Windows Programming

Lecture 10

Architecture of a standard Win32 API Application

Register the window class

Create the window

Retrieve messages from message queue and dispatch to ***WNDPROC*** for processing

Registering a window class

```
ATOM classAtom; WNDCLASS wc;

   wc.style            = 0;
   wc.lpfnWndProc      = myWindowProc;  // window procedure
   wc.cbClsExtra       = 0;
   wc.cbWndExtra       = 0;
   wc.hInstance        = hInstance;
   wc.hIcon            = NULL;
   wc.hCursor          = LoadCursor(NULL, IDC_UPARROW);
   wc.hbrBackground    = (HBRUSH)GetStockObject(GRAY_BRUSH);
   wc.lpszMenuName     = NULL;
   wc.lpszClassName    = "MyFirstWindowClass";

   classAtom = RegisterClass(&wc);

   if(!classAtom)
   {
           // error registering the window class
   }
```

Creating a window

```
HWND hwnd;
... ... // some code here

CreateWindow(
   "MyFirstWindowClass",    // registered class name
   "Virtual University",    // window name
   WS_OVERLAPPEDWINDOW | WS_VISIBLE,    // window style
   100,                    // horizontal position of window
   100,                    // vertical position of window
   400,                    // window width
   300,                    // window height
   NULL,                   // handle to parent or owner window
   NULL,                   // menu handle or child identifier
   hInstance,              // handle to application instance
   NULL                    // window-creation data
);
```

Window Styles

`WS_BORDER`	Creates a window that has a thin-line border.
`WS_CAPTION`	Creates a window that has a title bar (includes the WS_BORDER style).
`WS_CHILD`	Creates a child window. A window with this style cannot have a menu bar. This style cannot be used with the WS_POPUP style.
`WS_CHILDWINDOW`	Same as the WS_CHILD style.
`WS_CLIPCHILDREN`	Excludes the area occupied by child windows when drawing occurs within the parent window. This style is used when creating the parent window.

Window Styles Cont’d

WS_CLIPSIBLINGS	Clips child windows relative to each other; that is, when a particular child window receives a WM_PAINT message, the WS_CLIPSIBLINGS style clips all other overlapping child windows out of the region of the child window to be updated. If WS_CLIPSIBLINGS is not specified and child windows overlap, it is possible, when drawing within the client area of a child window, to draw within the client area of a neighboring child window.
WS_DISABLED	Creates a window that is initially disabled. A disabled window cannot receive input from the user. To change this after a window has been created, use EnableWindow

Window Styles Cont'd

`WS_DLGFRAME`	Creates a window that has a border of a style typically used with dialog boxes. A window with this style cannot have a title bar.
`WS_GROUP`	Specifies the first control of a group of controls. The group consists of this first control and all controls defined after it, up to the next control with the WS_GROUP style. The first control in each group usually has the WS_TABSTOP style so that the user can move from group to group. The user can subsequently change the keyboard focus from one control in the group to the next control in the group by using the direction keys. You can turn this style on and off to change dialog box navigation. To change this style after a window has been created, use SetWindowLong.

Window Styles Cont’d

`WS_HSCROLL`	Creates a window that has a horizontal scroll bar.
`WS_ICONIC`	Creates a window that is initially minimized. Same as the WS_MINIMIZE style.
`WS_MAXIMIZE`	Creates a window that is initially maximized.
`WS_MAXIMIZEBOX`	Creates a window that has a maximize button. Cannot be combined with the WS_EX_CONTEXTHELP style. The WS_SYSMENU style must also be specified.

Window Styles Cont’d

`WS_MINIMIZE`	Creates a window that is initially minimized. Same as the WS_ICONIC style.
`WS_MINIMIZEBOX`	Creates a window that has a minimize button. Cannot be combined with the WS_EX_CONTEXTHELP style. The WS_SYSMENU style must also be specified.
`WS_OVERLAPPED`	Creates an overlapped window. An overlapped window has a title bar and a border. Same as the WS_TILED style.
`WS_OVERLAPPEDWINDOW`	Creates an overlapped window with the WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles. Same as the WS_TILEDWINDOW style.

Window Styles Cont’d

`WS_POPUP`	Creates a pop-up window. This style cannot be used with the WS_CHILD style.
`WS_POPUPWINDOW`	Creates a pop-up window with WS_BORDER, WS_POPUP, and WS_SYSMENU styles. The WS_CAPTION and WS_POPUPWINDOW styles must be combined to make the window menu visible.
`WS_SIZEBOX`	Creates a window that has a sizing border. Same as the WS_THICKFRAME style.
`WS_SYSMENU`	Creates a window that has a window menu on its title bar. The WS_CAPTION style must also be specified.

Window Styles Cont’d

`WS_TABSTOP`	Specifies a control that can receive the keyboard focus when the user presses the TAB key. Pressing the TAB key changes the keyboard focus to the next control with the WS_TABSTOP style. You can turn this style on and off to change dialogbox navigation. To change this style after a window has been created, use **SetWindowLong**.
`WS_THICKFRAME`	Creates a window that has a sizing border. Same as the WS_SIZEBOX style.
`WS_TILED`	Creates an overlapped window. An overlapped window has a title bar and a border. Same as the WS_OVERLAPPED style.

Window Styles Cont'd

WS_TILEDWINDOW	Creates an overlapped window with the WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles. Same as the WS_OVERLAPPEDWINDOW style.
WS_VISIBLE	Creates a window that is initially visible. This style can be turned on and off by using ShowWindow or SetWindowPos.
WS_VSCROLL	Creates a window that has a vertical scroll bar.

Window Style Definitions

(defined in Winuser.H header file)

```
/*
 * Window Styles
 */
#define WS_OVERLAPPED                       0x00000000L
#define WS_POPUP                            0x80000000L
#define WS_CHILD                            0x40000000L
#define WS_MINIMIZE                         0x20000000L
#define WS_VISIBLE                          0x10000000L
#define WS_DISABLED                         0x08000000L
#define WS_CLIPSIBLINGS                     0x04000000L
#define WS_CLIPCHILDREN                     0x02000000L
```

Window Style Definitions Cont'd

```
#define WS_MAXIMIZE                     0x01000000L
#define WS_CAPTION                      0x00C00000L
                                           /* WS_BORDER |
                                           WS_DLGFRAME  */
#define WS_BORDER                       0x00800000L
#define WS_DLGFRAME                     0x00400000L
#define WS_VSCROLL                      0x00200000L
#define WS_HSCROLL                      0x00100000L
#define WS_SYSMENU                      0x00080000L
#define WS_THICKFRAME                   0x00040000L
#define WS_GROUP                        0x00020000L
```

Window Style Definitions Cont'd

```
#define WS_TABSTOP                   0x00010000L
#define WS_MINIMIZEBOX               0x00020000L
#define WS_MAXIMIZEBOX               0x00010000L
#define WS_TILED                     WS_OVERLAPPED
#define WS_ICONIC                    WS_MINIMIZE
#define WS_SIZEBOX                   WS_THICKFRAME
#define WS_TILEDWINDOW               WS_OVERLAPPEDWINDOW
```

Window Style Definitions Cont'd

```
/*
* Common Window Styles
 */
#define WS_OVERLAPPEDWINDOW          ( WS_OVERLAPPED | \
                                       WS_CAPTION | \
                                       WS_SYSMENU | \
                                       WS_THICKFRAME |\
                                       WS_MINIMIZEBOX |\
                                       WS_MAXIMIZEBOX )

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

#define WS_CHILDWINDOW               ( WS_CHILD)
```

Window Style Definitions Cont’d

```
/*
 * Extended Window Styles
 */
#define WS_EX_DLGMODALFRAME      0x00000001L
#define WS_EX_NOPARENTNOTIFY     0x00000004L
#define WS_EX_TOPMOST            0x00000008L
#define WS_EX_ACCEPTFILES        0x00000010L
#define WS_EX_TRANSPARENT        0x00000020L
#if(WINVER >= 0x0400)
#define WS_EX_MDICHILD           0x00000040L
#define WS_EX_TOOLWINDOW         0x00000080L
#define WS_EX_WINDOWEDGE         0x00000100L
#define WS_EX_CLIENTEDGE         0x00000200L
#define WS_EX_CONTEXTHELP        0x00000400L
```

Window Style Definitions Cont'd

```
#define WS_EX_RIGHT                  0x00001000L
#define WS_EX_LEFT                   0x00000000L
#define WS_EX_RTLREADING             0x00002000L
#define WS_EX_LTRREADING             0x00000000L
#define WS_EX_LEFTSCROLLBAR          0x00004000L
#define WS_EX_RIGHTSCROLLBAR         0x00000000L
#define WS_EX_CONTROLPARENT          0x00010000L
#define WS_EX_STATICEDGE             0x00020000L
#define WS_EX_APPWINDOW              0x00040000L
```

Window Style Definitions Cont'd

```
#define WS_EX_OVERLAPPEDWINDOW  (WS_EX_WINDOWEDGE |
                                 WS_EX_CLIENTEDGE)

#define WS_EX_PALETTEWINDOW     (WS_EX_WINDOWEDGE |
                                 WS_EX_TOOLWINDOW |
                                 WS_EX_TOPMOST)

#endif /* WINVER >= 0x0400 */
```

Bitwise Inclusive-OR Operator ‘|’

- The bitwise inclusive OR ‘|’ operator compares the values (in binary format) of each operand and yields a value whose bit pattern shows which bits in either of the operands has the value 1 (one). If both of the bits are 0 (zero), the result of the comparison is 0 (zero); otherwise, the result is 1 (one).

Bitwise Inclusive-OR Operator (|)

(Example)

The following example shows the values of a, b, and the result of a | b represented as 32-bit binary numbers:

bit pattern of **a**	0000000001011100
bit pattern of **b**	0000000000101110
bit pattern of **a | b**	0000000001111110

Messages and Message Queue

The system automatically creates a message queue for each thread. If the thread creates one or more windows, a message loop must be provided; this message loop retrieves messages from the thread's message queue and dispatches them to the appropriate window procedures.

Message Handling

- When a window is created, the system sends messages to the application message queue for each action.
- The application retrieves messages from the queue with `GetMessage()`
- Messages are dispatched to their respective windows' procedures with `DispathMessage()`
- A window processes messages sent to it, in the window procedure provided in window class.
- Optionally `TranslateMessage()` function can be used along with `GetMessage()` in the message loop

The Message Loop

```
MSG msg;
// code to register the window class
// code to create the window

while(GetMessage (&msg, NULL, 0, 0) > 0)
{
     TranslateMessage (&msg);
     // translate virtual-key messages into
character //messages
     DispatchMessage (&msg);
     // dispatch message to window procudure
}
```

The `GetMessage()` function *(prototype)*

```
BOOL GetMessage(LPMSG lpMsg,
                HWND hWnd,
                UINT wMsgFilterMin,
                UINT wMsgFilterMax );
```

The `GetMessage()` function *(explained)*

The `GetMessage()` function retrieves a message from the calling thread's message queue. The function dispatches incoming sent messages until a posted message is available for retrieval.

The `TranslateMessage()` function *(prototype)*

```
BOOL TranslateMessage(
                    const MSG* lpMsg);
```

The `TranslateMessage()` function *(explained)*

The `TranslateMessage()` function translates virtual-key messages into character messages. The character messages are posted to the calling thread's message queue, to be read the next time the thread calls the `GetMessage()` or `PeekMessage()` function.

The `DispatchMessage()` function *(prototype)*

```
LRESULT DispatchMessage(
                const MSG* lpmsg);
```

The `DispatchMessage()` function *(explained)*

The `DispatchMessage()` function dispatches a message to a window procedure. It is typically used to dispatch a message retrieved by the GetMessage function.

The Window Procedure *(prototype)*

Every window must have a Window Procedure. The name of the window procedure is user-defined.

The Generic application uses the following window procedure for the main window:

```
LRESULT WINAPI myWindowProc( HWND hwnd,
                             UINT message,
                             WPARAM wParam,
                             LPARAM lParam );
```

The **WINAPI** modifier is used because the window procedure must be declared with the standard call calling convention.

The Window Procedure *(explained)*

```
LRESULT CALLBACK myWindowProc(HWND hWnd, UINT message,
                              WPARAM wParam, LPARAM lParam)
{
  switch (message)
  {
     case WM_LBUTTONDOWN:
        MessageBox(hWnd, "Left mouse button pressed.",
                               "A Message", MB_OK);
        DestroyWindow(hWnd);
        break;

     case WM_DESTROY:
        PostQuitMessage(0);
        break;

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

Some Window Messages

(defined in **`Winuser.H`***)*

```
#define WM_CREATE                     0x0001
#define WM_DESTROY                    0x0002
#define WM_PAINT                      0x000F
#define WM_QUIT                       0x0012
#define WM_LBUTTONDOWN                0x0201
#define WM_RBUTTONDOWN                0x0204
```

Default Message Processing
`DefWindowProc()` *(prototype)*

```
LRESULT DefWindowProc( HWND hWnd,
                            UINT message,
                            WPARAM wParam,
                            LPARAM lParam );
```

Default Message Processing

`DefWindowProc()` *(explained)*

```
DefWindowProc(hWnd, message,
                        wParam, lParam)
```

This function calls the default window procedure to provide default processing for any window messages that an application does not process. This function ensures that every message is processed. DefWindowProc () is called with the same parameters received by the window procedure.

The `WM_QUIT` message and `GetMessage()` function

The `WM_QUIT` message indicates a request to terminate an application and is generated when the application calls the `PostQuitMessage()` function. It causes the `GetMessage()` function to return zero hence terminate the message loop.

The **MSG** Structure

The **MSG** structure contains message information from a thread’s message queue. It has the following form:

```
typedef struct tagMSG {// msg
               HWND   hwnd;
               UINT   message;
               WPARAM wParam;
               LPARAM lParam;
               DWORD  time;
               POINT  pt;
            } MSG;
```

`GetMessage()` and the `MSG` structure

Messages that Windows places in the application queue take the form of an `MSG` structure. This structure contains members that identify and contain information about the message. Application's message loop retrieves this structure from the application queue using `GetMessage()` function.

`DispatchMessage()` and the `MSG` structure

The message retrieved by `GetMessage()` function is dispatched to the appropriate window procedure using `DispatchMessage()` function. This message is packaged in an `MSG` structure whose different parameters contain information such as the window to which the message is destined, type of message and some additional information about the message.

What is a Windows Message?

A *message identifier* is a named constant *(a unique integer assigned to each event by the developers of Windows)* that identifies the purpose of a message. When a window procedure receives a message, it uses a message identifier to determine how to process the message. For example, the message identifier **`WM_PAINT`** tells the window procedure that the window's client area has changed and must be repainted.

Message Types

There are two types of messages that an application can receive:

- System-Defined Messages

  The system sends or posts a *system-defined message* when it communicates with an application.

- Application-Defined Messages

  An application can create messages to be used by its own windows or to communicate with windows in other processes.