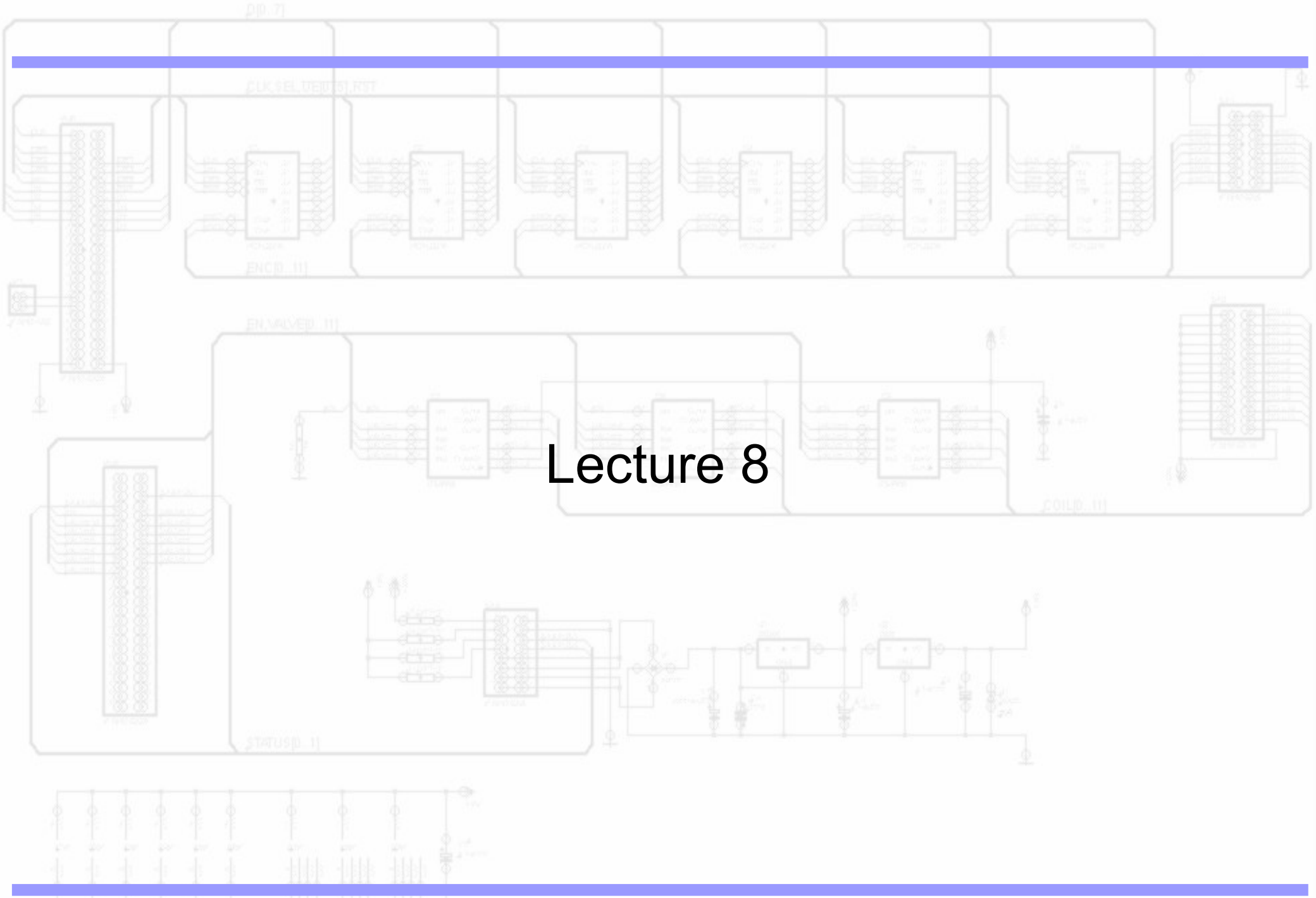


Lecture 8

Programming with Windows API

Windows API
Windows Application Programming Interface (API) – is a set of Windows OS service routines that enable applications to exploit the power of Windows operating systems.

The functional categories of Windows API are as follows:
- **administration and management**: routines allowing to service and configure Windows OS,
- **diagnostics**: a set of routines for troubleshooting and performance monitoring,
- **graphics and multimedia**: allow to enrich applications with 2D and 3D graphics, also covers the system multimedia services (libraries for handling audio, image, video files),
- **networking**: include all networking services,
- **security**: cryptography, authorization and authentication services,
- **system services**: enable applications to take advantage of system services referring to memory management, files systems, peripheral devices, processes and threads handling.
- **Windows User Interface**: enable applications to create and manage user interfaces.

There is not only backward compatibility in Windows Vista and Windows 7 but Windows API service routines are being constantly modernized and enriched with new functionalities.

Windows API can be used with any programming language that allows to execute code that is stored in Dynamically Linked Libraries (DLL). Hence there is the possibility to write Windows applications also with use of an assembly language.

Programming with Windows API

Simple Windows application

The simplest Windows application requires only one function – the entry one.

When such application is executed no visible (on screen) effect appears.

In Windows there can exist applications that do not have the graphical user interface.

```c
#include <windows.h>

int __stdcall WinMain (HINSTANCE hThisInstance,
                       HINSTANCE hPrevInstance,
                       LPSTR lpszArgument,
                       int nCmdShow)
{
    return 0;
}
```

Simple Windows application in C

```
.386
.model flat

ExitProcess     equ _ExitProcess@4

public  _start

.code

_start:

extrn   ExitProcess         : near

        push    large 0
        call    ExitProcess

end _start
```

Simple Windows application in assembly

In the assembly code in order to finish our application we have to call system function **ExitProcess**. That function takes one parameter – an exit code which normally is equal 0 and that parameter is passed via stack.

ExitProcess routine is named in system libraries as _ExitProcess@4. Hence for the sake of convenience explicit definition of ExitProcess name is introduced.

Programming with Windows API

“Hello world!” application

Below we present two codes (written in C and in an assembly language) that display welcome message “Hello world!” with help of a dialog message window.

```c
#include <windows.h>

int __stdcall WinMain (HINSTANCE hThisInstance,
                       HINSTANCE hPrevInstance,
                       LPSTR lpszArgument,
                       int nCmdShow)
{
    MessageBox(NULL, "Hello world!", "", MB_OK);
    return 0;
}
```

“Hello world!” application in C

MessageBox – this function displays dialog message box.
Parameters:
1. handle to the owner window,
2. address of a text to be displayed within dialog,
3. address of a text to be displayed as a caption,
4. style of a dialog window: MB_OK – only OK button.

```
.386
.model flat

ExitProcess     equ _ExitProcess@4
MessageBox      equ _MessageBoxA@16

MB_OK           =   00000000h

public  _start

.data

empty   db 0
message db "Hello world!"

.code

_start:

extrn   MessageBox          : near

        push    large MB_OK
        push    large offset empty
        push    large offset message
        push    large 0
        call    MessageBox

extrn   ExitProcess         : near

        push    large 0
        call    ExitProcess

end _start
```

“Hello world!” application in assembly

__stdcall calling convention

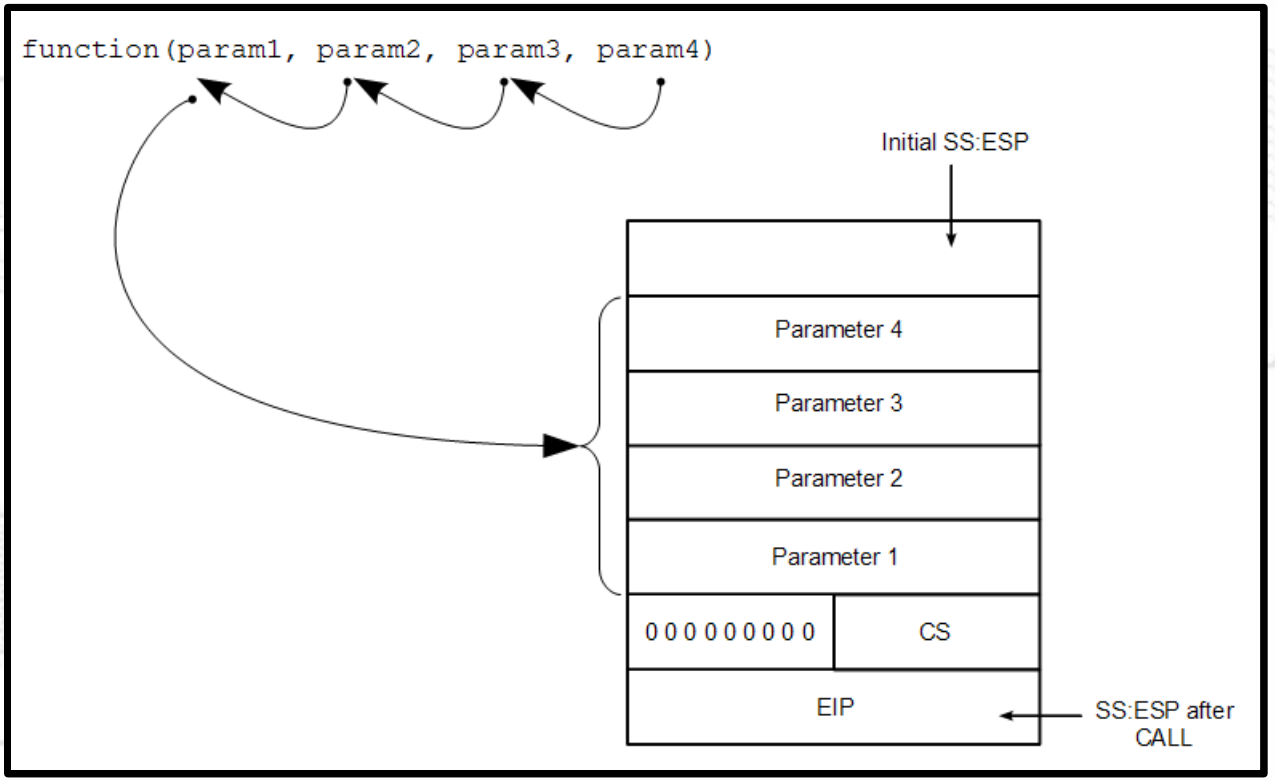
Calling conventions define the order the parameters are passed to the function and indicate which function is to clean parameters from the stack.

The **stdcall** convention is characteristic for Windows API functions. Here the parameters are passed from right to left.

The table below presents basic features of three most frequently used calling conventions.

Convention	Passing order	Cleaning
stdcall	Right to left	called
C	Right to left	calling
PASCAL	Left to right	called

Input parameters can be cleaned by a called function with RET *n* instruction, where value *n* stands for the number of bytes required by parameters.

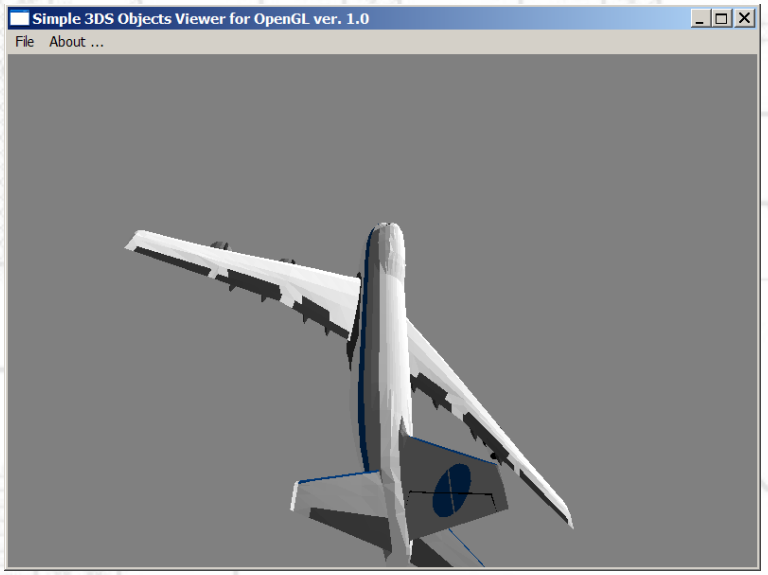


Parameters passing scheme for stdcall calling convention

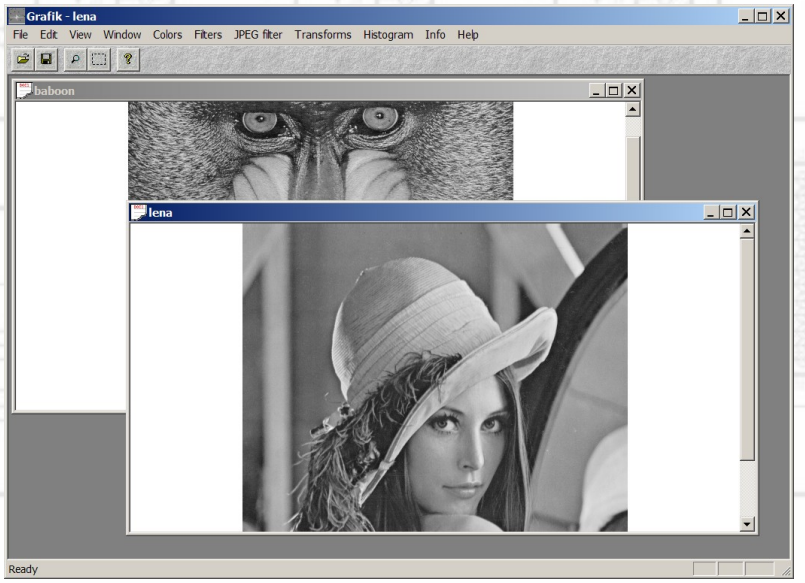
Programming with Windows API

Single Document Application (SDI)

An SDI application uses Graphical User Interface (GUI) that displays only one document window. Multiple Data Interface (MDI) application can handle several documents at one time.



An example of SDI application



An example of MDI application

Prior to execute a SDI application the following steps must be followed:
- prepare message loop,
- prepare message handling routine,
- register window class,
- create window,
- make window visible.

Programming with Windows API

Message loop

The Windows OS (i.e. keyboard, mouse) communicates with user application with the aid of messages. Messages are stored in FIFO queue of each application. In order to make our program fully functional it must read OS messages from its queue and properly handle them depending on the kind of a message. Message handling is performed in specially prepared procedure called message handling routine. To make messages flow to handling routine they must be read from the queue (**GetMessage** function) and sent to handling procedure (use **DispatchMessage** function).

```c
#include <windows.h>

int __stdcall WinMain (HINSTANCE hThisInstance,
                       HINSTANCE hPrevInstance,
                       LPSTR lpszArgument,
                       int nCmdShow)
{
    MSG msg;
    while(GetMessage(&msg, 0, 0, 0))
    {
        DispatchMessage(&msg);
    }
    return 0;
}
```

Message loop in C

```
GetMessage      equ _GetMessageA@16
DispatchMessage equ _DispatchMessageA@4

MSG     STRUC
        hwnd        dd  ?
        message     dd  ?
        wParam      dd  ?
        lParam      dd  ?
        time        dd  ?
        pt      dd  ?
MSG     ENDS

.data
        (....)
Msg             MSG <>
        (....)
.code
        (....)
extrn   GetMessage          : near
@strt0: push    large 0
        push    large 0
        push    large 0
        lea     eax, Msg
        push    eax
        call    GetMessage
        cmp     eax,0
        je      @strt1
extrn   DispatchMessage     : near
        lea     eax, Msg
        push    eax
        call    DispatchMessage
        jmp     @strt0
@strt1: nop
        (....)
end
```

Message loop in assembler

Programming with Windows API

Message handling routine

Message handling routine takes four parameters: handle to a window (hWnd), message id (uMsg), additional parameters (wParam, lParam) whose meaning depends on the kind of message. Those messages that don't need explicit handling can be passed to default handling routine **DefWindowProc**.

```
DWORD CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam,LPARAM lParam)
{
    HDC         hDC;
    PAINTSTRUCT ps;
    switch(uMsg)
    {
        case WM_CLOSE:

            DestroyWindow(hWnd);

        break;

        case WM_DESTROY:

            PostQuitMessage(0);

        break;
    }
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
```

Message handling routine in C (basic functionality)

```
PostQuitMessage     equ _PostQuitMessage@4
DefWindowProc       equ _DefWindowProcA@16
DestroyWindow       equ _DestroyWindow@4
WM_CLOSE            = 0010h
WM_DESTROY          = 0002h
.code
        (...)
WndProc proc    near
        push    ebp
        push    esp
        pop     ebp
        mov     eax,[ebp + 12]
        cmp     eax, WM_CLOSE
        je      @wndp0
        cmp     eax, WM_DESTROY
        je      @wndp1
        jmp     @wndp2
extrn   DestroyWindow       : near
@wndp0: mov     eax, [ebp + 8]
        push    eax
        call    DestroyWindow
        jmp     @wndp2
extrn   PostQuitMessage     : near
@wndp1: push    large 0
        call    PostQuitMessage
extrn   DefWindowProc       : near
@wndp2: mov eax,[ebp + 20]
        push    eax
        mov eax,[ebp + 16]
        push    eax
        mov eax,[ebp + 12]
        push    eax
        mov eax,[ebp + 8]
        push    eax
        call    DefWindowProc
        pop ebp
        ret 16
WndProc endp
        (...)
end
```

Message handling routine in assembly (basic functionality)

Programming with Windows API

Window class

The window class is a structure that describes basic features of a window, i.e. background color, icons, mouse pointer, etc., and indicates the message handling routine. Window class must be registered (**RegisterClassEx**) before the window is to be created.

```c
#include <windows.h>

HINSTANCE hInstance;

int __stdcall WinMain (HINSTANCE hThisInstance,
                       HINSTANCE hPrevInstance,
                       LPSTR lpszArgument,
                       int nCmdShow)
{
    WNDCLASSEX              wnc;
    hInstance               = hThisInstance;
    wnc.cbSize              = sizeof(WNDCLASSEX);
    wnc.cbClsExtra          = 0;
    wnc.cbWndExtra          = 0;
    wnc.hbrBackground       = (HBRUSH)COLOR_WINDOW + 1;
    wnc.hCursor             = LoadCursor(NULL, IDC_ARROW);
    wnc.hIcon               = LoadIcon(NULL, IDI_APPLICATION);
    wnc.hIconSm             = LoadIcon(NULL, IDI_APPLICATION);
    wnc.hInstance           = hInstance;
    wnc.lpfnWndProc         = (WNDPROC)WndProc;
    wnc.lpszClassName       = "MAINCLASS";
    wnc.lpszMenuName        = NULL;
    wnc.style               = 0;
    RegisterClassEx(&wnc);
    (...)
    return 0;
}
```

Registering window class in C

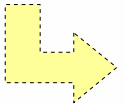
```
LoadIcon            equ _LoadIconA@8
LoadCursor          equ _LoadCursorA@8
RegisterClassEx     equ _RegisterClassExA@4
GetModuleHandle     equ _GetModuleHandleA@4
GetStockObject      equ _GetStockObject@4

CS_HREDRAW          = 0002h
CS_VREDRAW          = 0001h

WNDCLASSEX  STRUC
            cbSize      dd  ?
            style       dd  ?
            lpfnWndProc dd  ?
            cbClsExtra  dd  ?
            cbWndExtra  dd  ?
            hInstance   dd  ?
            hIcon       dd  ?
            hCursor     dd  ?
            hbrBackground   dd  ?
            lpszMenuName    dd  ?
            lpszClassName   dd  ?
            hIconSm     dd  ?
WNDCLASSEX  ENDS
```

Registering window class in assembly
(to be continued)

Window class (cont.)

```
.data
        (...)
        hInstance           dd 0
        className           db 'MYCLASS', 0
        wndClassEx          WNDCLASSEX <SIZE WNDCLASSEX,CS_HREDRAW or CS_VREDRAW,,0,0,>
        (...)
.code
        (...)
extrn   GetModuleHandle     : near
        push    large 0
        call    GetModuleHandle
        mov     hInstance,eax
        (...)
extrn   LoadIcon            : near
        push    large IDI_MAINICON
        push    hInstance
        call    LoadIcon
extrn   LoadCursor          : near
        mov     wndClassEx.hIcon,eax
        mov     wndClassEx.hIconSm,eax
        push    large IDC_CROSS
        push    large 0
        call    LoadCursor
extrn   GetStockObject      : near
        mov     wndClassEx.hCursor,eax
        push    large DKGRAY_BRUSH
        call    GetStockObject
```

Registering window class in assembly (cont.)



```
extrn   RegisterClassEx     : near
        mov     wndClassEx.hbrBackground,eax
        mov     eax,hInstance
        mov     wndClassEx.hInstance,eax
        lea     eax,className
        mov     wndClassEx.lpszClassName,eax
        xor     eax,eax
        mov     wndClassEx.lpszMenuName,eax
        lea     eax,WndProc
        mov     wndClassEx.lpfnWndProc,eax
        lea     eax,wndClassEx
        push    eax
        call    RegisterClassEx
        cmp     eax,0
        jne     @init0
        (...)
end
```

Programming with Windows API

Create and show window

When window class is registered a window object can be created with **CreateWindowEx** function. Such window is not visible yet. To make it display on the screen use **ShowWindow** function.

```
#include <windows.h>

HWND        hMainWnd;
HINSTANCE   hInstance;

int __stdcall WinMain (HINSTANCE hThisInstance,HINSTANCE hPrevInstance,LPSTR lpszArgument,int nCmdShow)
{
    (...)
    hMainWnd = CreateWindowEx(NULL, "MAINCLASS", "SDI application", WS_OVERLAPPED | WS_SYSMENU,
                              CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, NULL,
                              NULL, hInstance, NULL);
    ShowWindow(hMainWindow, SW_SHOWNORMAL);
    (...)
    return 0;
}
```

Creating window and making it visible in C

```
WS_OVERLAPPEDWINDOW = (00cfh shl 16)
CW_USEDEFAULT       = (8000h shl 16)
SW_SHOWNORMAL       = 0001h
.data
        (...)
        hMainWnd            dd 0
        appName             db 'SDI application',0
        className           db 'MYCLASS',0
        (...)
```

Creating window and making it visible in assembly (to be continued)

Programming with Windows API

Create and show window (cont.)

```
.code
        (...)
extrn   CreateWindowEx      : near
        push    large 0
        push    hMyInstance
        push    hMyMenu
        push    large 0
        push    large CW_USEDEFAULT
        push    large CW_USEDEFAULT
        push    large CW_USEDEFAULT
        push    large CW_USEDEFAULT
        push    large WS_OVERLAPPEDWINDOW
        lea     eax,appName
        push    eax
        lea     eax,className
        push    eax
        push    large 0
        call    CreateWindowEx
extrn   ShowWindow          : near
        mov     hMainWnd,eax
        push    large SW_SHOWNORMAL
        push    eax
        call    ShowWindow
        (...)
end
```

Creating window and making it visible in assembly (cont.)

Programming with Windows API

Drawing in window client area

The Windows OS sends WM_PAINT message whenever the window client area must be refreshed – change of window size, window covered by other moving window, etc. Hence the drawing code of our application must be triggered by that message.

Drawing in client area requires a handle to a window client **device context** that can be obtained with **BeginPaint** system routine. It is important to free such context with **EndPaint** function.

Drawing itself requires **pens** and **brushes** that must be created (**CreatePen**, **CreateSolidBrush**) and deleted when not required (**DeleteObject**). (Important !)

Previously created pen or brush must be selected to current device context (**SelectObject**), but earlier present pen or brush handles must be hold and restored before freeing a device context. (Important !)

```
#include <windows.h>
(...)
DWORD CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC         hDC;
    HPEN        hPen, hOldPen;
    HBRUSH      hBrush, hOldBrush;
    PAINTSTRUCT ps;
    switch(uMsg)
    {
        (...)
        case WM_PAINT:
            hDC         = BeginPaint(hWnd, &ps);
            hPen        = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
            hBrush      = CreateSolidBrush(RGB(0, 255, 0));
            hOldPen     = (HPEN)SelectObject(hDC, hPen);
            hOldBrush   = (HBRUSH)SelectObject(hDC, hBrush);
            Ellipse(hDC, 0, 0, 300, 200);
            SelectObject(hDC, hOldPen);
            SelectObject(hDC, hOldBrush);
            DeleteObject(hPen);
            DeleteObject(hBrush);
            EndPaint(hWnd, &ps);
        break;
        (...)
    }
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
(...)
```