[next] -->

Quick introduction to Windows API

Andrew M. Steane,
Exeter College, Oxford University and Centre for Quantum Computing, Oxford Unverisity
This page: started 2007, updated 2009

Contents

0. Introduction

The Windows API (application programming interface) allows user-written programs to interact with Windows, for example to display things on screen and get input from mouse and keyboard. All Windows programs except console programs must interact with the Windows API regardless of the language. The most recent version (as of 2007) is Win32 API. Microsoft maintains an extensive library of developer help information at msdn, but the presentation is often hard to follow and I would say this is not a good place to learn the basics.

This tutorial is intended for people reasonably familiar with C or C++ in the traditional scenario of console input/output (using things like gets and printf), and who now want to write windows applications. For style guidelines and other information on C++, go here.

I first met WinAPI (without knowing that is what I was meeting) through microsoft Visual C++. That developer environment provides a bunch of code to get you started, as well as something called a resources file which is new to programmers used to old-style console programming. This combination gives the user the distinct impression that you can no longer write your own self-contained program made purely of .h files and .cpp files and compiled with your favourite compiler. However, that is a false impression. Although the resources file is useful, my advice is to get started without it, and then bring it in once you are comfortable.

A further reservation I have about MSVC++, is that it makes it very difficult to edit the resources manually, or bring them in from other code. It is well-supported however, and excellent in many ways, so you may want to use it if you have the money. I have compiled a few notes on issues I have met with Visual C++ in hopes that they may help someone. I have also used a rather nice free developer environment for C++ on Windows called Dev-Cpp from Bloodshed software (don't worry about the odd name: the software is good). Another well-supported free developer environment is CodeBlocks

I found theForger's Win32 API Tutorial (Brook Miles) quite useful, especially the fully working code examples. Two other excellent sites are Reliable Software and Catch22. To learn Win32 API, I suggest you start here, then refer to theForger for further details, then when you have a good general sense of how to do things, go to the Reliable Software tutorial and Catch22 tutorials. The former provides, for example, general purpose classes and methods for API, and the second takes you through many useful methods that are not clear at msdn.

1. WinMain and WndProc

To get started, I suggest you write the simplest possible program and see if you can compile and run it. Here is the program:

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hThisInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
MessageBox(NULL, "Yes, I remember Adlestrop", "A minimal windows program", MB_OK);
return 0;
}
```

That's all you need. This program should pop up a simple message with an "OK" button. It's up to you to sort out any options you need to supply to your compiler in order to get it to work. For example, in the developer environment I am using, I had to tell it I wanted to create a new Windows application (not a "console") application. I then deleted all the "getting started" source code it offered me, and replaced it with the above. It compiled and ran fine. In Microsoft visual studio, however, I had to set a project option: see visual c notes for further information. To avoid the need for this option setting, you have to add a couple of type conversions to the above code, as follows:

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hThisInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
MessageBox(NULL, (LPCWSTR) "Yes, I remember Adlestrop", (LPCWSTR) "A minimal windows program", MB_OK);
return 0;
}
```

Now let's turn to longer programs. A typical windows program (in C or C++) begins with the following code:

```
#include <windows.h>

const char g_class_name[] = "myWindowClass";

// declare the WndProc function (see later)
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam);

// here is the main function:
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{ ... }
```

I will describe the code for the WinMain function in a moment. You have two basic functions: the WinMain one (which replaces the old "main(...)") and a function it calls, typically named WndProc or WindowProcedure or something like that. Although I will show you the code inside WinMain, you don't need to modify it for the moment. Your main business is with WinProc. You should regard WinMain as more or less a system-provided function (you might make minor modifications later on), while WndProc is your function, and you should make it your own by writing it from scratch!

Before we get on to that, however, let's fill in the body of WinMain:

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
```

```
    MSG Msg;

    //First we create a structure describing the window
    wc.cbSize        = sizeof(WNDCLASSEX);
    wc.style         = 0;
    wc.lpfnWndProc   = WndProc;       // N.B. here we specify the name of our function
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = hInstance;
    wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wc.lpszMenuName  = NULL;
    wc.lpszClassName = g_class_name;
    wc.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

    // Now we call RegisterClassEx. This is called "registering the window class".
    // It tells the operating system we are here and want a window.
    if (!RegisterClassEx(&wc))
    {
       MessageBox(NULL, "Window Registration Failed!", "Error!", MB_ICONEXCLAMATION | MB_OK);
       return 0;
    }

    // Ok, the operating system says we can have a window. Now we create it using CreateWindowEx.
    // The parameters here mainly define the appearance of the window: what the border looks like,
    // the title if any, etc.
    hwnd = CreateWindowEx(
       WS_EX_CLIENTEDGE,                        // what the border looks like
       g_class_name,
       "The title of my window",                // text appearing in top bar
       WS_OVERLAPPEDWINDOW,
       CW_USEDEFAULT, CW_USEDEFAULT, 240, 120,  // window xpos, ypos, width, height
       NULL, NULL, hInstance, NULL);

    if (hwnd == NULL)
    {
       MessageBox(NULL, "Window Creation Failed!", "Error!", MB_ICONEXCLAMATION | MB_OK);
       return 0;
    }

    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);

    /* And finally: the Message Loop. This while loop is perpetually going round and round
       at the "base" of your program, picking up messages from the keyboard, mouse and other
       devices, and calling the WndProc function (i.e. the function specified above when we set
       wc.lpfnWndProc).
    */
    while ( GetMessage(&Msg, NULL, 0, 0) > 0 )
    {
       TranslateMessage(&Msg);
       DispatchMessage(&Msg);
    }
    return Msg.wParam;
}
```

For the moment, I remind you, you are not going to modify this code, just include in your program source, in a file called "myprog_main.cpp" or something like that.

To get a fully functioning program, it only remains to write the WndProc function code. Here is an example:

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
// This is the function repeatedly called by WinMain. It receives and reacts to messages.
// hwnd is the window,
```

```
   {
   case WM_CLOSE:
      DestroyWindow(hwnd);
      return 0;
   case WM_DESTROY:
      PostQuitMessage(0);
      return 0;
   default:
      return DefWindowProc(hwnd, msg, wParam, lParam);
   }}
}
```

This particular example is rather minimal: it does almost nothing. If you run the program, it just displays the window and allows you to resize it in the normal way and close it by clicking the cross box.

For clarity, you can find the whole, fully functioning program and source code here.

Note that every case in the switch statement finishes by returning. This is the practice I encourage you to adopt. You must always include the default case, for any messages you don't pick up, and there call the windows-provided function DefWindowProc. I don't recommend having further code after the switch, because that switch is going to get longer as we add things, and if you put things after the switch it will begin to make the code hard to read. Rather, put everything you need within each case, even if that means repeating stuff. *Then make sure every case finishes with a return statement*. Obviously, once you are expert you can ignore this advice.

2. Hello world

We will now provide the message "hello world" in various ways. The first and easiest is to modify the main window title string from "The title of my window", to "hello world". Go back and find it in WinMain, carry out the modification, and observe the results. It works, but obviously that is not much use.

Now add the following function to your code (e.g. just above WndProc):

```
void disp(HWND hwnd, char* s)
// display a string
{  HFONT hfont, hOldfont;
   HDC hdc;

   hfont = (HFONT) GetStockObject(ANSI_VAR_FONT);    // obtain a standard font
   hdc = GetDC(hwnd);                          // point to the "device context" for this window
   hOldfont = (HFONT) SelectObject(hdc, hfont);      // select font into the device context
   if (hOldfont)  // if succesful
   {
      TextOut(hdc, 10, 50, s, strlen(s));
      SelectObject(hdc, hOldfont);         // put the previous font back into dc
   }
   else MessageBox(hwnd, "disp could not select the font", "Error!",
                   MB_ICONEXCLAMATION | MB_OK);

   ReleaseDC(hwnd, hdc);                   // tidy up
}
```

and modify WndProc to the following (you only need to add the six new lines):

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   switch(msg)
   {
   case WM_SIZE:
     disp(hwnd, "hello world");
     return 0;
   case WM_LBUTTONDOWN:
     MessageBox(hwnd, "hello again", "box title", MB_OK);
     return 0;
```

```
    return 0;
  case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
  default:
    return DefWindowProc(hwnd, msg, wParam, lParam);
  }
}
```

We have added two new features. First, the program displays "hello world". Second, if you click the window, it pops up a message box saying "hello again". You should guess from the code above that clicking the left mouse button in the window generates the message WM_LBUTTONDOWN, and causes our code to call the MessageBox function. The MessageBox function is provided by windows with syntax you can deduce from this example. In a large program, you could for example initiate some long calculation at the "case WM_LBUTTONDOWN:" statement, and then call MessageBox when the long calculation is finished.

Our own function, disp, is called in response to the WM_SIZE message. This message is generated whenever the window size is changed, and this includes when the window is first created. If you move another window on top of our program's little window, and then move it away again, you will see the "hello world" text has disappeared. This is because the window has not been instructed to redraw it. If you change the size of the window, however, the text will reappear, because the WM_SIZE message is sent.

Usually things which are to be displayed will be displayed in response to the WM_PAINT message, but that message requires some further code so I have not invoked it yet.

2.1 Device context

The disp(...) function introduces an important basic concept in Windows API programming. This is the concept of the "device context". Each device, such as a screen, a printer, etc., has a device context. When you want to display on the device, you interact with the device context (dc). This is a logical construct, and windows takes care of its interation with the hardware device. The result is that you have to go through the following sequence in order to display things: get the dc, write stuff to it, release the dc. In the case of text, there is the further issue of what font the device should use. You must say, and you do that by a call to SelectObject. This says to the dc "use this font" and the dc responds by saying, "ok, and here is the font I was using before". The idea is that a good user will remember that font and give it back the dc when he is finished, by another call to SelectObject. The final issue is, where to we get a font from? Later you will learn to construct one using CreateFont, but for now we just get a standard one provided by the system, using GetStockObject.

The types HWND, HDC, HFONT are handles (i.e. pointers) to the various objects: obviously it is quicker to pass pointers around rather than the objects themselves.

As you get more experienced with windows programming, you will find yourself using TextOut less and "controls" more. We will come to controls in a moment. However, it remains true that TextOut is a useful low-level function and it can be a good companion during debugging.

At the moment our disp() function always shows its text at the same point on screen. A further call to it will overwrite whatever is already there. To make it more powerful, you could provide it with parameters such as (..., int x, int y, ...) to allow the user to specify any point on screen. Sooner or later however you are likely to want to scroll the screen. Fortunately this is quite easy to do.

[top] . [next] -->