Windows Programming/Message Loop Architecture

< Windows Programming

Contents

WinMain [edit]

Programming Windows in C can be a little bit different from what most people are used to. First off, there isn't a **main()** function, but instead there is a **_tWinMain()** function to start your program off. _tWinMain() is defined as a macro in tchar.h as such:

```
#ifdef _UNICODE
#define _tWinMain wWinMain
#else
#define _tWinMain WinMain
#endif
```

This means that Windows functions can be written easily in Unicode or ASCII. Besides the difference in function name, _tWinMain also has different parameters than the standard main function:

```
int WinMain(HINSTANCE hThisInstance,
            HINSTANCE hPrevInstance,
            LPSTR lpszArgument,  // LPWSTR for wWinMain
            int iCmdShow);
```

HINSTANCE objects are references to a program instance. hThisInstance is a reference to the current program, and hPrevInstance is a reference to any previously running versions of this same program. However, in Windows NT, the hPrevInstance data object is always NULL. This means that we can't use the hPrevInstance value to determine if there are other copies of the same program running on your system. There are different ways of checking for other copies of a program running, and we will discuss those methods later.

lpszArgument essentially contains all the information that the argc and argv variables used to show, except that the command-line arguments are not broken up into a vector. This means that if you want to sort individual arguments from the command line, you need to either break them up yourself, or call one of the available functions to break it up for you. We will discuss these functions later.

The last argument, "int iCmdShow" is a data item of type int (integer) that determines whether or not a graphical window should be displayed immediately, or if the program should run minimized.

WinMain has a number of different jobs:

1. Register the window classes to be used by the program

2. Create any Windows used by the program
3. Run the message loop

We will explain each of those tasks in more detail.

Register Window Classes [edit]

Every little graphical detail you see on the screen is known as a "window". Each program with a box around it, each button, each text box are all called windows. In fact, all of these various objects all get created in the same manner. This must mean that there are a large number of options and customizations available in order to get things as different as textboxes and scroll bars from the same method. Each window has an associated "Window Class" that needs to be registered with the system, to specify the different properties of the window. This is done by following 2 steps:

1. Fill in the fields of the WNDCLASS data object
2. Pass the WNDCLASS object to the RegisterClass function.

Also, there is an "extended" version of this procedure, that can be followed with similar results:

1. Fill in the fields of the WNDCLASSEX object
2. Pass the WNDCLASSEX object to the RegisterClassEx function.

Either of these methods can be used to register the class, but the -Ex version has a few more options.

Once the class is registered, you can discard the WNDCLASS structure, because you don't need it anymore.

Create Windows [edit]

Creating Windows can be done with the CreateWindow or the CreateWindowEx functions. Both perform the same general task, but again, the -Ex version has more options. You pass some specifics to the CreateWindow function, such as the window size, the window location (the X and Y coordinates), the window title, etc. CreateWindow will return a HWND data object, that is a handle to the newly created window. Next, most programs will pass this handle to the **ShowWindow** function, to make the window appear on the screen. The way CreateWindow() creates a window is as follows:

```
hwnd=CreateWindowEx(
     WS_EX_CLIENTEDGE,
     g_szClassName,
     "The title of my window",
     WS_OVERLAPPEDWINDOW,
     CW_USEDEFAULT,CW_USEDEFAULT,240,120,
     NULL,NULL,hInstance,NULL);
```

The first parameter WS_EX_CLIENTEDGE is the extended window style. Next we have the class name g_szClassName, this tells the system what kind of window to create. Since we want to create a window from the class we just registered, we use the name of that class. After that we specify our window name or title which is the text that will be displayed in the caption, or title bar on our window. The parameter we have as WS_OVERLAPPEDWINDOW is the Window Style parameter. There are quite a few of these and you should look them up and experiment to find out what they do. The next four parameters (CW_USEDEFAULT,CW_USEDEFAULT,240,120) are the X and Y co-ordinates for the top left corner of your window, and the width and height of the window. The X and Y co-ordinates are set to CW_USEDEFAULT to let the window choose where on the screen to put the window. Next, (NULL,NULL,hInstance,NULL) we have the parent window handle , the menu handle, the application instance handle, and a pointer handle to window

creation data. In windows, the window on the screen is arranged in a hierarchy of the parent and child windows. When one sees a button on a window, the button is the Child and it is contained within the window that is it's Parent. In this example, the parent handle is NULL because we have no parent, this is our main or top level window. The menu is NULL for now since we don't have one yet. The instance handle is set to the value that is passed in as the first parameter of the WinMain(). The creation data that can be used to send additional data to the window that is being created is NULL.

Message Loop [edit]

Once the window is created, the window will interact with the rest of the system by way of **messages**. The system sends messages to the window, and the window sends messages back. Most programs in fact don't do anything but read messages and respond to them!

Messages come in the form of an MSG data type. This data object is passed to the GetMessage() function, which reads a message from the message queue, or waits for a new message from the system. Next, the message is sent to the TranslateMessage function, which takes care of some simple tasks such as translating to Unicode or not. Finally, the message is sent to the window for processing using the DispatchMessage function.

Here is an example:

```
MSG msg;
BOOL bRet;
while( (bRet = GetMessage( &msg, NULL, 0, 0 )) != 0)
{
   if (bRet == -1)
   {
       // handle the error and possibly exit
   }
   else
   {
       TranslateMessage(&msg);
       DispatchMessage(&msg);
   }
}

return msg.wParam;
```

That last line will be explained later.

The Window Procedure [edit]

A window procedure may be named anything you want, but the general prototype for one is as follows:

```
LRESULT CALLBACK WinProc (HWND hwnd,
                          UINT msg,
                          WPARAM wParam,
                          LPARAM lParam);
```

An LRESULT data type is a generic 32-bit data object, that can be type-casted to contain any 32-bit value (including a pointer). The hwnd parameter is a handle to the window itself. The msg data value contains the current message from the operating system, and the WPARAM and LPARAM values contain the arguments for that message. For instance, if a button is pressed on the keyboard, the msg field will contain the message

WM_KEYDOWN, and the WPARAM field will contain the actual letter pressed ('A' for instance), and the LPARAM field will contain information on whether or not the CTRL, ALT, or SHIFT buttons are down, and whether the type-matic repeat function has been triggered. Several macros have been defined that are very useful in separating out the WPARAM and LPARAM into different sized chunks:

LOWORD(x)
: returns the low 16-bits of the 32-bit argument

HIWORD(x)
: returns the high 16-bits of the 32-bit argument

LOBYTE(x)
: returns the low 8-bits of the 16-bit argument

HIBYTE(x)
: returns the high 8-bits of the 16-bit argument

For instance, to access the 2nd byte in the wParam field, we will use the macros as follows:

```
HIBYTE(LOWORD(wParam));
```

Since the window procedure only has two available parameters, these parameters are often packed with data. These macros are very useful in separating that information out into different fields.

Here is an example of a general Window Procedure, that we will explain:

```
LRESULT CALLBACK MyWinProc (HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
   switch (msg)
   {
       case WM_DESTROY:
           PostQuitMessage(0);
           break;
       default:
           return DefWindowProc (hwnd, msg, wParam, lParam);
   }
   return 0;
}
```

Most window procedures only contain a simple loop that searches for particular messages from the system, and then acts on them. In this example, we are only looking for the WM_DESTROY message, which is the message that the kernel sends to the window when the window needs to close. In response to the WM_DESTROY message, the window calls the PostQuitMessage function, which puts a WM_QUIT message (which is defined as 0) into the message queue. When the message loop (described above) gets the WM_QUIT message, it breaks from the loop, and returns the value from the PostQuitMessage function.

Any message that is not handled by the loop should be (must be) passed to the DefWindowProc function. DefWindowProc will perform some default actions based on the messages received, but it won't do anything interesting. If you want your program to do something, you will need to handle these messages yourself.

Messages [edit]

There are a few other messages that we will talk about later on:

WM_CREATE

Your window receives this message only once, when it is first created. Use this message to perform tasks that need to be handled in the beginning, such as initializing variables, allocating memory, or creating child windows (buttons and textboxes).

WM_PAINT

This message indicates that it is time for the program to redraw itself. Use the graphical functions to redraw whatever is supposed to be on the window. If you don't draw anything, then the window will either be a boring white (or grey) background, or if the background was not erased, will keep whatever image is already shown on it (which looks unstable.)

WM_COMMAND

This is a general message that indicates that the user has done something on your window. Either the user has clicked a button, or the user has selected a menu item, or the user has pressed a special "Accelerator" key sequence. The WPARAM and LPARAM fields will contain some descriptions on what happened, so you can find a way to react to this. If you do not process the WM_COMMAND messages, the user will not be able to click any buttons, or select any menu items, and that will be very frustrating indeed.

WM_CLOSE

The user has decided to close the window, so the kernel sends the WM_CLOSE message. This is the final chance to preserve the window as necessary - if you don't want it closed completely, you should handle the WM_CLOSE message and ensure that it does not destroy the window. If the WM_CLOSE message is passed to the DefWindowProc, then the window will next receive the WM_DESTROY message.

WM_DESTROY

The WM_DESTROY indicates that a given window is removed from the screen and will be unloaded from memory. Normally, your program can post the WM_QUIT message to exit the program by calling PostQuitMessage().

These are some of the most basic messages, and we will discuss other messages as the need arises.

- Hint SendMessage(hwnd,MACRO,NULL,NULL) can be used to send user defined messages.