Windows Native Programming

L. Csurgai 2016-18.

1. Introduction

- First IBM PC: 1981 with DOS operating system.

- The Microsoft Windows 1.0 released in November 1985. It runs on Intel 8086 processor in real-mode to access 1 MB of memory. It worked with tiled windows.

- Version 2.0 was released in 1987. This version worked with overlapped windows.

- The version 3.0 was introduced on May 1990. It supports the protected mode on the 80286/386/486 processors.

- Version 3.1 was released in April 1992. New features: TrueType fonts, multimedia, OLE. It runs only in protected mode with at least 1 MB of memory.

- Microsoft NT introduced in July 1993. It supports 32 bit programming, and it's portable to several RISC based workstations. (Last version: NT 4.0)

- Windows 95 was introduced in August 1995. It lacks some of the features of NT such as high security and portability. It supports 32-bit programming.

- New 32 bit Windows versions: Windows 98/Millenium, Windows 2000/XP (based on NT technology), Windows 7 32/64 bit

- Some special Windows systems: Windows CE, Windows embedded

- Windows is easy for users and difficult for programmers

- Windows programming: writing user-friendly programs

- Friendly programs are bigger and more difficult to write

- The help for programmers: API (Application Programming Interface

- It is a part of any modern graphical user interface. There is Win16 API for Win3.1 and Win32 API for Win95 and NT. Win32s: running 32 bit applications on Win3.1.

- Learning Windows programming: at least 6 month

- Why Windows ? - because we don't have a choice; the users won't like not Windows based applications.

With Windows 8 (Win8/8.1/10) released a new architecture:

- Windows Runtime (or WinRT)

- Component Object Model (COM) based API

- COM is enhanced
- easy interfacing with multiple languages
- unmanaged, native API
- API definitions are stored in ".winmd" files (ECMA 335 metadata format)
- C++/CX (Component Extensions) language (extension for C++ compilers)

- WinRT applications run within a sandbox (secure environment)
- WinRT applications are packaged in the .appx file format
-
- Metro-style: "Windows Store app" (even if local). New design and user interface.
- Windows 10 "desktop app": traditional program
-
- Universal Windows Platform (UWP) applications:
  - extension to the Windows Runtime platform
  - app from the Windows Store; it works exactly the same way on every platform (phone, tablet, computer, Xbox One)
  - coding for multidevice Windows is complex and involves compromises

2. Windows Programming Methods

- Plain C with API (Application Programming Interface)

- Use of C++ class libraries; MFC (Microsoft Foundation Classes) or Borland's OWL (Object Windows Library). They are hiding API under easier user interface.

- Visual Basic; Delphi; Java; Asymetrix's ToolBook. The best solution depends on the application is writing.

- IDE (Interactive Development Environment); helps the user in the developing process.
With the different Visual C++ versions (5.0/6.0) is possible to create different types of application (console/Win32/MFC).

3. Fundamentals

Windows provides significant advantages to both users and programmers over the conventional MS-DOS environment.

3.1 The Graphical User Interface (GUI)

- It is the “visual interface” or “graphical windowing environment”.

- The concepts date from the mid -1970s and first introduced by the Apple Macintosh in 1984.

- The GUI works on bitmapped video display (WYSIWYG).

- Icons, buttons, scrollbars and other on-screen object. Dragging, pushing, direct manipulating of objects.

- Direct user interaction with mouse and keyboard on the screen.

3.2 The Consistent User Interface

- All Windows-based programs have the same fundamental look and feel.

- The program occupies a window on the screen

- title bar with a menu
- scroll bars for large lists
- dialog boxes
- common “Open File” dialog box invoked from the same menu option
- both keyboard and mouse interface

The programmers are using the routines built in Windows for constructing menus and dialog boxes - so all menus have the same keyboard and mouse interface.

3.3 Multitasking

- Several Windows program can be displayed and running at the same time.

- The user can resize the windows, transfer data from one program to the other.

- Earlier versions of Windows used non pre-emptive multitasking. (The system didn’t use the system timer to allocate processing time for tasks, one task stops the other)

- Under Win95 multitasking is pre-emptive and programs can split to multiple threads.

3.4 Memory Management

- Multitasking cannot implement without memory management. After terminating a program, the memory became fragmented, and the system must consolidate free space.

- The system must be able to move blocks of code and data in memory.

- A program can contain more code, than can fit into memory (swapping).

- The user can run several instances of a program, and all the instances share the same code in memory.

- Link the programs with the routine at run time: DLLs.

- Windows 3: use up to 16 MB extended memory.
- Windows 95: 32 bit operating system with flat memory space (4 GB).

3.5 The Device-Independent Graphics Interface

- Windows provides full graphics on both the screen and printer.

- Windows doesn't directly access the hardware, it can be done by the device drivers. Windows includes a graphics programming language (GDI, Graphics Device interface).

- GDI makes easy display graphics and text.

- Windows virtualizes the hardware: a program runs with any type of video board or printer, which has a device driver.

3.6 Function Calls

- Windows supports over thousand function calls that application can use.

- Each Windows function has a descriptive name, such as *CreateWindow*.

- The main header file for function declaration is windows.h. It includes many other header files.

- Windows functions are located outside of the user program, in DLLs. Most of them are located in the WINDOWS\SYSTEM directory.

3.7 Message-Driven Architecture

- Windows can send messages to a user program. (e.g. resize a window). It means: Windows calls a function in the user program. This function is known as *window procedure*.

- Every window that a program creates has an associated window procedure. When it receives a message does some processing based on the message and returns control to Windows.

- Many windows based on the same window class (e.g. buttons, scrollbars). They use the same window procedure, which processes all the messages. The window class identifies the window procedure, which processes the messages.

- At the start of a program, Windows creates a *message queue* for the program, which stores messages to all windows of this program.

- The program retrieves the messages from the queue in the *message loop* of the WinMain function and dispatches them to the appropriate window procedure. However, it is possible to send messages directly to the window procedure.

4. The First Windows Program

- The MS-DOS based code:

```
#include    <stdio.h>

      main()
            {
            printf("Hello, world\n");
            }
```

Unfortunately this program doesn't work under Windows. This program directs the characters to the display device used as teletype.
(However, you can write this program as a Win32 console application.)

4.1 The "Hello, Windows!" Program

- The Windows solution:

```
/*------------------------------------------------------------
   HELLOWIN.C -- Displays "Hello, Windows!" in client area
------------------------------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "HelloWin" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName,         // window class name
                    "The Hello Program",     // window caption
                    WS_OVERLAPPEDWINDOW,     // window style
                    CW_USEDEFAULT,           // initial x position
                    CW_USEDEFAULT,           // initial y position
```

```
                    CW_USEDEFAULT,           // initial x size
                    CW_USEDEFAULT,           // initial y size
                    NULL,                    // parent window handle
                    NULL,                    // window menu handle
                    hInstance,               // program instance handle
                    NULL) ;                  // creation parameters

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

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

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
lParam)
     {
     HDC         hdc ;
     PAINTSTRUCT ps ;
     RECT        rect ;

     switch (iMsg)
          {
          case WM_CREATE :
               PlaySound ("hellowin.wav", NULL, SND_FILENAME | SND_ASYNC) ;
               return 0 ;

          case WM_PAINT :
                 hdc = BeginPaint (hwnd, &ps) ;

               GetClientRect (hwnd, &rect) ;

               DrawText (hdc, "Hello, Windows!", -1, &rect,
                              DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;

                 EndPaint (hwnd, &ps) ;
               return 0 ;

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }

     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

- The program creates a normal application window.
- It displays the “Hello, Windows!” in the center of client area.
- After starting it plays a sound file.
- The window is movable, resizable, can be maximize and minimize it.

- Compiling the program:
    - from command line with use of *makefile* and the *nmake* utility program
    - using an IDE environment

4.2 The Main Parts of the Code

- It contains only two functions: *WinMain* and *WndProc*

- Every Windows program has a *WinMain* function; it is equivalent with the standard C *main* function. This is the entry point of the program.

- Every window has a WndProc window procedure. It responds to the inputs (keyboard or mouse) and displays graphics on the screen by processing messages.

- There is no direct call of the WndProc in the program: it is called only from Windows.

- There are a lot of Windows function calls in the code (*LoadIcon, CreateWindow,* ...).
These are documented in the compiler's online reference.
- Uppercase identifiers (WM_CREATE, IDC_ARROW, CS_VREDRAW, ...) are simply constants. The prefix indicates a general category to which the constant belongs. (e.g. WM = Window message)

4.3 New Data Types

- Some convenient abbreviations: UINT = unsigned int
PSTR = pointer to string; char*

- Some less obvious abbreviations: WPARAM, LPARAM = parameters for WndProc
LRESULT = LONG type return parameter
WINAPI = the Windows function type
CALLBACK = WndProc function type

- Structures: MSG = message structure
RECT = rectangle structure

- Handles: HWND = handle to window
HDC = handle to device context
HICON = handle to icon

The handles are simply a number, referring to the object.

4.4 The Program Entry Point

The entry point of a Windows program is always like this:

```
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
```

- It returns an integer to Windows when terminates.

- hInstance = instance handle; it uniquely identifies the running program
- hPrevInstance = was used in the previous Windows versions, under Win95 this is always NULL
- szCmdLine = a pointer to a 0 terminated string that contains any command-line parameters passed to the program.
- iCmdShow = a number is indicating how the window is initially displayed (e.g. SW_SHOWNORMAL)

4.5 Registering the Window Class

- A window is always created based on a window class.
- More than one window can be created based on a single class.
- The base class determinate the characteristic of the window based on that class.

- Before creating a window, it must register with the RegisterClassEx function.
- The WNDCLASSEX structure is the single parameter of RegisterClassEx function.
- The fields of the WNDCLASSEX structure:

```
Size of the structure:
wndclass.cbSize         = sizeof (wndclass) ;
Indicate to redraw the window after horizontal or vertical resize:
wndclass.style          = CS_HREDRAW | CS_VREDRAW ;
Address of the window procedure:
wndclass.lpfnWndProc    = WndProc ;
Extra space reservation in the structure for own purpose (byte count):
wndclass.cbClsExtra     = 0 ;
wndclass.cbWndExtra     = 0 ;
Instance handle:
wndclass.hInstance      = hInstance ;
Set an icon for the window (predefined one):
wndclass.hIcon          = LoadIcon (NULL, IDI_APPLICATION) ;
Set a cursor for the window (predefined one):
wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
Specify the background color of client area:
wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
Window class application menu name:
wndclass.lpszMenuName  = NULL ;
Name of the window class:
wndclass.lpszClassName = szAppName ;
Set a small icon for the window (predefined one):
wndclass.hIconSm        = LoadIcon (NULL, IDI_APPLICATION) ;
```

4.6 Creating the Window

Create the window and get a handle:

```
hwnd = CreateWindow (szAppName,              // window class name
                     "The Hello Program",    // window caption
                     WS_OVERLAPPEDWINDOW,    // window style
                     CW_USEDEFAULT,          // initial x position
                     CW_USEDEFAULT,          // initial y position
```

```
                    CW_USEDEFAULT,           // initial x size
                    CW_USEDEFAULT,           // initial y size
                    NULL,                    // parent window handle
                    NULL,                    // window menu handle
                    hInstance,               // program instance handle
                    NULL) ;                  // creation parameters
```

4.6 Displaying the Window

Two calls are needed:

Put the window on the screen:

```
ShowWindow (hwnd, iCmdShow) ;
```

The two parameters: window handle and initial display mode.

Paint the client area:

```
UpdateWindow (hwnd) ;
```

(This function sends a WM_PAINT message.)

4.7 The Message Loop

- After UpdateWindow the window is visible on the screen.
- All Windows program has a message queue.

- The message loop of the program retrieves messages from the queue:

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

- msg is a MSG type structure.
- We get successively all messages for all windows created by the program. The fields of the msg structure are containing the next information:

- handle of window to which the message is directed
- message identifier (e.g. WM_LBUTTONDOWN)
- two 32 bit optional message parameter
- the time of the message was placed in the queue (time-stamp)
- the mouse coordinates at the time the message was placed in the queue

- In case of WM_QUIT the GetMessage returns 0 and the program terminates.

- The TranslateMessage function passes the msg structure back to Windows for some keyboard translations.

- The DispatchMessage passes the msg structure back to Windows, and Windows calls the appropriate window procedure for processing (*WndProc* in our case). After processing the message the message loop continues with the next GetMessage call.

4.8 The Window Procedure

- The real action occurs in the window procedure.

- The window procedure can have any name; in our case it's WndProc.

- A Windows program can contain more than one window procedure. A window procedure always associated with a particular window class. It is registered with the RegisterClassEx and created with CreateWindow.

4.9 Processing the Messages

- The window procedure receives the message identified by a number.
- When the window procedure processes a message, it should return 0; otherwise it must be passed to DefWindowProc function.

```
switch (iMsg)
          {
          case WM_CREATE :
          .....
          return 0 ;

          case WM_PAINT :
          .....
          return 0 ;

          case WM_DESTROY :
          .....
          return 0 ;
          }

return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
```

- WM_CREATE: it used often to perform one-time window initialization. The example program plays a sound at the initialization.

- WM_PAINT: it informs the program that the client area must be repainted (in case of resizing, maximizing ... the window).

- In the example program the program writes text at the center of the screen.

```
case WM_PAINT :
           hdc = BeginPaint (hwnd, &ps) ;
```

```
            GetClientRect (hwnd, &rect) ;

            DrawText (hdc, "Hello, Windows!", -1, &rect,
                           DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;

              EndPaint (hwnd, &ps) ;
            return 0 ;
```

- BeginPaint erases the background of the client area and returns a handle to a device context. A device context refers to a physical device and its device driver. The device context needs to display text and graphics in the client area.

- EndPaint releases the device context.
- When the program doesn't handle the WM_PAINT message, it must be passed to the DefWindowProc, which calls the BeginPaint and the EndPaint functions.

- GetClientRect gets the size of the client area.

- DrawText displays a text centered in the client area.

- WM_DESTROY: the message indicates that the user pressed the close button on the window. In the example the PostQuitMessage function inserts a WM_QUIT message in the programs message queue. When GetMessage retrieves this message, the program terminates.

5. Painting with Text

- A program can write on the client area of the window.
- This area can be resize, move or overlap by another programs. In this case it needs repainting.

5.1 The WM_PAINT message

- WM_PAINT informs the window procedure that the client area must be repainted.
- It must be done in case of:
    - moving or uncovering the window
    - resize the window
    - scrolling the window
    - remove overlaid message boxes
    - when pull-down menu is released

- Windows always saves the overwritten display area when:
    - the mouse cursor moved across the screen
    - an icon is dragged across the client area

- Almost not the entire client area needs to be repainted - only the "invalid area". Windows calculates the place of "invalid region" in the client area before sending the WM_PAINT message.

- A window procedure can invalidate a rectangle by calling the *InvalidateRect* function (it places a WM_PAINT message in the message queue).

- After *BeginPaint* the entire client area is validated; by calling *ValidateRect* is possible to validate a part of the area. This action removes the WM_PAINT message from the queue.

- Calling the *UpdateWindow* function, it is possible to immediately call the window procedure with WM_PAINT message without waiting the message flow through the queue. It is usual after *CreateWindow* and before the message loop.

5.2 The GDI

- To paint the client area we use the GDI (Graphics Device Interface) functions of Windows.

- A handle to device context is associated with a particular display device, and it is necessary to use GDI. (a window on display, printer or plotter)
- Before use a device must first obtain a handle, and after finished painting must release the handle.

- To get the device context: *BeginPaint*

This operation erases the client area with the brush defined in WNDCLASSEX.

- To release device context: *EndPaint*

- The window procedure must call *BeginPaint* and *EndPaint* when processing WM_PAINT message. Otherwise it must be passed to *DefWindowProc*!

- It is possible to update the client area without calling generating WM_PAINT message:

```
hdc = GetDC(hwnd);            // obtain device context
[use GDI functions]
ReleaseDC(hwnd, hdc);         // release device context
```

This method is used for prompt response to keyboard or mouse movements.

5.3 The TextOut function

- Writing text to the screen:

```
TextOut (hdc, x, y, psString, iLength);
```

- hdc: handle to device context, returned by *BeginPaint* or *GetDC*
- x, y: the starting point of the character string within the client area (upper left corner of the first character). The 0,0 point is the upper left corner of the client area.
- psString: pointer to character string (no special characters allowed: TAB, LF, BS,...)
- iLength: length of the string without any non-printable characters

- The attributes of device context control the characteristic of the displayed text (color, background, font, ...)

- The GDI coordinates are logical coordinates. The "mapping mode" tells how to translate logical coordinates to screen coordinates. The default mapping mode is MM_TEXT, where logical units are the same as physical units.

- The default font is SYSTEM_FONT. It is variable-pitch font: different characters have different widths. This is a raster font and guarantees that at least 25*80 characters can fit on the display.

5.4 Text Metrics

- It is possible to obtain character dimensions with the *GetTextMetrics* function:

```
TEXTMETRIC tm;

hdc = GetDC (hwnd);
GetTextMetrics (hdc, &tm);
ReleaseDC (hwnd, hdc);
```

- The TEXTMETRIC structure provides information about the current selected font in the device context. (average width, maximal width, height, ...)

- The dimensions of the system font is dependent of the resolution of display! It can obtain with the GetTextMetrics function.

- Print formatted text: use *sprintf* to buffer and then *TextOut* with this buffer.

5.5 Scroll Bars

- There are horizontal and vertical scroll bars.
- Include a scroll bar in the application: include the WS_VSCROLL (vertical scroll bar) or WS_HSCROLL (horizontal scroll bar), or both, in the window style in the *CreateWindow* statement.

```
hwnd = CreateWindow (szAppName,                      // window class name
                     "The Hello Program",            // window caption
                     WS_OVERLAPPEDWINDOW | WS_VSCROLL,// window style
                     CW_USEDEFAULT,                  // initial x position
                     CW_USEDEFAULT,                  // initial y position
                     CW_USEDEFAULT,                  // initial x size
                     CW_USEDEFAULT,                  // initial y size
                     NULL,                           // parent window hndl.
                     NULL,                           // window menu handle
                     hInstance,                      // prog. inst. handl.
                     NULL) ;                         // creation parameters
```

- Windows handles all mouse logic for scroll bars, but keyboard activities are not processed automatically.

- The default range of scroll bars is 0-100, it can be change with the *SetScrollRange* function.

- The position of the thumb can set with the *SetScrollPos* function.

- Scroll bar messages:
    - WM_VSCROLL and WM_HSCROLL when the scroll bar clicked or the thumb dragged. One message is generated when the mouse pressed and another one when released.
    - The low and high words of *wParam* corresponds to pre-defined identifiers: SB_LINEUP, SB_PAGEDOWN,...

```
case WM_VSCROLL:
      switch (LOWORD (wParam)) {
            case SB_LINEUP:
            ......
            break; }
```

6. Graphics

- The Graphical User Interface (GDI) is responsible for displaying graphics in Windows. This is a subsystem of Windows, also the system is using it for drawing.

- Never write directly to the video display!

6.1 The Structure of GDI

- GDI32.DLL and GDI.EXE contains the graphical functions; they call the routines in the driver (.DRV) files.

- GDI provides device independent graphics.
- Windows9x provides 16 bit integer coordinates. Under NT can be used 32 bit coordinates.

- Raster devices: they represent images as a pattern of dots (display, printer).
- Vector devices: draw images using lines (plotter).

- Function types:
    - Get / release device context
    - Obtain information about device context
    - Drawing functions
    - Set/get attributes of device context
    - Functions that work with GDI objects (e.g. CreatePen).

- Graphics types:
    - Lines and curves
    - Filled areas
    - Bitmaps
    - Text

- Other GDI stuff:
    - Mapping modes (working in inches or millimeters,...) and transforms (skewing or rotations, available only with NT).
    - Metafiles (a collection of GDI commands).
    - Regions (a complex shape).
    - Paths (a collection of lines and curves).
    - Clipping (a particular section of client area, defined by a region or path).
    - Palettes.

6.2 The Device Context

- Before drawing the user must first obtain a handle to device context. This gets permission from Windows to use the device.

- The device context contains many attributes.

- Getting a handle to the DC:

```
hdc = BeginPaint (hwnd, &ps);
```

The ps PAINTSTRUCT returns the invalid region. Only in this region can draw. The *BeginPaint* validates this area.

This DC applies to the region of the window defined in the *ps*, whose handle is *hwnd*. The area validated automatically.

or

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

This DC applies to the whole client area of the window whose handle is *hwnd*. The area not validated automatically.

- To obtain a handle to DC applies to the entire window, not only the client area:

```
hdc = GetWindowDC (hwnd);
```

- A more general function:

```
hdc = CreateDC (pszDriver, pszDevice, pszOutput, pData);
```

e.g. pszDevice = “EPSON FX-80”

- For example, obtain a DC for the entire display:

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

- To get information about DC but not to do any drawing:

```
hdcInfo = CreateIC ("DISPLAY", NULL, NULL, NULL);
...
DeleteDC (hdcInfo);
```

- Obtain a memory device context (for working with bitmaps):

```
hdcMem = CreateCompatibleDC (hdc);
...
DeleteDC (hdcMem);
```

- Obtain a metafile DC:

```
hdcMeta = CreateMetaFile (pszFilename);
...
// any GDI calls become part of the metafile
...
```

```
hmf = CloseMetaFile (hdcMeta);      // get handle to metafile
```

- Get information about the device (pixels, physical dimensions, colors,...):

```
iValue = GetDeviceCaps (hdc, iIndex);
```

iIndex is one of the 28 identifiers defined in Windows header file (e.g. HORZRES, VERTRES, ...)
It is important to get graphical information about screen, plotters or printers.

- Get color information:

```
for example:
      iValue = GetDeviceCaps (hdc, PLANES);          //  number  of  color
planes
      iValue = GetDeviceCaps (hdc, BITSPIXEL); // color bits per pixel
```

Windows uses an unsigned long 32 bit integer to represent colors.

bit 0-7:	RED
bit 8-15:	GREEN
bit 16-23:	BLUE
bit 24-31:	not used / alpha blending

6.3 Drawing Lines

- Set/get pixel values: *SetPixel*, *GetPixel*
- The *LineTo (hdc, xEnd, yEnd)* draws a line.
- With the *MoveToEx* function can be set the starting point.

- Obtain the current position: *GetCurrentPositionEx*

- Connect points with lines: *Polyline (hdc, pt, number_of_points);* where pt is a POINT structure with number_of_points elements. *Polyline* doesn't change the current position.
PolylineTo sets the current position to the last position of the polygon.

- Draw a rectange: *Rectangle (hdc, xLeft, yTop, xRight, yBottom)*. This function draws within the bounding box. The rectangle will be filled inside with the current brush color.

- Draw an ellipse: *Ellipse (hdc, xLeft, yTop, xRight, yBottom)*. It draws an ellipse in the bounding box.

- Draw a rectangle with rounded corners:
RoundRect (hdc, xLeft, yTop, xRight, yBottom, xCornerEllipse, yCornerellipse).
The *XCornerEllipse* and *yCornerellipse* are the x and y size of the corner ellipses.

- Draw an arc:
Arc (hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd).

- Draw a chord:
Chord (hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd).

- Draw a pie:
Pie (hdc, xLeft, yTop, xRight, yBottom, xStart, yStart, xEnd, yEnd).

- Draw a Bezier line:
PolyBezier (hdc, pt, iCount) or
PolyBezierTo (hdc, pt, iCount) where pt is an array of POINT structures. In pt the first four points are the begin point, the first control point, second control point and end point of the Bezier curve. Each subsequent Bezier requires three more points because the next begin point is the end point of the last curve. iCount is 1+3*number of curves.

6.4 Pens

- Windows uses the pen currently selected in the device context to draw a line.

- The pen has color, width and style (solid, dotted, dashed or combination of them). The default is the BLACK_PEN.

- There are three stock pens: BLACK_PEN, WHITE_PEN and NULL_PEN.

- Drawing with stock pens:

```
HPEN hPen;

hPen = GetStockObject (WHITE_PEN);  // get handle to pen
SelectObject (hdc, hPen);           // select pen
...                                 // draw with it
```

- Creating own pens:

```
hPen = CreatePen (iPenStyle, iWith, rgbColor);
```

iPenStyle: PS_SOLID, PS_DASH, ...
iWith: 0 = one pixel wide
rgbColor: unsigned long integer, RGB(rvalue, gvalue, bvalue)

Create a pen indirect:

```
LOGPEN logpen;    // structure to store the pen information
hPen = CreatePenIndirect (&logpen);
```

- To draw with a pen is possible only after the *SelectObject* command.

- After using the pens it must be delete them with the *DeleteObject(hPen)* command.

- In OPAQUE background mode (this is the default) Windows fills the line gaps (e.g. in dotted lines) with the background color (the default background color is white).
In TRANSPARENT mode Windows ignores the background color.

- Drawing modes: Windows uses binary raster operations (ROP2) to combine original screen pixel and the drawing pixel.
The default mode is R2_COPYPEN, where Windows copies the pixel of the pen to the destination.
To set the drawing mode:

```
SetROP2 (hdc, iDrawMode);
```

6.5 Drawing Filled Areas

- The seven Windows area drawing function:
Rectangle
Ellipse
RoundRect
Chord
Pie
Polygon – a closed *Polyline*
PolyPoligon - multiple polygons

- The border of figures drawed with the same method, as at the lines.
- The figure is filled with the current brush.

- There are six stock brushes, or can create customized brushes:

```
HBRUSH hBrush;                          // handle for brush

hBrush = GetStockObject (GRAY_BRUSH);   // get handle to stock brush
SelectObject (hdc, hBrush);             // select brush to use
...                                                 // draw with it
DeleteObject (hBrush);                  // delete brush
```

Create a logical brush:

```
hBrush = CreateSolidBrush (rgbColor);    // pure color brush

hBrush = CreateHatchBrush (iHatchStyle, rgbColor);   // hatched color
brush

// e.g. HS_HORIZONTAL

hBrush = CreatePatternBrush (hBitmap);   // brush based on bitmap
```

Create a brush indirectly:

```
hBrush = CreateBrushIndirect (&logbrush);
```
The fields in logbrush structure: Style, Color, Hatch

6.6 The Mapping Mode

- The GDI functions are working on logical coordinates. In all GDI functions Windows translates the coordinate values from logical units to device units, or pixels.

- The translation and the axis orientation are depending on mapping mode.

- The eight mapping modes:

MM_TEXT	1 logical unit = 1 pixel
MM_LOMETRIC	1 logical unit = 0.1 mm
MM_HIMETRIC	1 logical unit = 0.01 mm
MM_LOENGLISH	1 logical unit = 0.01 in
MM_HIENGLISH	1 logical unit = 0.001 in
MM_TWIPS	1 logical unit = 1/1440 in
MM_ISOTROPIC	1 logical unit = arbitrary, x=y
MM_ANISOTROPIC	1 logical unit = arbitrary, x!=y

- Select mapping modes: *SetMapMode (hdc, iMapMode);*

- The default mapping mode is MM_TEXT.

- The coordinate values must be signed short integers from -32768 to 32767.
- The mapping mode is device context attribute, can be obtained by the Get

6.7 The Viewport and the Window

- Device coordinate systems:
    - screen coordinates: the entire screen
    - whole-window coordinates: the programs entire window
    - client-area coordinates: the client area

- The viewport is often the client area, but can refer to whole-window or screen coordinates. The viewport is in terms of device coordinates (pixels).

- Windows can translate window (logical) coordinates to viewport coordinates.

- The formulas use the window and viewport origin coordinates in the translation.
- The formulas also use the window and the viewport extents. The extent means nothing itself, but the ratio of viewport and window extent is a scaling factor for converting logical units to device units.

```
xViewPort = (xWindow-xWinOrg)    * (xViewExt/xWinExt) + xViewOrg
xWindow   = (xViewPort-xViewOrg) * (xWinExt/xViewExt) + xWinOrg
```

- The window is always in logical coordinates, the viewport is in device coordinates.

- Function to convert device points to logical points: *DPtoLP (hdc, pPoints, iNumber);*
- Converting logical points to device points: *LPtoDP (hdc, pPoints, iNumber);*

6.8 Working with Mapping Modes

6.8.1 Text mapping mode

Window origin:	(0,0)	can be changed
Viewport origin:	(0,0)	can be changed
Window extent:	(1,1)	cannot be changed
Viewport extent:	(1,1)	cannot be changed

```
xViewPort = xWindow - xWinOrg + xViewOrg
```

The x coordinate grows from left to right, and the y from up to down.

- The ratio of extents is 1, so no scaling performed between logical an device coordinates.

- The viewport (device coordinates) and the window origins (logical coordinates) can be set with the *SetViewportOrgEx* and *SetWindowOrgEx* functions.
The device point 0,0 is always the upper left corner of the client area!

- It is not usual to change both viewport and window origins.

- The GetViewportOrgEx and GetWindowOrgEx functions retrieves the origins.

- It is possible to shift the display by changing the window or viewport origins, than repaint the display by sending a WM_PAINT message.

6.8.2 Metric mapping mode

MM_LOMETRIC	1 logical unit = 0.1 mm
MM_HIMETRIC	1 logical unit = 0.01 mm
MM_LOENGLISH	1 logical unit = 0.01 in
MM_HIENGLISH	1 logical unit = 0.001 in
MM_TWIPS	1 logical unit = 1/1440 in

Window origin:	0,0	can be changed
Viewport origin:	0,0	can be changed
Window extent:	depends on mapping mode	cannot be changed
Viewport extent:	depends on mapping mode	cannot be changed

The meaning of ViewExt/WinExt = number of pixels in 1 logical unit e.g. 0.1 mm.

The x coordinate grows from left to right, and the y from down to up.

- The standard VGA display pixel is 0.28/0.325 mm wide and high, so VGA device coordinates are coarser than any of the metric mapping modes.
On 300 dpi printer each pixel is 0.0033 inch.

6.8.3 Mapping mode with scalable extents

MM_ISOTROPIC	1 logical unit = arbitrary, x=y
MM_ANISOTROPIC	1 logical unit = arbitrary, x!=y

- In the isotropic mode is possible to control the physical size of the logical unit by changing the extents. It is possible to change the window and viewport extents, but Windows adjust them so, that the x and y logical units represents the same physical size.
In an-isotropic mapping mode Windows doesn't adjust the values, so the x and y dimensions can be different.

- The viewport and the window extents can be set with the *SetViewportExtEx* and *SetWindowExtEx* functions.

- With isotropic mapping mode is possible to fit any logical window into the client area. In an-isotropic mode is also possible to fit any picture in any size client window by stretching the image.

6.9 Rectangles

- A RECT structure contains the upper left and lower right coordinates of a rectangle.

- *FrameRect (hdc, &rect, hBrush)*: draw a rectangular frame with a one logical wide brush.

- *FillRect (hdc, &rect, hBrush)*: it fills the rectangle with the specified brush.

- Invert*Rect (hdc, &rect)*: inverts bitwise the pixels in the rectangle.

- Functions working with rectangles:

```
SetRect (&rect,  xLeft,  yTop,  xRight,  yBottom);
OffsetRect (&rect, x, y);                  // move
InflateRect (&rect, Δx, Δy);   // increase or decrease size
SetRectEmpty (&rect);                  // set the coordinates to 0
CopyRect (&DestRect, &SourceRect);
IntersectRect (&DestRect, &SourceRect1, &SourceRect2);
```

```
UnionRect (&DestRect, &SourceRect1, &SourceRect2);
empty = IsRectEmpty (&rect);
isIn = PtInRect (&rect, point);
```

6.10 Regions

- Region: an area on the screen that is a combination of rectangles, polygons and ellipses. It can be used for drawing or clipping. The clipping area is a part of the screen where drawing is allowed.

- A region is a GDI object - it should be deleted after use with *DeleteObject*.

create rectangular region:
hRgn = CreateRectRgn (xLeft, yTop, xRight, yBottom);
create elliptical region:
hRgn = CreateEllipticRgn (xLeft, yTop, xRight, yBottom);
create region with rounded corners:
hRgn = CreateRoundRectRgn (xLeft, yTop, xRight, yBottom);
create a polygonal region:
hRgn = CreatePolygonRgn (&point, iCount, iPolyFillMode);

- Combination of regions:
CombineRgn (hDestRgn, hSrcRgn1, hSrcRgn2, iCombine);
iCombine: AND, OR, XOR, DIFF, COPY
- After region created, it can use for drawing:

```
FillRgn (hdc, hRgn, hBrush);// fill with hBrush
FrameRgn (hdc, hRgn, hBrush, xFrame, yFrame);   // xFrame; yFrame: width and
                                                // height of frame to be paint
InvertRgn (hdc, hRgn);                          // invert colors
PaintRgn (hdc, hRgn);                           // fill with the current brush
```

- Invalidate a region (generate WM_PAINT message):
InvalidateRgn (hwnd, hRgn, bErase);

- Validate a region (make validation and delete any WM_PAINT message from queue):
ValidateRgn (hwnd, hRgn);

- The invalid region also defines a *clipping region*.
SelectClipRgn(hdc,hRgn);

6.11 Path

- Path: a collection of straight lines and curves stored internally to GDI.

- Begin path definition: *BeginPath(hdc);*

.... any line will be stored internally to GDI

- End path definition: *EndPath(hdc);*

- After path definition can call these functions:

    Draw the path with the current pen:
    StrokePath(hdc);

    Fill the path with the current brush:
    FillPath(hdc);

    Both together:
    StrokeAndFillPath(hdc);

- Convert path to region: *hRgn = PathToRegion(hdc);*

- Select a path for clipping area: *SelectClipPath(hdc, iCombine);*
    iCombine: AND, OR, XOR, DIFF, COPY; it shows how to combine the path with the current clipping region.

- For path Windows supports the extended pen creation function:

    hPen = ExtCreatePen (iStyle, iWidth, &lBrush, 0, NULL);

style examples: PS_ENDCAP_ROUND // line end style
PS_JOIN_MITER // line join style

This function can be used only with path, and it has more options than *CreatePen.*

6.12 Bitmaps

- Bitmap: representation of digital image.

    Monochrome bitmap: 1 bit/pixel
    Color bitmap: more than 1 bit/pixel (4/8/24)

- Metafile: description of picture, scalable, but slow to display.

- Bitmaps are device dependents: color/resolution
- The size of bitmaps is large; it can be displayed fast but distortion may occur on magnifying.

6.12.1 The Device Independent Bitmap (DIB)

- Prior to Win 3.0: device dependent bitmaps as GDI objects.
- The DIB contains a color table that describes how to convert pixel values to nearest RGB colors.

- The DIB is not GDI object. It can be converted to/from device-dependent bitmap. This operation is a nearest-color conversion.

- The DIB file begins with the header and the color table followed by the bitmap bits.

header: BITMAPFILEHEADER
BITMAPINFOHEADER
RGBQUAD(s)
bitmap bytes

- Displaying a DIB file:

Method 1:

- read it into an allocated memory as "packed DIB memory" format (all except of BITMAPFILEHEADER)
- display it with the *SetDIBitsToDevice* or *StretchDIBits* functions

Method 2:

- The *LoadImage* function loads the bitmap image (or an icon/cursor) from a file (or from resource) and returns its handle. It can be displayed by selecting it into a memory DC and calling the *BitBlt* or *StretchBlt* functions.

6.12.2 Converting DIBs to Bitmap Objects

- Using the *CreateDIBitmap* creates a device dependent GDI bitmap object from a DIB specification and returns a handle to bitmap object.

- With *CreateDIBitmap* is possible to create an non-initialized GDI bitmap object.

- Set/obtain bits of the bitmap: *SetDIBits* and *GetDIBits*.

6.12.3 The GDI Bitmap Object

- Working with it is easier and faster than DIBs. It can be used to work with bitmaps inside the program.

- Creating bitmaps in a program:

CreateDIBitmap(hdc, &bmih, 0, NULL, NULL, 0); // bmih=bitmap info header
CreateBitmap (cxWidth, cyHeight, iPlanes, iBitsPixel, pBits);
CreateBitmapIndirect(&bitmap); // the bitmap structure defines the bitmap

CreateCompatibleBitmap(hdc,cxWidth, cyHeight); //compat. with current device

CreateDiscardableBitmap(hdc,cxWidth, cyHeight); // obsolete version

- Get size and color organisation: *GetObject*
- Get actual bits: *GetDIBits*
- Set bitmap bits: *SetDIBits*

- After using bitmaps, it should delete with DeleteObject command!

6.12.4 The Monochrome Bitmap Format

- 1 pixel = 1 bit

- Creating a bitmap:

```
// set the parameters
static BITMAP bitmap = { type, width, height, width_in_bytes, color_planes,
color_bits_per_pixel} ;

// set the pixel bits
static BYTE byBits[] = { ..... the bits of bitmap };
bitmap.bmBits = (LPVOID) byBits;

// create
hBitmap = CreateBitmapIndirect (&bitmap);
```

6.12.5 The Color Bitmap Format

- The color bitmap is device-dependent!

- The image stored as sequential scan lines, beginning with the top line. The bits are representing the red, green, blue and intensity values sequentially.

- Nothing in the bitmap specifies how color bits are corresponding to actual display colors! Thus *CreateBitmap* or *CreateBitmapIndirect* is not recommended to create color bitmaps.

- The *CreateCompatibleBitmap* is the good method to create color bitmaps compatible with the real output device.

- When compatibility with the real graphics output device is not necessary, use DIBs. This is a slower but transportable solution.

6.12.6 The Memory Device Context

- A memory device context has a "display surface" that exists only in memory. In a memory device context are all GDI functions allowed.

- Creating a memory device context: *hdcMem = CreateCompatibleDC (hdc);*
The attributes are set to default values (a display surface which contains 1 monochrome pixel). On a memory DC allowed the use of all GDI functions.

- Select a real bitmap to memory device context: *SelectObject (hdcMem, hBitmap);*

- With *hBitmap = CreateCompatibleBitmap (hdcMem, xWidth, yHeight)* is possible to create a monochrome bitmap with xWidth, yHeight sizes.

- A bitmap is a GDI object; however, it cannot select into a normal device context, only into a memory device context. A block transfer operation is needed to display it.

- After using the memory device context, it must be deleted with *DeleteDC (hdcMem).*

6.12.7 Block Transfer Functions - Displaying Bitmaps

- Patter Block Transfer: *PatBlt (hdc, xDest, yDest, xWidth, yHeigh, dwROP);*
  The x/y parameters are in logical units
  xDest,yDest is the upper left coordinate of the rectangle
  xWidth,yHeight is the size of the rectangle
  dwROP is (one of the 256) Raster Operation performed between destination and the current brush (e.g. DSTINVERT, BLACKNESS, PATCOPY)

  The ROP code is the result of the source, destination and the operation pattern, and it is a 32 bit code and can be determined from any C Reference Guide.

- Working with source and destination devices:
  BitBlt (hdcDest, xDest, yDest, xWidth, yHeigh,
  hdcSource, xSrc, ySrc, dwROP);

  It performs a logical combination of the source device context pixels, the destination device context pixels and the destination device context brush.

  With *BitBlt* is possible to display a bitmap image on the screen. It can be used to transfer image from memory DC to the screen.

  Windows converts colors and mapping modes, when the source and the destination modes are not equal.

- Stretching bitmaps:
  StretchBlt (hdcDest, xDest, yDest, xDestWidth, yDestHeigh,
  hdcSource, xSrc, ySrc, xSrcWidth, ySrcHeight, dwROP);

  This function allows stretch or compress images. The result may have some distortion, and the operation can be slow.

  When shrinking a bitmap, the combination mode of the rows and columns can set by the *SetStretchBltMode (hdc, iMode)* function.

Valid modes are: BLACKONWHITE AND combination
WHITEONBLACK OR combination
COLORONCOLOR no combination

6.13 Metafiles

- Metafiles are vector graphics (bitmaps are raster graphics). It is a collection of graphics function calls encoded in binary form.

- The metafiles can be scaled without loss of resolution or convert them to bitmaps.

- The size of metafiles is not too big.

- Create a metafile: *CreateMetaFile* (can be a memory or disk file)
- With GDI function calls can draw in metafile.
- The *CloseMetaFile* function finishes the construction of the metafile.
- The *PlayMetaFile* draws the image stored in the metafile.
- After using metafiles it must be deleted with the *DeleteMetaFile* command.

- Open a disk metafile: *hmf = GetMetaFile (szFileName);* the file extension is WMF.

- Problem: it is difficult to get the image size from the metafile (only the analyze of file may help)

6.13.1 Enhanced Metafiles

- Windows 95 and up supports a new metafile with new functions, structures clipboard format, etc. The file extension is EMF. This file contains an extended header.

- Create an enhanced metafile:
  CreateEnhMetaFile(hdc, "fname", &rect, "description");

- Playing the metafile: *PlayEnhMetaFile(hdc, hEmf, &rect);*

6.14 Text and Fonts

6.14.1 Simple Text Output

- The most common function:

TextOut (hdc, xStart, yStart, pString, iCount);

The text alignment can be set with the *SetTextAlign* function (left, right, center, up, down, etc. alignments). The position is based on xStart.

- The tabbed text out can be used if the text contains TAB's:

TabbedTextOut (hdc, xStart, yStart, pString, iCount
iNumTabs, piTabStops, xTabOrigin);

- Extended text out:

ExtTextOut (hdc, xStart, yStart, iOptions, &rect, pString, iCount, pxDistance);
This function allows to set the intercharacter spacing.

- Higher level text drawing function:

DrawText (hdc, pString, iCount, &rect, iFormat);
It draws a text in a rectangle.

6.14.2 Device Context Attributes for Text

- Color, set with *SetTextColor*

- Background mode: SetBkMode
OPAQUE: Windows uses the background color between the character strokes.
TRANSPARENT: Windows doesn't color the area between the character strokes.

- Set background color: *SetBkColor*

- Set intercharacter spacing: *SetTextCharacterExtra.* The default value is 0 logical units.

6.14.3 Stock Fonts

- Windows provides various stock fonts to use (e.g. System Font, which is used in menus, dialog boxes, message boxes, etc.).

- Get handle to a stock font: *hFont = GetStockObject (iFont);*

- Select font into the device context: *SelectObject (hdc, hFont);*

- The *GetTextMetrics* function can be used to get information about the font parameters.

6.14.4 Font Types

- GDI fonts: stored in files on hard disk
    - raster fonts: stored as bitmap pixel pattern; non-scalable fonts, fast to display, good quality
    - stroke fonts (plotter fonts); "connect the dots" format, scalable but poor performance
    - true type fonts

- device fonts: dedicated to output devices (printers, plotters, ...)

6.14.5 TrueType Fonts

- The characters are defined as the outline of lines and curves. It can be used both for display and printers; WYSIWYG. It was introduced in the 3.1 version of Windows.

- Windows rasterizes the font before use (builds a bitmap).

- The fonts equipped with Windows: Courier, Times, Ariel, Symbol

- Courier is a fixed pitched font (every character has the same width)

7. The Keyboard

7.1 Basics

- The message path:

    - user keypress
    - store the keystrokes in the system message queue
    - transfer the keyboard message to the message queue of the program, which has the current input focus
    - the program dispatches the message

- The program doesn't need to act on every keyboard message:
    - CTRL / ALT combinations,
    - Automatic edit control handling
    - Dialog box keyboard interface provided by Windows
- Focus: the window that receives the keyboard message is the window with the input focus. (active window or child window of the active window) Windows highlights the title bar of the active window.

Messages with focus:
- WM_SETFOCUS = if Windows gets the focus
- WM_KILLFOCUS = if Windows lost the focus

7.2 Keystroke Messages

- On system key pressed: WM_SYSKEYDOWN (ALT+key)
- On system key released: WM_SYSKEYUP

- The system messages are more important for Windows than to the application. (function keys, Alt, Ctrl, etc.) The programs usually passes this messages to the *DefWindowProc*.

- On nonsystem key pressed: WM_KEYDOWN
- On non-system key released: WM_KEYUP

- This messages may use the application program, Windows ignores them.

- The keystroke messages are time-stamped, which can call the *GetMessageTime* function.

- Autorepeat: many WM_KEYDOWN and a single WM_KEYUP message will be placed in the queue.

7.2.1 The lParam Variable

- For the four keystroke messages the 32 bit lParam variable is set and it contains six fields (coded in the bits):

- Repeat count (bit 0-15): number of keystrokes
- OEM scan code (bit 16-23): the hardware generated keystroke code
- Extended key flag (bit 24): if the keystroke is one of the IBM Extended Keyboard
- Context code (bit 29): 1 if Alt pressed
- Previous key state (bit 30): 0 if the key was previously up and 1 if it was down
- Transition state (bit 31): 0 for KEYDOWN and 1 for KEYUP

7.2.2 The wParam Variable

- The wParam contains the virtual key code of the key pressed.

- The virtual key codes are defined in windows.h header file (e.g. VK_CANCEL, VK_F12). It contains 145 different codes in all.

7.2.3 The Shift States

- The *GetKeyState (VK_SHIFT)* returns negative value, if Shift key is down.

- The *GetKeyState (VK_CAPITAL)* sets the low bit of return value, if Caps Lock is on.

- This is not a real-time check for keyboard. Use this function only during processing keystroke messages! For real-time purpose the *GetAsyncKeyState* function can be used.

7.2.4 Adding Keyboard Interface to a Program

```
case WM_KEYDOWN:

      switch (wParam)
            {
            case VK_HOME:
            // set scrollbar position to home
            break;
            .
            .

            }
```

7.3 Character Messages

- Translating keystroke messages to character messages is possible, but must take into account some other information: shift state and country-dependent information. Thus let Windows translate this messages into character codes!

- The four character messages:

WM_CHAR	(deriver from WM_KEYDOWN)
WM_DEADCHAR	(deriver from WM_KEYDOWN)
WM_SYSCHAR	(deriver from WM_SYSKEYDOWN)
WM_SYSDEADCHAR	(deriver from WM_SYSKEYDOWN)

- In most cases the user must handle only the WM_CHAR messages.

- The wParam contains the ANSI code of the pressed character. It also generates the Ctrl codes from Ctrl-A to Ctrl-Z, included backspace, tab, enter, etc..

example:

```
case WM_CHAR:

      switch (wParam)
            {
            case '\n':
            // handle line feed
            break;

            case 'A':
            // program lines
            break;
            .
            .

            }
```

7.3.1 Dead - Character Messages

- Some non-U.S. keyboard defines diacritics to a letter. These keystrokes defined as dead-keys.

- Pressing the dead-key Windows sends a WM_DEADCHAR message in wParam the diacritic. Then the user presses the letter itself, and Windows passes the ASCII code of the character with diacritic to the user with WM_CHAR message.

- Thus the user program doesn't have to process the WM_DEADCHAR message.

7.4 The Caret

- On a character screen a little underline or box appears; the name of this is the *caret*. (The *cursor* expression is used for the bitmap images.)

- Caret functions:

CreateCaret (it creates caret but doesn't display it)
SetCaretPos
ShowCaret (show the created caret)
HideCaret
DestroyCaret

- There is only one caret in the system: the program must handle caret state on WM_SETFOCUS and WM_KILLFOCUS.

7.5 The Windows Character Sets

- The OEM character set: this is the IBM character set, programmed into the ROM BIOS chip supported by Windows. It is used with running DOS applications in a window.

- International character sets: code pages; Windows supports it by installing the appropriate OEM fonts. They used for DOS applications running in a window.

- ANSI character sets (ISO standard): after receiving the WM_CHAR message, the wParam is the ANSI character code. (from 0x20 to 0x7e represents the same characters, as the OEM set)

- Windows displays ANSI and OEM characters with different fonts. In default Windows uses the SYSTEM_FONT with ANSI character set.
To select OEM fonts call *SelectObject (hdc, GetStockObject(OEM_FIXED_FONT));*

- Use ANSI character set with international keyboards!

- Converting character sets: *CharToOem (lpszAnsiStr, lpszOemStr)* and *OemToChar (lpszOemStr ,lpszAnsiStr).*

8. The Mouse

8.1 Basics

- Windows 95 and up supports the one- two- and three button mouse; the two-button mouse has become the de facto standard.

- The *fMouse = GetSystemMetrics (SM_MOUSEPRESENT);* function determines the present of a mouse.
The *cButtons = GetSystemMetrics (SM_CMOUSEBUTTONS);* function determines the number of buttons on the mouse.

- There are several predefined mouse cursors: arrow, cross, hourglass. Load an arrow cursor with the *LoadCursor (NULL, IDC_ARROW)* function.

- Mouse actions: click, double click, dragging.

8.2 Client-Area Mouse Messages

- A window procedure receives mouse messages whenever the mouse passes over the window or is clicked within the window, even if the window is not active or doesn't have the input focus.

- When the mouse moved over the window, the window procedure receives the WM_MOUSEMOVE message.

- When the mouse button pressed or released, the window procedure receives the next messages:

WM_LBUTTONDOWN	WM_LBUTTONUP	WM_LBUTTONDBLCLK
WM_MBUTTONDOWN	WM_MBUTTONUP	WM_MBUTTONDBLCLK
WM_RBUTTONDOWN	WM_RBUTTONUP	WM_RBUTTONDBLCLK

MBUTTON messages are generated only in case of three button mouse, RBUTTON messages are generated only in case of three or two button mouse.

- The value of lParam contains the coordinate of the mouse, the low word is the x, the high word is the y coordinate. WParam indicates the state of the mouse buttons and the Ctrl and Shift keys.

- Windows doesn't generate WM_MOUSEMOVE messages for every pixel position; the number of messages depends on the hardware and speed.

- When the left button is pressed in the client area of a window, Windows sets this window as active window.

- If a system modal message box or dialog box is on the screen, no other program can receive mouse messages!

- The window procedure receives double-click messages only when the program initializes the window style properly:

wndclass.style = CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS;

8.3 Non-client Area Mouse Messages

- When the mouse is outside the client area, but inside the window, Windows sends a non-client area message. Non-7client area includes the title bar, the menu and window scroll bars.

- The non-client area messages usually passed to the DefWindowProc:

WM_NCLBUTTONDOWN
WM_NCLBUTTONUP
WM_NCLBUTTONDBLCLK
WM_NCMBUTTONDOWN
WM_NCMBUTTONUP
WM_NCMBUTTONDBLCLK
WM_NCRBUTTONDOWN
WM_NCRBUTTONUP
WM_NCRBUTTONDBLCLK

- The wParam indicates the non-client area where the mouse was moved or clicked (HTxxx, predefined identifiers).

HTCLIENT	client area
HTNOWHERE	not on any window
HTTRANSPARENT	a window covered by another
HTERROR	it produces a beep

- The lParam variable contains a x coordinate in the low word and the y coordinate in the high word, but these are screen coordinates. (the upper left corner of the screen is the 0,0 point)

8.4 The Hit-Test Message

- It precedes all other client and non-client area messages. (WM_NCHITTEST)

- lParam contains the x,y screen coordinates, wParam is not used.

- Usually the *DefWindowProc* processes this message, and generates from it the other mouse messages according to the mouse coordinates.

9. The Timer

9.1 Basics

- The Windows timer is a device, which periodically notifies an application when the specified interval elapsed.

- It places the WM_TIMER message in the message queue.

- Uses of the timer:

  - Multitasking with divide and execute the code into small parts which executed on timer event.
  - Display status lines repeatedly.
  - Auto-save.
  - Auto-termination.
  - Processor-independent pacing.
  - Multimedia applications.

9.2 Setup the Timer

- The *SetTimer* function sets the timer interval and starts the timer. The timing interval can range from 1 ms to ~50 days. The timer can be stopped with the *KillTimer* function.

- The Windows system handles the 54.925 ms hardware tick to generate the WM_TIMER messages.

- The WM_TIMER message provides not accurate timing. The timing interval is divided by 54.925 ms and the result is rounded. Under 55 ms timing period the system generates only one WM_TIMER message in every 55 ms.

- The timer messages are placed into the message queue - this is asynchronous messaging.

9.3 Using the Timer - Method One

The simplest method - timer messages are processed in the window procedure:

```
#define MY_TIMER1       1                       // timer identifiers
#define MY_TIMER2       2
```

```
// setup the timers
SetTimer (hwnd, MY_TIMER2, delay1, NULL);
SetTimer (hwnd, MY_TIMER2, delay2, NULL);

// handle timer messages
case WM_TIMER:
      switch (wParam)
            {
            case MY_TIMER1:
                  ...
                  break;
            case MY_TIMER2:
                  ...
                  break;
            }
      return 0;

// kill timers
KillTimer (hwnd, MY_TIMER1);
KillTimer (hwnd, MY_TIMER2);
```

9.2 Using the Timer - Method Two

Direct the timer messages to a user function = callback method:

```
#define MY_TIMER        1     // timer identifier

// the callback function; the obligatory name is TimerProc
// iMsg is always WM_TIMER
// dwTime is the system time; the time in ms elapsed since Windows was
started
void CALLBACK TimerProc (HWND hwnd, UINT iMsg, UINT iTimerID, DWORD dwTime)
      {

      // do any processing

      }

// setup the timer
SetTimer (hwnd, MY_TIMER, delay1, (TIMERPROC)TimerProc);
```

10. Child Window Controls

10.1 Basics

- Controls for child windows:
    - buttons
    - check boxes
    - edit boxes
    - list boxes
    - combo boxes
    - text strings
    - scroll bars

- Create a child window with the *CreateWindow* function
- Receive the messages from the child window by trapping the WM_COMMAND messages.
- A predefined child window class (e.g. "button", "listbox") doesn't need registering (*RegisterClassEx*); the class is already exists within Windows and there is a window procedure also for this windows.

-Create a button child window:
CreateWindow("button", "Push", WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON, x, y, width, height, hwndParent, childID, hInstance, NULL);

- The non-predefined child windows needs registering and defining window-procedure.

10.2 Messages

- The child window control (e.g. by pressing a pushbutton) sends a WM_COMMAND message to its parent window.

LOWORD (wParam)	Child window ID (the value when the window was created)
HIWORD (wParam)	Notification code (submessage code, e.g. BN_CLICKED)
lParam	Child window handle

- The parent window can send a message to its child window. (e.g. BM_SETCHECK: set the check state on of a check box)

10.3 The Button Class

- Push Buttons
    - BS_PUSHBUTTON, BS_DEFPUSHBUTTON

- 3D style shading
- Set a button to pushed state: *SendMessage(hwndButton, BM_SETSTATE, 1, 0);*

- Check Boxes (a square box with text)
  - left/right text alig
  - normal/autocheckbox (setting of the check mark)
  - 3 state check box

- Radio Buttons
  - a check box with circle

- Group Boxes
  - a rectangular outline to enclose other button controls

- Owner-Draw Buttons
  - buttons with own images
  - this button sends a WM_DRAWITEM message when needs to repaint it

- Changing the button text: *SetWindowText(hwnd, string);*

- Visible and enabled buttons:
  - When a child window is visible but not enabled, the text is displayed in gray rather than black.
  - The non-visible window can displayed by calling the *ShowWindow(hwndChild, SW_SHOWNORMAL)* function.
  - Hide a window: *ShowWindow(hwndChild, SW_HIDE);*
  - Enable a window: *EnableWindow (hwndChild, TRUE);*

- After pressing a button, the parent window loses the keyboard input focus. Trapping the WM_KILLFOCUS message may help.

10.4 The Static Class

- A rectangle or frame on the screen with optional text; they don't accept keyboard or mouse input, don't send WM_COMMAND message.

10.4 The Scrollbar Class

- There is difference between a "window scroll bar" (on the right and bottom of the window), and a child window scrollbar control.

- Scroll bar controls don't send WM_COMMAND messages, instead WM_HSCROLL and WM_VSCROLL messages. The difference between window scroll bars and scroll bar controls is in the lParam parameter: it is 0 for window scrollbars and the scroll bar handle for scroll bar controls.

- The size of a scroll bar control can be any.

- Creating a scroll bar child window: *CreateWindow("scrollbar", ...*

- If a scroll bar has the input focus, it controls the keyboard automatically:

Home key	SB_TOP message
End key	SB_BOTTOM message
Page Up key	SB_PAGEUP message
Page Down key	SB_PAGEDOWN message
Left or Up key	SB_LINEUP message
Right or Down key	SB_LINEDOWN message

10.4 The Edit Class

- A rectangular window with editable text. (text input, cursor movements, Cut and Paste)

- The number of edit rows can be more than one.

- Left, right and center text formatting.

- Horizontal and vertical scrolling (also with scrollbars).

- The edit control sends a WM_COMMAND message with some notification codes (changed, updated, scrolled, etc.).

- Messages to the edit control from the parent window:
    WM_CUT, WM_COPY, obtain current line, etc.

10.4 The Listbox Class

- A collection of text strings displayed as a scrollable columnar.

- The listbox sends a WM_COMMAND message to the parent window when an item is selected in the list.

- It can be single or multiple selection (more than one item can be selected) listbox. The selected item(s) is displayed by highlighting.

- The user can use the spacebar for selection, the cursor keys or the mouse, or move to an item begins with a letter actually pressed.

- Main listbox styles:

LBS_NOTIFY	-	send WM_COMMAND
LBS_SORT	-	sorted items
LBS_MULTIPLESEL	-	multiple selection

- Putting strings in the listbox:

SendMessage(hwndlist, LB_ADDSTRING, 0, (LPARAM)string);
SendMessage(hwndlist, LB_INSERTSTRING, index, (LPARAM)string);

- Deleting strings in the listbox:

SendMessage(hwndlist, LB_DELETESTRING, index, 0);
SendMessage(hwndlist, LB_RESETCONTENT, 0, 0);

- Manipulating entries:

count = SendMessage(hwndlist, LB_GETCOUNT, 0, 0); // get item no.
index = SendMessage(hwndlist, LB_GETCURSEL, o, 0); // get selected item no.
SendMessage(hwndlist, LB_SETCURSEL, index, 0); // highlight item

- The listbox control sends a WM_COMMAND message with some notification codes (selection changed, double click, etc.).

11. Resources

11.1 Basics

- The resources in a Windows program:
    icons
    cursors
    menus
    bitmaps
    dialog boxes
    character strings
    keyboard accelerators
    user defined resources

- The resource data are stored in the programs EXE file, but they are loaded by Windows on run-time, when the program uses the resource.

- Resources are defined in a resource script file, which is an ASCII text file with RC extension. It can reference to other files.

- The RC.EXE resource compiler program compiles the resource file to binary form and builds the .RES file.

- To include compiled resources in the EXE file can be done by linking.

11.2 Icons and Cursors

- The icons and cursors are bitmaps, and can be edited by the image editor, which is a development tool in the IDE.

- Icon and cursor sizes: small = 16*16 pixel, 16 colors
    standard = 32*32 pixel, 16 colors or monochrome

- The resource script in the .RC file:

```
    my_icon_name    ICON  iconfile.ico          (using icon name)
    123             ICON  iconfile.ico          (using ID number)
or
    my_cursor_name  CURSOR  curfile.cur         (using cursor name)
    456             CURSOR  curfile.cur         (using ID number)
```

- To load an icon in the program file:

hIcon = LoadIcon(hInstance, "my_icon_name");

or

hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(123));

- To load a cursor in the program file:

hCursor = LoadCursor(hInstance, "my_cursor_name");

or

hCursor = LoadCursor(hInstance, MAKEINTRESOURCE(456));

- Using #define in header file:

Header file: #define my_icon_name 123
Resource script: my_icon_name ICON iconfile.ico
Program source: *hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(my_icon_name));*

In the resource script file should be placed the *#include "my_header.h"* line.

11.3 Bitmaps

- Bitmaps are used to: - draw images on the screen
- create brushes

- The resource script in the .RC file:

my_bitmap_name BITMAP bmpfile.bmp

- To load a bitmap in the program file:

hBitmap = LoadBitmap(hInstance, "my_bitmap_name");

hBrush = CreatePatternBrush(hBitmap); // create brush from the bitmap

- Bitmaps are GDI objects - they should be deleted when they are no longer needed:

DeleteObject(hBitmap);

11.4 Character Strings

- They are used primarily for easing the translation of the program to other languages.

- The resource script in the .RC file:

```
STRINGTABLE
        {
        id1, "string 1"
        id2, "string 2"
        }
```

- The resource script can contain only one string-table; each string in the table can be only one line long with max. 255 characters; it cannot contain C style control characters except of \t (tab). The string can contain octal constants: e.g. \011 (tab).

- To load a string into a buffer in the program file:

LoadString (hInstance, id, buffer, maxlength);

11.5 User-Defined Resources

- This type of resource is useful to store any type of data in the program's EXE file. From the user program is possible to access this data. The data can be any type: text or binary form.

- The resource script in the .RC file in case of text data:
helptext TEXT proghelp.txt

- The name of resource (helptext) and type of the resource (TEXT) can be any user defined names, except of the names defined by the system.

- To obtain a handle to this resource in the program file:

hResource = LoadResource (hInstance, FindResource(hInstance,"TEXT", "helptext"));

hResource is HGLOBAL type.

- Access the resource data:

pHelpText = LockResource (hResource); *// load resource into memory and returns // a pointer to it*

- Free resource from memory: *FreeResource (hResource);*

12. Menus and Accelerators

12.1 Menus

A window's menu bar is displayed below the title bar of the program main window. This is the "main menu" or top-level menu". The items usually invoke a drop-down menu, which is so called "popup-menu" or "submenu". Sometimes items invoke a dialog box.
The three menu types are the *system menu*, the *main menu* and a *popup menu*.
Menu items can be checked with a small check mark, and can be enabled, disabled or grayed.
Windows sends a WM_COMMAND message when the user selects an enabled item from the menu.

Each menu defined by three characteristic. The first is what appears in the menu: text or bitmap. The second is either an ID number that Windows sends to the main program in a WM_COMMAND message, or a popup menu that Windows displays when the user selects that menu item. The third characteristic describes the attribute of the item: enabled, disabled or grayed.

12.2 Creating Menus

- In the resource script:

```
MyMenu MENU
      {
      MENUITEM "&Text", ID [,options]
      POPUP "&Text" [,options]
            {
            [menu list]
            }
      }
```

Options in the top-level menu:

GRAYED	(inactive grayed menu item)
INACTIVE	(inactive menu item)
MENUBREAK	(begins a new line)
HELP	(this and the next items are right-justified)

Options for popup menus:

CHECKED	
GRAYED	
INACTIVE	
MENUBREAK	(begins a new column of the menu)
MENUBARBREAK	(begins a new column of the menu separated with vertical bar)

12.3 Referencing the Menu in the Program

wndclass.lpszMenuName = "MyMenu";

or load from resource:

hMenu = LoadMenu (hInstance, "MyMenu");

After loading specify the menu in *CreateWindow*:

```
hwnd = CreateWindow (szAppName,         // window class name
                     "The Hello Program",     // window caption
                    WS_OVERLAPPEDWINDOW,     // window style
                    CW_USEDEFAULT,           // initial x position
                    CW_USEDEFAULT,           // initial y position
                    CW_USEDEFAULT,           // initial x size
                    CW_USEDEFAULT,           // initial y size
                    NULL,                    // parent window handle
                    hMenu,                   // window menu handle
                    hInstance,               // program instance handle
                    NULL) ;                  // creation parameters
```

Assign a new menu to a window allows dynamically change the menu:

SetMenu(hwnd, hMenu);

12.4 Menu Messages

- Most of the menu messages can be ignored and pass them to *DefWindowProc*:
  WM_INITMENU
  WM_MENUSELECT (when the user moves on menus)
  WM_INITMENUPOPUP
  WM_MENUCHAR (on selecting a not-defined Alt-char combination)

-The most important menu message is the WM_COMMAND. It indicates the selection of an enabled menu item.

LOWORD (wParam):	Menu ID
HIWORD (wParam):	0
lParam:	0

- WM_SYSCOMMAND: On selecting an enabled system menu item.

LOWORD (wParam):	Menu ID (Some predefined menu ID's: SC_MINIMIZE, SC_MOVE)
HIWORD (wParam):	0
lParam:	0

12.5 Floating Popup Menu

- Define in the resource only one popup menu that invokes the other menu options:

```
MyMenu MENU
{
     POPUP ""
     {
          POPUP "&File"
          {
               MENUITEM "&New",IDM_NEW
               MENUITEM "&Open...",IDM_OPEN
               MENUITEM "&Save",IDM_SAVE
               MENUITEM "Save &As...",IDM_SAVEAS
               MENUITEM SEPARATOR
               MENUITEM "E&xit",IDM_EXIT
          }
          POPUP "&Edit"
      }
}
```

- Load the menu during WM_CREATE and get the handle to the first submenu:

hMenu = LoadMenu (hInstance, "MyMenu");
hMenu = GetSubMenu(hMenu, 0);

- In case of right mouse button message call:

TrackPopupMenu (hMenu, 0, x, y, 0, hwnd, NULL);

12.6 Using the System Menu

- Modifying the system menu: a quick but dirty way to add a menu to short programs.

- The ID's of the new items must be under 0xF000.

- Adding items to the system menu: get handle to the system menu (*GetSystemMenu*) and use the *AppendMenu* function.
- After receiving and handling WM_SYSCOMMAND messages the other WM_SYSCOMMAND messages must pass to *DefWindowProc*.

12.7 Changing the Menu and Menu Commands

AppendMenu
DeleteMenu
InsertMenu
ModifyMenu
RemoveMenu (removes an item but doesn't delete the menu)

DrawMenuBar	(redraws the menu)
GetSubMenu	(get handle to popup menu)
GetMenuItemCount	
GetMenuItemID	
CheckMenuItem	(get the checked state of item)
GetMenuState	
DestroyMenu	(remove the whole menu from the program)

12.8 Keyboard Accelerators

- Keyboard accelerators are key combinations that generate WM_COMMAND messages.

- It is used to duplicate the common menu actions, but it is useful to perform non-menu functions.

- The Windows usually sends keyboard messages to the window procedure which has the input focus. However for keyboard accelerators Windows sends the WM_COMMAND message to the window whose handle is specified in the *TranslateAccelerator* function.

12.9 The Accelerator Table

- Defining in the RC resource script:

```
MyAccelerators ACCELERATORS
{
"char", ID          [,SHIFT][,CONTROL][,ALT]    // character
"^char", ID         [,SHIFT][,CONTROL][,ALT]    // character + CTRL
nCode, ID, ASCII    [,SHIFT][,CONTROL][,ALT]    // ASCII code
nCode, ID, VIRTKEY  [,SHIFT][,CONTROL][,ALT]    // Virtual key code
}
```

12.10 Loading the Accelerator Table

- Define a handle to the accelerator table:

HACCEL hAccel;

- Load the table:

hAccel = LoadAccelerators (hInstance, "MyAccelerators");

12.11 Translating the Keystrokes

- The standard message loop:

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

- The message loop with accelerator table:

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

The *TranslateAccelerator* function determines and translates the message to the appropriate window. If the keyboard accelerator ID corresponds to a system menu ID, than the message is WM_SYSCOMMAND, otherwise the message is WM_COMMAND.

The *TranslateAccelerator* function does not translate keyboard messages when a modal dialog box or message box has the input focus.

12.12 Receiving the Accelerator Messages

- The *TranslateAccelerator* function sends a WM_COMMAND message to the window procedure with the next parameters:

LOWORD (wParam):	Accelerator ID
HIWORD (wParam):	1
lParam:	0

- If the keyboard accelerator corresponds to a system menu item, the *TranslateAccelerator* function sends a WM_SYSCOMMAND message.

- If the active window is minimized, the *TranslateAccelerator* function sends only WM_SYSCOMMAND messages to enable menu items.

13. Dialog Boxes

13.1 Basics

- A dialog box is a popup window with various child window controls.
Windows is responsible to create the dialog box window, provide a window procedure and process the messages (dialog box manager).

- The programmer must create the dialog box, process the messages and ending the dialog box.

- The dialog box template is in the resource script file, the dialog box procedure is in the source code file, and the identifiers are usually going to the program's header file.

13.2 Modal Dialog Boxes

- Dialog boxes are either modal or modeless. The most common dialog box is the modal. The modal dialog box must be ended by the OK or Cancel button before switching to other windows. To switch to other running program is allowed, but in case of 'system modal' dialog boxes must ended before any user action.

13.3 Creating an 'About' Dialog Box

- Designing a dialog box template (resource script file):

```
AboutBox DIALOG  20, 20, 160, 80
     STYLE WS_POPUP | WS_DLGFRAME
     {
     CTEXT "About1"                      -1,   0, 12, 160,  8
     ICON  "About1"                      -1,   8,  8,   0,  0
     CTEXT "About Box Demo Program"      -1,   0, 36, 160,  8
     DEFPUSHBUTTON "OK"                   IDOK,  64, 60,  32, 14, WS_GROUP
     }
```

- The coordinate units are based on the system font size: x unit is 1/4 of the average character with, y unit is 1/8 of the character height. This allows creating video resolution independent general dimensions. The *GetDialogBaseUnits* function let’s to determine the system font sizes.

- The dialog box procedure:

```
BOOL CALLBACK AboutDlgProc (HWND hDlg, UINT iMsg, WPARAM wParam, LPARAM
lParam)
     {
     switch (iMsg)
```

```
          {
          case WM_INITDIALOG :
               return TRUE ;

          case WM_COMMAND :
               switch (LOWORD (wParam))
                    {
                        case IDOK :
                        case IDCANCEL :
                         EndDialog (hDlg, 0) ;
                              return TRUE ;
                    }
               break ;
          }
     return FALSE ;
     }
```

- Invoking the dialog box:

```
DialogBox (hInstance, "AboutBox", hwnd, AboutDlgProc) ;
```

- The messages for a modal dialog box don't go through the program's message queue, so there is no need to worry about keyboard accelerators within the dialog box.

13.4 Dialog Box Styles

- The most common style:
  STYLE WS_POPUP | WS_DLGFRAME

- Create title bar:
  STYLE WS_POPUP | WS_CAPTION

- Put text in the title bar:
  CAPTION "Caption text"
  or from the program:
  SetWindowText(hDlg, "Caption text");

- Add a system menu to the title bar:
  STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU
- The FONT statement let’s use different types of fonts.

13.5 Dialog Box Controls

- In a dialog box can be used all usual child window controls (button, text, icon, edit control, scrollbar, listbox, combobox).

- The format of the controls in the resource script:

  control-type "text", ID, x, y, w, h, [,style]

- For edit, scrollbar, listbox and combobox are no "text" field.

- There is a generalized control statement:

CONTROL "text", ID, "class", style, x, y, w, h

The next two statements are identical:

PUSHBUTTON "OK", IDOK, 10, 20, 32, 14
CONTROL "OK", IDOK, "button", WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON, 10, 20, 32, 14

- Get the window handle of dialog control (useful to send message to a control):
hwndCtrl = GetDlgItem(hDlg, ID);

- The OK and CANCEL buttons: When the user presses the OK button or the ENTER key, Windows sends to the dialog box procedure a WM_COMMAND message with wParam == IDOK. If the user presses the ESC or Ctrl-Break, wParam is IDCANCEL.

13.6 Tab Stops and Groups

- When a control has the WS_TABSTOP style, the user can move between these controls with the TAB key.

- With a group style in the resource script the first element with WS_GROUP style up to (but not included) the last element of control with WS_GROUP style can define a group of controls. Inside the group the user can move with the cursor keys between the controls.

13.7 Message Boxes

- A message box is a simple dialog box when the program needs a simple response from the user.

- The syntax:
iItem = MessageBox(hwndParent, "Text", "Caption", type);

- The message box has a caption, one or more text lines, one or more buttons, and an optional icon. The return value indicates the button click action e.g. IDOK, IDYES. The type is some of the pre-defined style identifiers: e.g. MB_OK, MB_YESNOCANCEL.

13.8 Modeless Dialog Boxes

- The modeless dialog box allows the user to switch between the dialog box and other windows or programs running on the system.

- Creating modeless dialog boxes:

```
hdlg = CreateDialog (hInstance, "MyModalDlg", hwndParent, DlgProc) ;
```

This function returns immediately.

- The usual style of a modeless dialog box:

STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU | WS_VISIBLE

- The messages for a modeless dialog box come through the program's message queue, so the message loop must be altered to pass these messages to the dialog box window procedure.

The message loop looks like this:

```
while (GetMessage (&msg, NULL, 0, 0))
          {
          if (hDlgModeless == 0 || !IsDialogMessage (hDlgModeless, &msg))
               {
               TranslateMessage (&msg) ;
               DispatchMessage  (&msg) ;
               }
          }
```

If the message is for the dialog box, the *IsDialogMessage* sends it to the dialog box.

- Ending a modeless dialog box: *DestroyWindow(hDlg)*

13.9 Common Dialog Boxes

- Windows 95 and up provides a common dialog box library with some pre-defined dialog box to support creating consistent user interface.

- The COMMDLG.H defines the necessary functions and structures, the COMDLG32.LIB is the import library, the COMDLG32.DLL is the run-time library for this functions.

- The common dialog boxes:

ColorDialog	Lets user select colors.
FileDialog	Lets user select a filename to open or to save.
FindReplaceDialog	Lets user initiate a find or replace operation in a text file.
FontDialog	Lets user specify a font.
PrintDialog	Lets user specify information for a print job.

14. User Interface Enhancements

14.1 The Common Control Library

With Windows95 the COMCTL32.DLL introduced 17 newly defined control categories:

Frame window controls:	Toolbar Tooltip Status bar
Compound dialog boxes:	Property page Property sheet
Windows explorer controls:	Tree view List view
Miscellaneous controls:	Animation Drag list Header Hot-key Image list Progress bar Rich edit Tab Trackbar Up-down

- Each common control is defined as a window class.
- Common control sends WM_NOTIFY messages with unique notification codes.

14.2 Using of the Common Controls

- Include the *commctrl.h* file.
- Initialisation: *InitCommonControls();*
- The rich edit control requires the RICHED32.DLL. Load it with the *LoadLibrary("RICHED32.DLL");* function.
- Creating the controls:
    - *CreateWindow* with special arguments
    - Control specific functions: *CreateToolbarEx*, *CreatePropertySheetPage*, etc.
- Sending messages to common controls: - with *SendMessage*
  - with macros e.g. *TreeView_InsertItem*

15. Memory Management

15.1 The Segmented Memory

- The 8086 and 8088 processors were capable of addressing 1 MB of memory (20 bit addressing). It was real-mode addressing.
- The 20 bit address was formed from two 16 bit values:

Segment: ssssssssssssssss0000
Offset: 0000oooooooooooo
Address: aaaaaaaaaaaaaaaaa

- With constant kept segment registers the processor was able to access 64 KB of memory.
- The larger programs were required multiple segments for code and data.
- Near (short) pointers: 16 bit wide using default code or data segments.
- Far (long) pointers: 32 bit wide including segment and offset addresses. The far pointers cannot directly access memory.
- Different programming models were introduced:
  - Small: one code and one data segment.
  - Medium: multiple code segments.
  - Compact: multiple data segments.
  - Large: multiple code and data segments.
  - Huge: large model with built-in address increment logic.
- MS-DOS did not have much memory management: due to the multitasking Windows programs the memory becomes fragmented, and it was necessary to solve this problem.
- If the programs use only offset addresses, the segment addresses can be changed by the operating system (memory manager). The early Windows versions were thus limited to small or medium models.
- Memory allocation under 16 bit Windows: local memory with 16 bit offset pointers, global memory with 32 bit address.
- Memory allocation returned a handle to the memory block.

15.2 Protected Mode on 80286 Processors

- The 80286 used 24 bit addressing for 16 MB of memory.
- The segment address is called as selector.

Selector: ssssssssssssssss

↓

Descriptor table

↓

Base: bbbbbbbbbbbbbbbbbbbbbbbb

Offset: 00000000oooooooooooooooo

Address: aaaaaaaaaaaaaaaaaaaaaaaa

15.3 The 32 bit Memory Addressing

- The 386, 486 and Pentium processors use 32 bit address space capable of addressing 4 GB memory.
- These processors are still capable of using the segment registers, but Windows keeps them fixed and it uses 32 bit flat addressing space.
- The 32 bit addresses used by Windows are not physical addresses. The applications are using *virtual* addresses.
- The virtual addresses are translated to physical addresses through a *page table*.
- The physical memory is divided to 4096 byte pages. Each page address begins at an address with the lower 12 bits zero.

- Every process has an own *directory page* with up to 1024 32-bit entries. The physical address of the current directory page is stored in the processors CR3 register, which is changed when the processor switches control between processes.

- The structure of the virtual address:
  - Upper 10 bits: specify one of the 1024 entry in the directory page
    - The structure of the directory entry:
      - The upper 20 bit indicates a physical address of a page table. (The bottom 12 bits are zero.) This references another directory page with up to 1024 entries.
  - Middle 10 bits: reference one of the 1024 entry of the second directory page. This entry references to a page frame, which is a physical address.
  - Bottom 12 bits: specify the physical position within the page frame.

d: directory page entry
p: page table entry
o: offset

Virtual address: dddd-dddd-ddpp-pppp-pppp-oooo-oooo-oooo

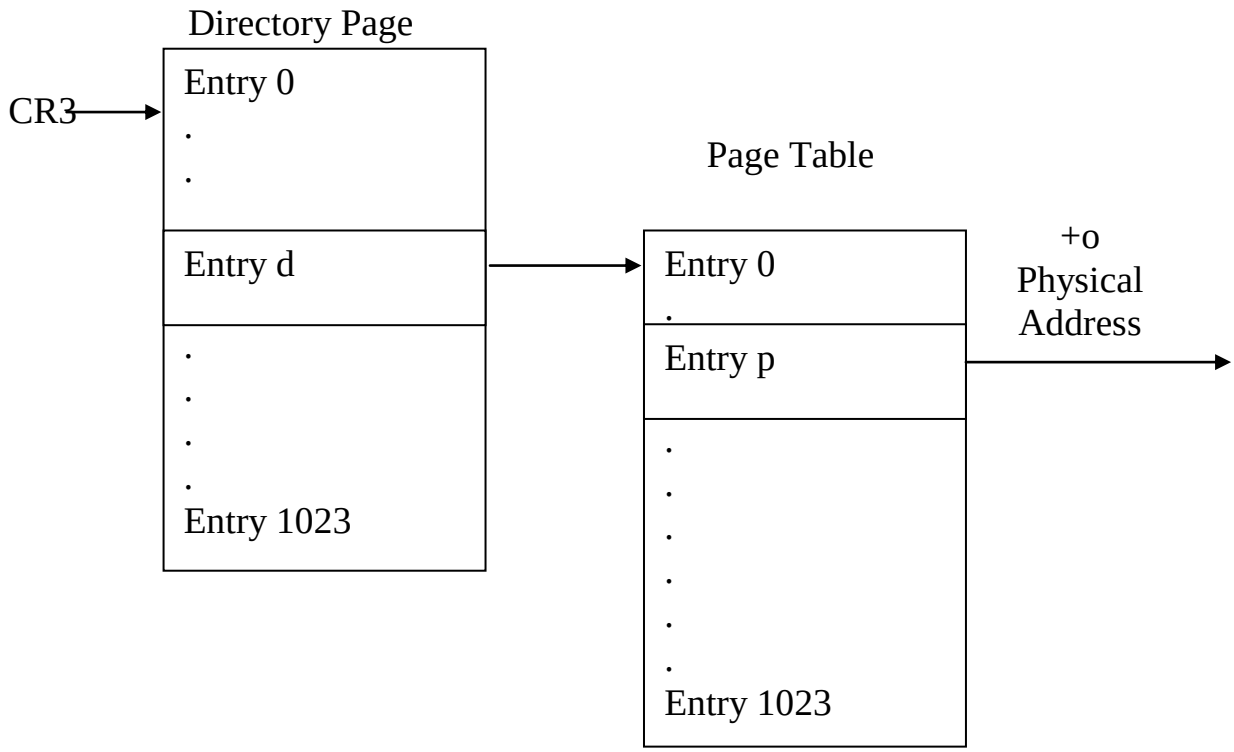
The contents of the CR3 register:
rrrr-rrrr-rrrr-rrrr-rrrr (20 bits)
The starting physical address of the process's current directory page:
rrrr-rrrr-rrrr-rrrr-rrrr-0000-0000-0000 (32 bits)

First processor access (directory page):
rrrr-rrrr-rrrr-rrrr-rrrr-dddd-dddd-dd00

This location contains a 20 bit table entry:
tttt-tttt-tttt-tttt-tttt
The starting physical address of a page table:
tttt-tttt-tttt-tttt-tttt-0000-0000-0000

The next processor access (page entry):
tttt-tttt-tttt-tttt-tttt-pppp-pppp-pp00
This location contains a 20 bit page frame value:
ffff-ffff-ffff-ffff-ffff

The final 32 bit physical address:
ffff-ffff-ffff-ffff-ffff-oooo-oooo-oooo



- Advantages of the double-paging:
  - Each application can access 4GB of memory.
  - Applications are insulated from each other.
  - No memory defregmentation is necessary: the pages need not to be contiguous.

- There are extra bits in the 32 bit page table entries (aside from the 20 bit addresses):
  - accessed bit
  - the page has been written to bit
  - page swapped out to disk bit
  - indicate if the page can be written to

15.4 Memory Allocation

- The C library functions:
  malloc(size)
  calloc(number, size)
  realloc(ptr, size)
  free(ptr)

- Windows 95 and up memory allocation:
  GlobalAlloc(flags, size) - it returns a virtual address
  flags: GMEM_FIXED, GMEM_MOVEABLE, GMEM_ZEROINIT,
  GMEM_DISCARDABLE (use together with moveable memory when reloading is faster then swapping)
  GlobalLock(hGlobal) lock the moveable memory before use
  GlobalUnlock(hGlobal) unlock the moveable memory after use
  GlobalRealloc(ptr, size, flags)
  size = GlobalSize(ptr)
  GlobalFree(ptr)
  GlobalMemoryStatus

- Heap functions:
  - Create and maintain contiguous block of virtual memory
    - *HeapCreate, HeapAllocate, HeapReAllocate, HeapFree*

15.5 File I/O

- In Windows is possible to read/write a file on big blocks.
- Windows supports long filenames.
- There is a common dialog box for file I/O.
- Functions:
  - *CreateFile, ReadFile, WriteFile*

15.6 Memory-Mapped I/O

- Sharing memory between processes:
  1. Create a file with *CreateFile*
  2. Get a handle to the map-file: *CreateFileMapping*
  3. Open the same file from another process: *OpenFileMapping*
  4. Access a part of this file: *MapViewOfFile* (map a part of the file in memory)
  5. Write back data to disk: *FlushViewOfFile*
  6. Unmapping of the mapped part: *UnmapViewOfFile*
  7. Close the file: *CloseHandle*

16. Dynamic-Link Libraries

16.1 Library types

DLL's are separate files containing functions that can be called by programs and other DLLs to perform certain jobs.

Dynamic linking: Windows uses to link a function call in one module to the actual function in the library module. Dynamic linking occurs at run time.
Static linking: occurs during program development to link various object (.OBJ) modules, run-time library (.LIB) files, and usually a compiled resource (.RES) file to create a Windows .EXE file.

16.2 Main features

- A DLL is an extension of Windows.
- Multiple programs may use the same DLL in runtime, different processes are sharing the same code. For large applications can be used effectively.
- All applications have separated:
  - Allocated memory
  - Windows created
  - Files opended
- The shared memory can be used for sharing data among the applications.

16.3 Writing an own DLL

1. Header file for variable/function/class declarations.

Use the **#define EXPORT extern "C" __declspec (dllexport)** to define the EXPORT.

2. C/C++ file for function/class definitions.
3. Compile/link the code to generate the DLL and the LIB import library.

16.4 Using the DLL

1. Include the header file with DLL function declarations.
2. Add the LIB import library to the project.
3. Move the DLL to the application's directory or to the standard search path.

References

[1] Charles Petzold: Programming Windows, Microsoft Press, 1998, ISBN 1-57231-995-X
[2] Microsoft Developer Network: https://msdn.microsoft.com/

Table of contents