

How Windows Works

Notes for CS130

Dr. Beeson

Event-Driven Programming. In Windows, and also in Java applets and Mac programs, the program responds to user-initiated *events*: mouse clicks and key presses. In a traditional program, the user responds to prompts issued by the program. In event-driven programming, the user is in control. In traditional programming, the program is in control. Event-driven programming is more difficult than traditional programming, because the program must be ready to handle any possible event at any time.

Some events, moreover, are internally generated (that is, only indirectly caused by the user). An example is this: when a dialog box is closed, the part of the screen that was covered by the dialog box must be “repainted”. The program that owns that window is responsible for being able to repaint the window at any time. A traditional program could print output to the screen, and there its responsibility ended.

I will explain the architecture of Windows that makes event-driven programming possible. This architecture underlies all the methods of writing Windows programs.

Different Ways of Writing Windows Programs

Windows SDK or API (Software Development Kit or Application Programming Interface). This was the original method. There are still many Windows programs written this way that need maintenance (including for example Microsoft Word and Excel).

MFC (Microsoft Foundation Classes) This was how people wrote Windows programs in the period 1994-2003, approximately.

There are still many MFC programs in the world that need maintenance.

.NET. This is the modern way to write Windows programs. You will use this method in this course.

Applications, windows, and messages. These are the three main elements of Windows. An application is a program meant to be used by people (rather than by the operating system or another program). Windows is a multi-tasking operating system, which means that more than one application can be running at once. Each application owns one or more *windows*. Events are initiated by the user or by the operating system, and information about these events must be communicated by the operating system to the applications, and by each application to its windows. This communication is done using *messages*. I will explain these three elements (applications, windows, and messages) in more detail.

The terminology *message* is not used in .NET programming; instead one speaks of *events* and *event handlers*. We will see that this is just a matter of words.

Applications. An application consists of executable code. But in Windows, the executable code need not all be contained in one file. Several types of files can contain executable code, and many files may be used to supply the executable code for a single application. There will be one and only one .exe file, and there may in addition be .dll files (dynamic link libraries) and ActiveX controls (a special kind of dynamic link library). The use of dynamic link libraries permits several applications to share some common executable code. This is absolutely necessary to Windows since the Windows kernel itself is used by all Windows applications. The Windows kernel is supplied in dynamic link

libraries such as user.dll and gdi.dll (gdi = graphics device interface). Further executable code may be in device drivers. (In Windows, your application cannot call device drivers directly; they must be called through Windows.) The code for the .NET libraries that are used to write Windows programs in .NET is supplied in dynamic link libraries, as was the code of MFC.

Windows. You are familiar with the visual appearance of a window on the screen: a rectangular portion of the screen that is used either to present information to the user or collect information from the user, or both. But a window is more than its visual appearance: it is an internal object, or data structure. You can think of a window as an object instantiating a certain class, containing on the order of a hundred data fields. This data structure is not entirely public, so we can only guess at its actual contents. It certainly includes the pixel coordinates of the left and right sides of the window and the top and bottom of the window, and a character string for the title, and some numbers to identify the type of border, the background color, etc. One of the fields is a number (called a *handle*) that identifies the application that owns the window. Each window is owned by a unique application. (For example, the same window cannot be shared by Netscape and Word--it's either a Word window or a Netscape window, not both.) Each window is itself identified by a number (a *window handle*). We can speculate that somewhere in the guts of Windows, there is a master array of pointers to windows, and the handle might be the index of the window in this array. It doesn't matter if this speculation is correct or not, what matters is that each window is uniquely identified by a nonzero number called its *window handle*.

In .NET, which is entirely object-oriented, you will almost never deal directly with window handles, but you should know what the phrase *window handle* means, because you will surely encounter it if you become a Windows programmer.

There is some terminology about the visual appearance of a window that you must learn. The *title bar* is the narrow band at the top of some windows, containing the title of the window. Not every window has a title bar. The *border* of a window is a line (or sometimes a double line) around the window. Not every window has a border. The *client area* of a window is the area of the window that is not in the border or title bar.

There is also an ordering of windows called the *Z-order*. This tells which window is “in front of” which other window. When windows are displayed on the screen, if they occupy some of the same pixels, the one “in front” will get to paint those pixels. The other one will be “obscured.”

Messages and events

In the past, what is now called an *event* was called a *message*. What is now called *firing an event* was called *sending a message*. To understand the basic idea, we’ll use the terminology *message*. A message is a certain small object. As defined in the Windows API, a message has the following fields:

- a time stamp (used only internally by Windows, not used by programmers)
- a *message identifier* (explained below)
- two unsigned longs for message-specific information. These fields are usually called *wParam* and *lParam* in the Windows API documentation.
- The window handle of the window that is destined to receive this message.

The *message identifier* is an unsigned integer which tells what kind of event this message is about. These integers are never written as integers, but instead are referred to by certain predefined constants. There are hundreds of different message identifiers in Windows. I will give several important examples:

- *WM_LBUTTONDOWN*. This message is sent when the left mouse button is depressed.
 - *WM_KEYDOWN*. This message is sent when a key is depressed.
 - *WM_CHAR*. This message follows the *WM_KEYDOWN* message, when the key corresponds to a character with an ASCII code number. Thus function keys and arrow keys cause a *WM_KEYDOWN* but no *WM_CHAR*.
 - *WM_PAINT*. This message is generated by the operating system, when a portion of a window needs its appearance “refreshed”, perhaps because it was resized, or because it was partially obscured by another window which has been moved.
- In .NET, you don’t use these identifiers, but they are still there “under the hood”, so if you want to understand how .NET really works, you will need to understand how Windows itself works.

In each case, the fields *wParam* and *lParam* are used to store additional information. For instance, the extra information for a *WM_LBUTTONDOWN* message is the pixel coordinates of the mouse, relative to the client area of the window. The extra information for a *WM_CHAR* message is the ASCII code of the key that was pressed. Originally, Windows programmers had to learn how the information for each different kind of event is packed into *wParam* and *lParam*. You don’t have to do that.

How events or messages are generated. Let’s suppose the user depresses the left mouse button. The mouse hardware sends some electrical signals down the cable to the computer. These are

intercepted by hardware which generates a “hardware interrupt”. The BIOS, or basic input-output system, stops whatever else it is doing and transfers control to a certain address, where it expects to find an “interrupt handler” for the mouse interrupt. Indeed, if Windows is the operating system running, Windows has installed its own interrupt handler at that address, which must construct a Windows message. This message will have message identifier `WM_LBUTTONDOWN`, and appropriate *wParam* and *lParam* encoding the mouse coordinates, and the correct window handle *hwnd* representing the window which should receive this message (normally the one in which the mouse cursor was visible at the time the mouse button was depressed). What the BIOS supplies is the absolute screen coordinates at which the mouse cursor was located. But Windows can consult its internal table of windows, and the information about the Z-order of the windows, to determine which window had control of the pixel where the mouse button was depressed, and then compute the coordinates in the client area of that window. (If the mouse is depressed in the title bar or border, a different message is constructed--never mind that for now.) This window’s handle will be entered in the *hwnd* field of the message.

The application message queue. Well, what does Windows do with the message so constructed? Answer: each application has an *application message queue*. This is a linked list of messages destined for windows owned by that application. Windows places the newly-constructed message in the correct application message queue. It can compute which application owns the window, because each window is owned by a unique application and the application’s handle is recorded in the window data structure.

The main message loop. What happens to the messages after they are placed in the application message queue? Simple: the application keeps taking them out, in the order they arrived, and sending them to the destination window. This catch phrase

sending a message will be explained below. For now, let's look at how the application gets the messages out of the queue. Each Windows application must have a function called *WinMain*. This function is called when the application is first started by the operating system. It performs some initialization tasks and then enters the *main message loop*, which (slightly simplified) looks like this:

```
while (GetMessage(&msg))
    DispatchMessage(&msg)
```

Here *GetMessage* removes a message from the application message queue, and *DispatchMessage* sends it to the destination window.

This loop executes until there are no more messages in the message queue, and then it terminates. Premature termination is prevented by WM_IDLE messages that the operating system generates when nothing else is happening.

Window procedures. Each window must have an associated *window procedure*, which is a function whose job it is to respond to messages. This function has the form

```
MyWindowProc(hwnd, message_identifier, wParam, lParam)
```

For readability I have not identified the types of the parameters or the return value. But notice that the parameters correspond exactly to the fields of a message. Now we come to a crucial point:

- to *send a message to a window* means to call its window procedure

The call will pass as parameters the fields of the message. Normally a window procedure will look like this:

```

switch(message_identifier)
{ case WM_LBUTTONDOWN:
    /* code to respond to this message */
    break;
  case WM_CHAR:
    /* code to respond to this message */
    break;
  case WM_PAINT:
    /* code to respond to this message */
    break;
    ...
}
...

```

Messages for which there is a case in this procedure are said to be “processed” by the window procedure. Messages which are not processed fall through the switch and at the end there is a call to the *default window procedure*, which is defined in the Windows kernel and provides default processing. There are lots of internally generated messages passing through this loop all the time which are not processed by your code.

What about MFC and .NET? The above discussion applies equally to all Windows programs, however they are written. (Of course if they are written in Basic, then Basic code is used instead of C code.) Now we will discuss what the difference is between programming directly in the Windows API and programming in MFC or .NET.

In the traditional Windows API, you begin by block-copying *WinMain* and a window procedure from some simple program, for example Hello Windows, or the Microsoft-supplied “generic” Windows program. You then edit the window procedure to supply specific code to process the messages for your application.

In MFC, as mentioned above, part of your application consists of `mfc.dll`. This is where *WinMain* is located; MFC supplies *WinMain* and you will not see it in your source code. MFC also supplies window procedures for your windows, so you don't see a window procedure either. Similarly, in a .NET program, there is a *WinMain* somewhere in one of the .NET dynamic link libraries. But you still have to write code to process messages, or "handle events", as it is called in .NET. Where do you put it?

In MFC or .NET you define, or Visual Studio defines for you, a class corresponding to each (kind of) window in your application. The hidden *WinMain* will call methods in these classes to handle different events. For example, there may be an *OnMouseDown* method. In the class corresponding to one of your windows, you can override that method and put your own mouse-handling code.

Why use .NET?

What has been gained over just putting the code into the window procedure under case `WM_LBUTTONDOWN`? First, there are some purely clerical savings:

- You don't have to decode *wParam* and *lParam* yourself. For example, in the case of a mouse message, you get two integers *x* and *y* as parameters to *OnLButtonDown*, instead of having to extract them from the lower two bytes and upper two bytes of *lParam* (via macros `LOWORD` and `HIWORD` supplied in `windows.h`)
- You don't have to perform certain mechanical manipulations. For example, in processing `WM_PAINT` traditionally, you always have to start with *BeginPaint* and finish with *EndPaint*. If you forget the *EndPaint* your program crashes. MFC and .NET both do this for you, invisibly, and you only have to supply the actual painting code.
- You don't have to use block copy and look at all that code that never changes in `WinMain`. Instead that remains invisible and you just click a few times in AppWizard, and then go right to the desired place to add code using Class View in the Workspace window.

In addition to these clerical advantages, .NET offers libraries full of useful object-oriented code—much more extra code than MFC did, and rewritten from the ground up in a coherent, object-oriented way. .NET also offers other advantages, including easy integration of modules written in different languages.