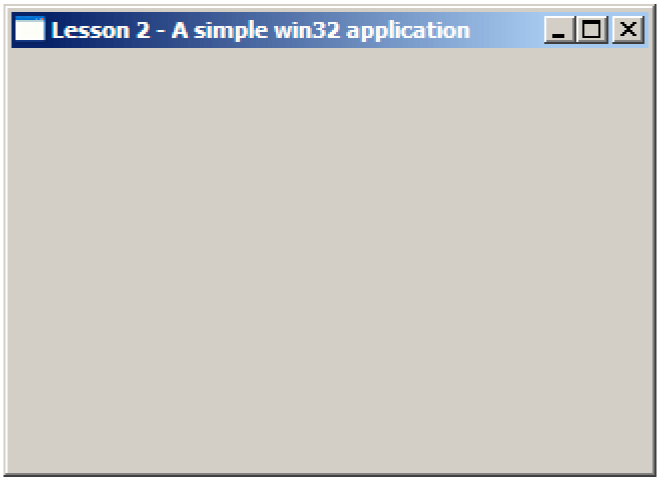
Lesson 4

Understanding messages and events

In this tutorial you will be introduced to the event-driven programming model. You will learn how Windows uses messages to communicate with applications, how event based programming works, what callback functions are, and while doing this create a basic windows application



Program : win4.c

```
#include <windows.h>

HWND hwndMain;        //Main window handle

// Callback function
LRESULT CALLBACK MainWndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam);
// Windows entry point
int WINAPI
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, INT nCmdShow)
{
    MSG msg; // MSG structure to store messages
    WNDCLASSEX wcx; // WINDOW class information

    // Initialize the struct to zero
    ZeroMemory(&wcx,sizeof(WNDCLASSEX));
    wcx.cbSize = sizeof(WNDCLASSEX); // Window size. Must always be sizeof(WNDCLASSEX)
    wcx.style = CS_HREDRAW|CS_VREDRAW |CS_DBLCLKS ; // Class styles
    wcx.lpfnWndProc = (WNDPROC)MainWndProc; // Pointer to the callback procedure
    wcx.cbClsExtra = 0; // Extra byte to allocate following the wndclassex structure
    wcx.cbWndExtra = 0; // Extra byte to allocate following an instance of the structure
    wcx.hInstance = hInstance; // Instance of the application
    wcx.hIcon = NULL; // Class Icon
    wcx.hCursor = LoadCursor(NULL, IDC_ARROW); // Class Cursor
    wcx.hbrBackground = (HBRUSH)(COLOR_WINDOW); // Background brush
    wcx.lpszMenuName = NULL; // Menu resource
    wcx.lpszClassName = "Lesson2"; // Name of this class
    wcx.hIconSm = NULL; // Small icon for this class

    // Register this window class with MS-Windows
    if (!RegisterClassEx(&wcx))
        return 0;

    // Create the window
    hwndMain = CreateWindowEx(0, //Extended window style
                              "Lesson2", // Window class name
```

```
                                        "Lesson 2 - A simple win32 application", // Window title
                                        WS_OVERLAPPEDWINDOW, // Window style
                                        CW_USEDEFAULT,CW_USEDEFAULT, // (x,y) pos of the window
                                        CW_USEDEFAULT,CW_USEDEFAULT, // Width and height of the window
                                        HWND_DESKTOP, // HWND of the parent window (can be null also)
                                        NULL, // Handle to menu
                                        hInstance, // Handle to application instance
                                        NULL); // Pointer to window creation data

          // Check if window creation was successful
          if (!hwndMain)
                    return 0;

          // Make the window visible
          ShowWindow(hwndMain,SW_SHOW);

          // Process messages coming to this window
          while (GetMessage(&msg,NULL,0,0))
          {
                    TranslateMessage(&msg);
                    DispatchMessage(&msg);
          }

          // return value to the system
          return msg.wParam;
}

```

```
LRESULT CALLBACK MainWndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam)
{
	switch (msg)
	{
		case WM_DESTROY:
			// User closed the window
			PostQuitMessage(0);
			break;
		default:
			// Call the default window handler
			return DefWindowProc(hwnd,msg,wParam,lParam);
	}
	return 0;
}
```

Breaking it up

Messages and the MSG Structure

The MSG structure is what stores the messages received by your application. Before going any further, lets take a look at the event-driven programming model.

Event driven programming model

Let us understand how event driven programming works.

1. The user clicks the maximize button.
2. Windows tells your application that the maximize button has been pressed.
3. Your application then redraws its window so that it covers the screen.

Every time windows has to communicate with your application, it sends messages to your application. Once all initializations have been done and the window shown on screen, all your application has to do is poll for windows messages.

The lines upto 51 create and show the window and the lines 52-57 poll for messages.

The GetMessage() function gets the next message to be processed from the message queue. GetMessage() returns a non-zero value for every message other than WM_QUIT. This means that the while loop continues until it is time to quit.

TranslateMessage() translates virtual key messages to character messages.

DispatchMessage() dispatches the message to a window procedure. This means that for messages coming to our window, the MainWndProc() is called by Windows(tm) through DispatchMessage().

How does Windows(tm) know which function to call? Well, we tell Windows(tm) during WNDCLASSEX initialization [line 18].

WNDCLASSEX structure

Every window that you create has an associated WNDCLASSEX structure. The WNDCLASSEX structure provides all the information necessary for Windows(tm) to do perform window related functions like drawing its icon, cursor, menu, calling the callback function which will receive messages and so on.

The WNDCLASSEX structure has been defined in lecture-3.

Registering your window class

After you've created your window class, you need to tell Windows(tm) about it. This is done by registering the class with windows. The function call is RegisterClassEx(..). Once this is done, you can create instances of this window by calling CreateWindowEx(..) with the proper arguments.

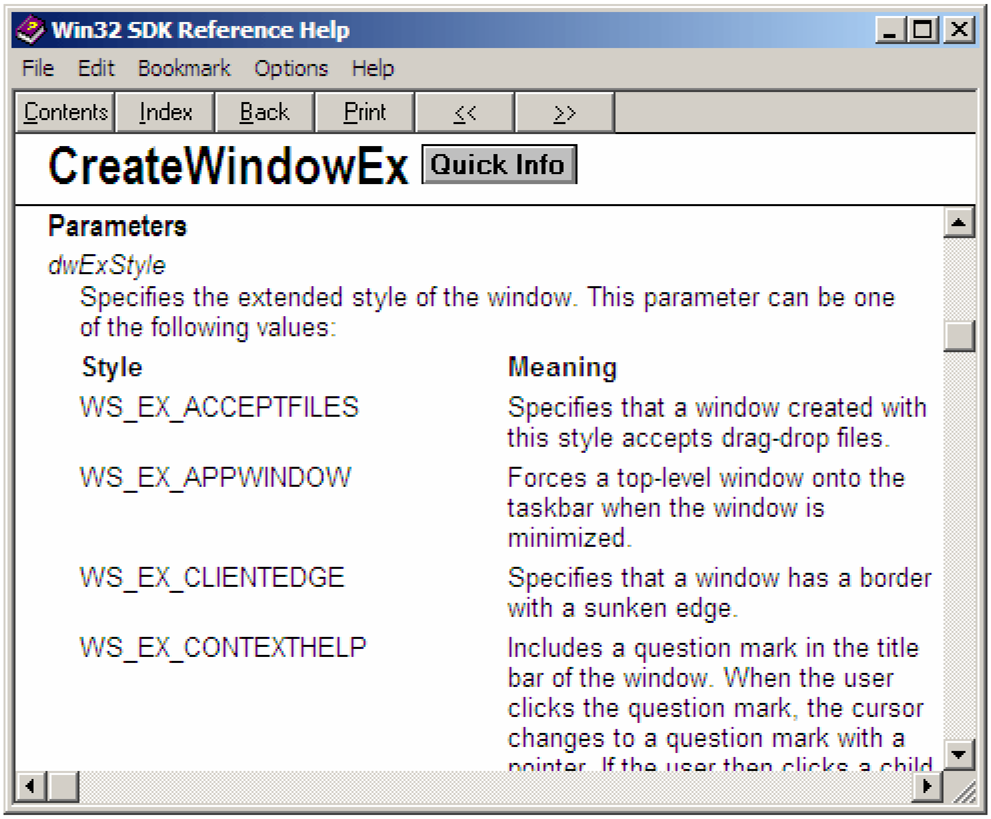
Creating the window

A window is created by calling the CreateWindowEx(..) defined as,

```
HWND CreateWindowEx(
   DWORD dwExStyle,                     // extended window style
   LPCTSTR lpClassName,          // pointer to registered class name
   LPCTSTR lpWindowName,       // pointer to window name
   DWORD dwStyle,                        // window style
   int x,                         // horizontal position of window
   int y,                         // vertical position of window
   int nWidth,                            // window width
   int nHeight,                // window height
   HWND hWndParent,                  // handle to parent or owner window
   HMENU hMenu,                        // handle to menu, or child-window identifier
   HINSTANCE hInstance,            // handle to application instance
   LPVOID lpParam                      // pointer to window-creation data
  );
```

Lines 34-43 create the window. If the creation was successful a non-zero handle is returned by CreateWindowEx after which ShowWindow() shows the window on the screen.

TIP: It is a good idea to keep referring to these functions in your sdk docs while reading this tutorial.



CreateWindowEx

Parameters

dwExStyle

Specifies the extended style of the window. This parameter can be one of the following values:

Style	Meaning
WS_EX_ACCEPTFILES	Specifies that a window created with this style accepts drag-drop files.
WS_EX_APPWINDOW	Forces a top-level window onto the taskbar when the window is minimized.
WS_EX_CLIENTEDGE	Specifies that a window has a border with a sunken edge.
WS_EX_CONTEXTHELP	Includes a question mark in the title bar of the window. When the user clicks the question mark, the cursor changes to a question mark with a pointer. If the user then clicks a child

Callback functions

A callback function is the one that receives the messages sent to your application. This is where you do something about the message. We provide a pointer to this function while defining the window class [line 18].

Callback functions have to be defined as,

```
LRESULT CALLBACK
function-name(
                    HWND hwnd,     // Handle of window which received this
message
                    UINT msg,          // The message
                    WPARAM wParam,          // Extra information
                    LPARAM lParam// Extra information
                    );
```

HWND hwnd

The handle of the window is specified so that you know which window to act upon. This is necessary because you may have created more than one instance of the window.

UINT msg

This contains the message sent.

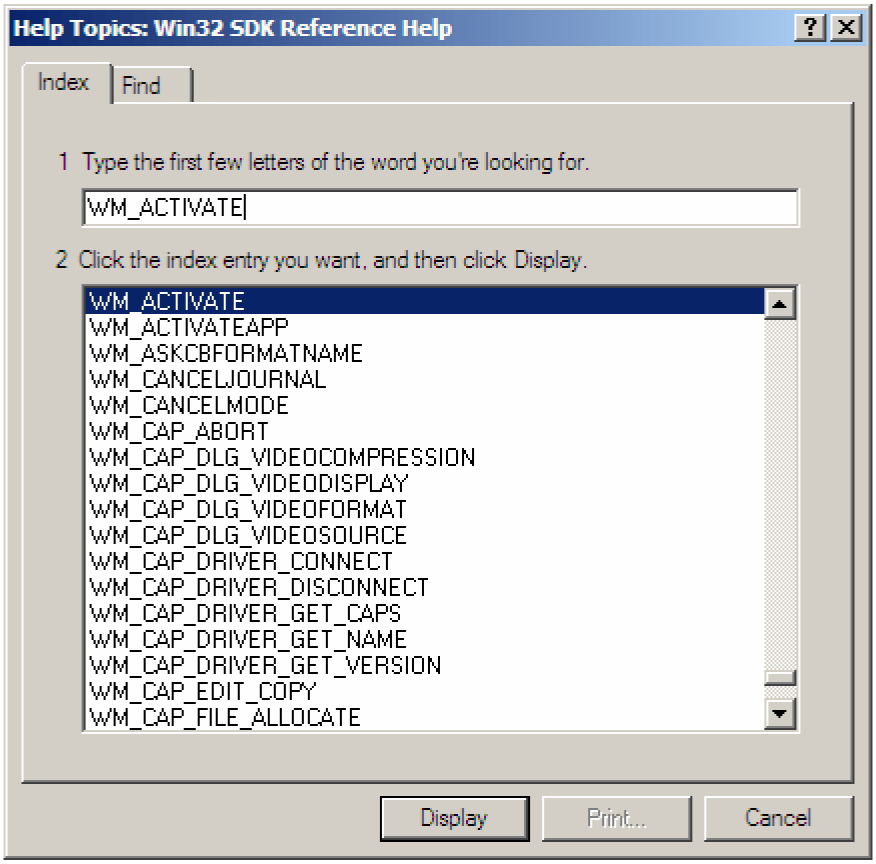
WPARAM wParam and WPARAM lParam

wParam and lParam are used to pass extra info about the message. For example a WM_LBUTTONDOWN (left mouse button down) message will have the x and y co-ordinates as the upper and lower word of lParam and wParam will tell if any modifier keys (ctrl, alt, shift) have been pressed.

MainWndProc

```
LRESULT CALLBACK MainWndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam)
{
    switch (msg)
    {
    ...
```

The switch statement lets us select which message was sent. There are over 200 messages that windows can send your application. To read about them, just search for WM_ in your sdk docs.



1 Type the first few letters of the word you're looking for.

WM_ACTIVATE

2 Click the index entry you want, and then click Display.

WM_ACTIVATE
WM_ACTIVATEAPP
WM_ASKCBFORMATNAME
WM_CANCELJOURNAL
WM_CANCELMODE
WM_CAP_ABORT
WM_CAP_DLG_VIDEOCOMPRESSION
WM_CAP_DLG_VIDEODISPLAY
WM_CAP_DLG_VIDEOFORMAT
WM_CAP_DLG_VIDEOSOURCE
WM_CAP_DRIVER_CONNECT
WM_CAP_DRIVER_DISCONNECT
WM_CAP_DRIVER_GET_CAPS
WM_CAP_DRIVER_GET_NAME
WM_CAP_DRIVER_GET_VERSION
WM_CAP_EDIT_COPY
WM_CAP_FILE_ALLOCATE

WM_DESTROY

```
            ...
67          case WM_DESTROY:
68                  // User closed the window
69                  PostQuitMessage(0);
70                  break;
            ...
```

The WM_DESTROY message is sent to your application when the user teminates the application either by clicking the X at the upper right corner, pressing Alt+F4, or quits the application by other means.

PostQuitMessage() causes GetMessage(..) [line 53] to return false and thus breaking out of the while loop and exiting the application. The argument to PostQuitMessage is the return value to the system.

DefWindowProc(..)

What about the other 200 or so messages? Surely you, the programmer, aren't going to write code for all the 200 messages. Fortunately, Windows(tm) provides the DefWindowProc(..) function which handles all the messages. For the purposes of displaying a simple window, your MainWndProc could very well have consisted of

```
LRESULT CALLBACK MainWndProc(HWND hwnd,UINT msg,WPARAM
wParam,LPARAM lParam)
{
            return DefWindowProc(hwnd,msg,wParam,lParam);
}
```

What this means is that every time you want to do something about a message, add the case switch for the message and write the code which does something about it. All messages that you don't want to handle should be passed to the DefWindowProc(). This is what we have done in our code.

```
            ...
71                      default:
72                                  // Call the default window handler
73                                  return DefWindowProc(hwnd,msg,wParam,lParam);
            ...
```

Adding Functionality

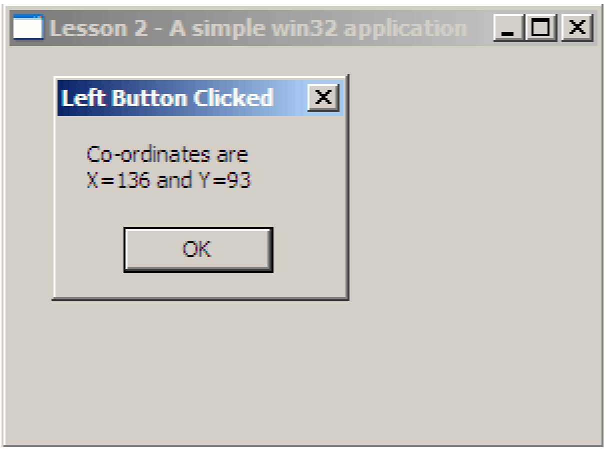
Lets pop up a MessageBox which will display the co-ordinates of the point where the left mouse button was pressed. To do this you will have to handle the WM_LBUTTONDOWN message.

Add this code at line 70

```
            ...
```

```
        PostQuitMessage(0);
                break;
        case WM_LBUTTONDOWN:
                pt.x = LOWORD(lParam);
                pt.y = HIWORD(lParam);
                wsprintf(str,"Co-ordinates are\nX=%i and Y=%i",pt.x,pt.y);
                MessageBox(hwnd, str, "Left Button Clicked", MB_OK);
                break;
        default:
        ...
```

Press F9. This is what you should see when you click anywhere inside the window.



Co-ordinates are
X=136 and Y=93

Exercise

Try this exercise. Pop up a message every time a key is pressed on the keyboard.
Hint: Handle the WM_CHAR message. Remember to refer to your sdk docs.

If you can manage that, give yourself a pat on the back. You now understand the basics of event-driven programming - the mechanism which Windows(tm) uses to communicate with your application. You have crossed one of the more difficult hurdles in learning windows programming.

Don't worry if you could not do the exercise or if things are still a bit hazy. These concepts will be used in every single lesson after this and it will soon become second nature to you.

Previous | Up | Next

Last Updated: February 12, 2003