_FROM_SYSTEM
    /* флаг сообщает функции, что нужна строка,
       соответствующая коду ошибки, определённого
       в системе
    */
    | FORMAT_MESSAGE_ALLOCATE_BUFFER /* необходимо
       выделить соответствующий блок памяти
       для хранения текста с описанием ошибки */ ,
    NULL /* текст с описанием ошибки будет находиться
       в буфере, выделенном системой. Адрес задается
       в 5-м параметре */ ,
    dwError /* код ошибки */ ,
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT) /*
       идентификатор языка, на котором выводится
       описание ошибки */ ,
    (LPTSTR) &lpMsgBuf /* указатель на буфер, в
       который запишется текст с описанием ошибки */ ,
    0, // память выделяет система
    NULL // список аргументов форматирования
);

if(lpMsgBuf != NULL)
{
    wsprintf(szBuf, TEXT("Ошибка %d: %s"), dwError,
        lpMsgBuf);
    MessageBox(hDialog, szBuf,
        TEXT("Сообщение об ошибке"),
        MB_OK | MB_ICONSTOP);
    LocalFree(lpMsgBuf); // освободим память,
                        // выделенную системой
}

```

Особенно полезно отслеживать код последней ошибки в процессе отладки приложения. Отладчик в Microsoft

Visual Studio позволяет настраивать окно **Watch** так, чтобы оно всегда показывало код и описание последней ошибки после выполнения текущей команды. Для этого необходимо в окне **Watch** ввести «@err,hr».

Необходимо отметить, что **Microsoft Visual Studio** предоставляет утилиту **Error Lookup**, которая позволяет получить описание ошибки по ее коду.

Домашнее задание

Написать приложение, в котором ведётся подсчёт количества «кликов» левой, правой и средней кнопки мыши. Обновляемую статистику необходимо выводить в заголовок окна.

Предположим, что существует прямоугольник, границы которого на 10 пикселей отстоят от границ клиентской области окна. Необходимо при нажатии левой кнопки мыши выводить в заголовок окна сообщение о том, где произошел щелчок мышью: внутри прямоугольника, снаружи или на границе прямоугольника. При нажатии правой кнопки мыши необходимо выводить в заголовок окна информацию о размере клиентской области окна (ширина и высота клиентской области окна).

Написать приложение, позволяющее при нажатии левой кнопки мыши изменить текст в заголовке окна стандартного приложения «Калькулятор», а при нажатии правой кнопки мыши сместить вправо кнопку «пуск», изменив на ней надпись.

Написать приложение, обладающее следующей функциональностью:

- при нажатии кнопки **<Enter>** окно приложения позиционируется в левый верхний угол экрана с размерами (300×300);
- с помощью клавиш управления курсором осуществляется перемещение окна.

Написать приложение, обладающее следующей функциональностью:

- после нажатия клавиши <Enter> через каждую секунду (или иной промежуток времени) «прячется» одна из кнопок «Калькулятора», выбранная случайным образом;
- после нажатия клавиши <Esc> данный процесс останавливается.

Написать приложение, обладающее следующей функциональностью:

- при нажатии клавиши <Enter> главное окно позиционируется в левый верхний угол экрана с размерами (300×300) и начинает перемещаться по периметру экрана с определённой скоростью;
- при нажатии клавиши <Esc> перемещение окна прекращается.

Написать приложение, обладающее следующей функциональностью:

- при последовательном нажатии клавиши <Enter> дочерние окна главного окна приложения «Калькулятор» минимизируются;
- при последовательном нажатии клавиши <Esc> дочерние окна восстанавливаются в обратном порядке, т.е. дочернее окно, которое минимизировалось последним, первым будет восстановлено.



Урок №2

Обработка событий мыши и таймера

© Компьютерная Академия «Шаг», www.itstep.org

Все права на охраняемые авторским правом фото-, аудио- и видеопроизведения, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объёме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии со ст. 1274 ч. 4 ГК РФ и ст. 21 и 23 Закона Украины «Про авторське право і суміжні права». Объём и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования.

Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника.

Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством Украины.