# Windows Programming 101

### or Some Things You Need To Know About Windows Programming

## Where Did Windows Come From?

Way back in the misty dawn of Personal Computing, there were pretty much only two or three personal computing platforms. One (by far the most popular) was Intel hardware on an S-100 bus. The very first Personal Computers were home brew things the details of which were published in Popular Electronics magazine. The first commercially available kit came from the MITS corp. in New Mexico and was called the Altair 8800a, using the S-100 bus. This kit was featured on the January 1975 cover of PE magazine and it marked the birth of the Personal Computer industry.

I got my own Altair in the summer of 75 and spent most of the winter building it for a high school electronics project. When the thing finally ran, I was hooked. Searching for hardware and software was a particularly disheartening exercise in 1976. Plenty of hardware was becoming available, but it was way out of my price range. Software was very expensive too, but there was one thing I wanted more than anything else: a Microsoft BASIC interpreter. I had learned BASIC a couple years before and I was dying to get my hands on a computer to try it out on (remember, in 1976 there weren't many computers around that a 14 year old could just walk up to and use).

Microsoft was a teeny tiny company at the time, struggling to sell just about anything. Stories abound about Bill and Steve in the early days but one thing was certain: when they got a BASIC interpreter running on an Altair, they had bagged their first Big Fish. The years went by and Microsoft grew and grew, and finally Bill made his infamous deal with Seattle Computer Systems and IBM. As the story goes, Gary Killdale (of Digital Research) wasn't home and the IBM execs left in a huff. They had wanted CP/M-86 for their operating system, the successor to DR's extraordinarily popular CP/M-80. Bill told IBM he had an operating system for their new PC (he didn't) and IBM agreed to listen. Bill quickly convinced Seattle Computer to sell him their toy operating system (called QDOS, for Quick and Dirty Operating System). Bill tweaked it here and there and then *leased* it to Big Blue for a nominal charge per copy. That was a stroke of genius.

As IBM PCs sold with PC-DOS, so Microsoft grew with the infusion of cash. Bill also had the smarts to retain the right to sell his practically stolen operating system (he paid a scant $15,000 for QDOS - today that single deal is considered the financial foundation of the wealthiest man on earth) on his own as MS-DOS. Bill was raking in the bucks, but there was trouble afoot. Steve & Steve (of Apple Computer) had not been idle. Their Apple and Apple II computers sold, but not like the IBM PC. Nor did they sell operating systems like Bill did.

In the early 1980s Apple released a totally new kind of machine - the Apple Lisa. This computer was based on experimental machines constructed by Xerox at their Palo Alto Research Center. Yes, the Lisa was a rip-off too! But Steve was headed that direction anyway with some of his later software efforts. These machines were not operated from a command line, like practically every other machine on earth at the time was (though many windowing systems had been created in the years since Xerox demonstrated theirs, they were always add-ons to traditional command-line oriented operating systems), but it was based on graphic images. There were pictures on the screen that represented things, and you used a rolling device called a "mouse" to move a cursor across the screen to point to things. You used a button on the "mouse" to signal that you wanted to do something with that object. You held the button down and moved the mouse to move the objects, and you quickly struck the button twice to let the machine know that you wanted to "launch" the object.

Apple's Lisa was only an understudy in a much bigger plan. Soon Apple released a much more refined version called the Macintosh (as in Macintosh Apples). This machine further refined the "commandless computer". And Bill wasn't asleep either. As Apple passed out these machines to schools everywhere and they started replacing older, more cumbersome typesetting machines in the Graphic Arts, Bill had already seen the same PARC research and wanted something like it too. Windows development began in September 1981. I was stoned at the time and didn't pay any attention to it whatsoever.

Windows was announced in November of 1983. It was pitiful compared to the Lisa. The Macintosh wasn't announced until a year later, but it was clearly superior in every way to Windows. Windows had to contend with myriad different display devices & resolutions, input devices, output devices ... the list went on and on. And it was built on top of MSDOS to boot, which started life as a mere example - a pet project never intended to be used in the real world. Despite these shortcomings, Bill had the marketing power that Apple didn't, and by 1990 (a full seven years later!), Windows version 3.0 was released and Windows was *finally* a fairly usable windowing environment.

Where did Windows come from? It was the bastard child of Seattle Computer Systems and Xerox, developed by Microsoft tinkering with software they didn't invent and driven by Bill Gates' all-consuming passion for cool, Gee Wiz stuff. During the development of the next version, 3 .1, MS was again working with IBM on a completely new operating system. Windows had been cruelly criticized for being nothing more than a cheap windowing environment hastily glued on a hopelessly irrelevant operating system - MS-DOS. This new operating system was to be called OS/2. There were differences between the two companies though, and when Microsoft split they took their concept with them and released Windows NT- New Technology. IBM released OS/2 on their own, and there were some really hateful times in usenet and other discussion forums as OS/2 enthusiasts flamed Windows enthusiasts and they flamed the OS/2 people back. It was a holocaust of flames that still goes on today in various useless guises. There's Windows vs. Mac, Windows vs. OS/2, C vs. Pascal, Java vs. C++, Linux vs. Windows ... all of 'em are pointless.

Windows vs. OS/2 was all really to no purpose though. Windows NT contains an OS/2 subsystem, and OS/2 contains a Windows subsystem. They both began life as almost exactly the same thing. In the beginning the two looked and acted very much alike. Sure, the window dressing is different, but when you get right down to it, under the hood they were pretty much twins.

## Real Windows and Fake Windows

So where does that leave us? There are basically two versions of Windows: that which doesn't require DOS and that which does. Don't be fooled by Windows 95, 98 and ME. They're all the same. They have DOS underfoot. You can't necessarily see it much any more, but it's there none the less. Windows NT is far superior to any DOS version of Windows. Lots of people see me make references to "Real Windows" and "Fake Windows" and this is what I'm talking about. Judging by the names you might think Windows 2000 and Windows XP are different versions. They're not. They are Windows NT, plain and simple. Again, the window dressing is different and some capabilities have been expanded here and some bugs fixed there, but they're all the same basic operating system. Server? Nope, that's NT with different condiments. Home, Professional, Enterprise? Bla, Bla and Bla. All doublespeak for "pay me more and I'll tweak your registry different". In general, the more you pay the more capability & add-ons you get.

In your Windows programming exploits, you'll be far more content working with Windows NT. Mistakes will be made, and they can be disastrous for fake Windows. Not that you'll lose anything (probably), but because you'll be rebooting the system a lot. That doesn't happen with NT (unless you're playing around with drivers anyway). If something gets messed up real bad, it's no big deal. Just kill the process and start over. Sometimes you can get away with that on fake Windows, but not enough that you shouldn't spend the extra money and just get real Windows to begin with.

What about Windows 3.0, 3.1 and 3.11? Well, they are all part of the old guard. I'm not considering them here. They are all relatives of fake Windows. There is also Win32s, an add-on for Windows 3.1 & 3.11. This gets Win32 support for some Win32 calls, but it's still fake. And of course there's Windows CE for palmtop devices and NT Embedded too. Fortunately these are pretty much NT also - though CE does have some really funky quirks. Windows NT is the only Windows worth dealing with. In spite of the fact that it makes I/O much, much more difficult to deal with.

Yeah, what about I/O? Well, here's the beef: Fake windows deals with one really pathetic OS. It always runs on x86 hardware and it's not used in situations where data integrity or security are priorities (or at least they most certainly *shouldn't* be used in these situations). If you want to write a value to a port nobody is going to care, so working with hardware is super easy. Windows NT, on the other hand, was always intended to maintain data integrity and security and most importantly was always intended to run on a multitude of hardware. How do you deal with hardware in this environment? You come up with a machine-independent I/O subsystem. You never let any user code touch hardware directly, and you tie in the security subsystem to make sure that user code has the right to make hardware I/O transactions. In the end, this means that NT is a royal pain in the ass for people who have to work with hardware on a daily basis. Especially when the hardware design changes from time to time.

Under normal circumstances, no user can get anywhere near touching hardware - *any hardware.* This includes any of the hardware that came with the machine as well as any hardware that you're trying to develop for whatever purpose. Normally, when you want to touch any hardware at all you have to go through a kernel-mode driver. So you have to either get somebody else to write a driver or you have to do it yourself. There is one exception to this rule, but in order to take advantage of that exception you have to have some pretty heavy duty privileges and it has to be set up from kernel-mode.

## How Does Windows Work?

The basic concept is pretty simple. The basic Windows "object" is called a window. All windows have various attributes, but one of the most important is it's handle. A "handle" is a unique, 32-bit identifier through which nearly every other aspect of a window can be discovered. Some windows. are meant to be virtual "screens" on which output text or graphics are drawn. This is what most people think of when they think of windowing environments. But nearly everything you can see (and several things you can't see) is a window. A button is a window, as is a list box or a radio button, a tool bar, the buttons on a tool bar, menus, menu items ... *everything is a window.*

Another important window attribute is it's class. A window class is just what it's name implies: a certain class of a window. There are many, many predefined types of windows - or window classes. They are called things like BUTTQN, ListView32, MENU and so on. All of the objects created from a certain window class have basically the same visual aspect and behavior. An EDIT window always looks and behaves a certain way by default. There are three ways to modify how a window looks and behaves. They are: setting style bits, overriding messages and subclassing.

Style bits are another attribute of windows. There are the standard style bits and the extended style bits. Each quantity is 32-bits wide, and in general each bit is used to define some visual or operational standard, such as whether a window has a sizeable border or whether a window processes the Enter key or passes it on to it's parent window. Window classes have styles too. Classes use one 32-bit quantity to define the attributes of a class.

Windows does nearly everything it does by sending messages to various windows. When you press the left mouse key over a button for instance, a click message is sent to the button. In response the button redraws itself, modifying it's appearance to make it look as if it has sunken down into the surface of the screen. See, the brain is easy to fool. If you look at the top and left sides of any object on any Windows machine, you'll see that it appears as if light is falling on that "edge", coming from an unseen source over the top left corner of your screen. Conversely, on the bottom right edge of everything you'll see a darker "shadow" area. This is nothing more than shading differences (it's called chiaroscuro in the art world - the interplay of light and shadow and how it affects the brain) but to the brain it makes it look as if the highlighted objects are sticking up out of the screen. Reverse the shading and it looks as if the object is sunken down.

Incidentally, this may be one of the more subtle reasons Windows became so much more popular than the Macintosh. Up until recently (OS 7 or 8) the Mac OS was "flat". Buttons were flat, and responded to clicks by reversing their color from black on white to white on black (the OS itself was also quite colorless until recently, although drawing, publishing and pre-press applications have always dealt with color). The 3D look gave Windows a more comfortable "feel", in spite of the fact that it took more time to draw all the subtle highlights and shadows. I've used both platforms fairly extensively. Windows always felt better to me, regardless of the fact that I was running Lisa and Mac machines at least four years before I ever laid hands on Windows. When the Mac OS did go 3D, it went in the opposite direction of Windows (predictably, and I think it was a good choice actually). With the release of the Windows version 4 shell (a.k.a. Windows 95), Windows took on a very angular and precise look - like it's virtual twin OS/2. The Mac OS went for a much more "organic" or bulbous look. It's no surprise that given the history of Apple vs. Microsoft, the Windows version 5 shell (a.k.a. Windows XP) took on the "bulbous" look. Personally I don't like it. I much prefer the angular, precise look. Naturally, Windows XP gives you the option to make it look like NT 4 or 2000.

But none of this has anything to do with messages. Windows windows respond to events by responding to messages sent to them by the system, other windows or application code. This is what overriding messages is all about, the second way to modify how a window looks or behaves. One of the attributes of a window is the address of it's Window Proc (typically called a WndProc). A WndProc is a function that processes the messages that are sent to a window. A WndProc takes the form of basically a large If...Then structure (actually, it usually takes the form of a switch ... case block in C, but VEE folks generally have never heard of a switch ... case block. The effect is basically the same).

Basically you test the message for certain values that you want to handle like WM_DRAW or WM_CLICK. When you find one you want to handle differently than the window class defines, you simply do whatever you want with it and return a value to Windows telling it that you handled the message and it doesn't need to. If you want to, you can explicitly hand the message on to another WndProc that implements the behavior you want, or you can explicitly tell Windows to implement the default behavior. If you're the author of the WndProc then you have this option. Many times you are not the author, and you want to "hijak" the WndProc in order to do things with the messages that are being sent to that window. This is Subclassing.

The third way to modify a window's visual aspect or how it behaves is called subclassing. Subclassing is replacing any given window's WndProc with your own, so you have a chance to respond to window messages in a different way than the original WndProc. All windows created from a common class (initially) have a common WndProc. This WndProc provides the default behavior of any given window. A BUTTON for instance. Each and every button created on the screen shares the same exact WndProc unless the button window has been subclassed. And this is more than just "virtually" true, it is absolutely true. There exists one particular block of code that implements the BUTTON class object's WndProc, and only one. No matter how many applications are loaded, no matter how many buttons are on the screen in any particular application, the same exact physical set of instructions are executed every time a button receives a message. Multiple copies of the same code are ***not loaded.*** All applications share the same code. Why do I prattle on so about this? Because it's a very, *very* important point.

How is this possible? If every single button in evey single application shares the same code, wouldn't they all have the same title and wouldn't they all get pressed when you press one? No. They do share the same code (as long as they haven't been subclassed), but they *do not* share the same data. This is one example of what's called "object orientation" in the programming biz. Any given *instance* of an object is composed of code and data that the code operates on. Though all the instances of all the objects of the same type share the same code, the data associated with each instance is entirely separate even if one instances' data is an exact duplicate of another instances' data. The instance identifier for a Windows window is called it's *handle* and now we have come back to the single most important attribute of a window object - it's handle.

For those of you who speak C++, you can think of a window handle as the window's *this* pointer. In Win32 it serves one of the same purposes of C++'s this pointer, or any language's this pointer no matter what it's called: **it is a pointer to this object's instance data.** For those of you who *don't* speak C++ don't worry about it. It's not exactly the same thing. The important point is that all window objects are composed of code and instance data, and the data associated with any given instance is completely separate from any other instance's data. In cases where common window objects have been subclassed, then the code is separate too, or at least some of the code is separate. The purpose of subclassing is to be able to intercept window messages. Well, lets say that the only reason you want to do this is to create a new kind of button with a blue background instead of a grey background. The easiest way to do this would be to simply copy the BUTTON class, replace the window class's hbrBackground member with the color blue instead of grey, register a new class called "BlueButton" or something and create instances of that class. Presto - blue buttons.

But that doesn't illustrate the real power of subclassing - *overriding* (or intercepting) messages. Lets say that you wanted to go about it a different way (there are usually dozens of different ways to get things done). Lets say that you want to override the paint message that causes a window to draw itself. In this case, first you write a WndProc function. In this function, you check to see if the message parameter is WM_PAINT. If it is, you draw the button with a blue background instead of the default gray background. If the message *is not* WM_PAINT then you pass it along to the previous WndProc with CallWindowProc. Here's the beauty of subclassing. You don't have to handle all the messages, just the ones you want to modify.

To subclass the button and make it have a blue background, you call SetWindowLong with the handle of the button, the GWL_WNDPROC index value and a pointer to your WndProc function. This is what subclassing.is: replacing a WndProc. You save the return value because that is the address of the previous WndProc and you'll need it to pass messages you're not interested in on to the previous WndProc using CallWindowProc. Bam. You're done.

Now, there are other mysteries here. Remember I said there's only one copy of WndProc code loaded for any window class? Well, a lot of you have probably seen me babble on endlessly about "process boundaries" and how pointers valid in one process are not valid in another. If you think about that, it seems to contradict the statement that there's one and only one block of code that implements any classes' WndProc. If these buttons are appearing in different processes, then how is it possible that one block of code can be shared between them all given that their WndProc addresses probably aren't even the same?

Well, it's a good thing you noticed that those two statements seem to be in contradiction. It seems a logical impossibility unless you know about *virtual memory.* I'm not talking about disk swapping, I'm talking about real, honest-to-goodness virtual memory - memory that isn't really there but looks as if it is. The BUTTON window classes' WndProc is located in user32.dll, and at any one point in time there is one and only one copy of user32.dll loaded in memory. Regardless of how many processes are loaded. It all has to do with how Windows sets up a process, and the magic of virtual memory is all due to a CPU facility called the *Memory Mapping Unit,* or MMU. The MMU has at it's disposal all of the physical RAM in your computer. Based on directions from Windows, it makes some of that memory appear to be located at selected virtual addresses in any given process, regardless of what that memory's "actual" address is (before it's mapped it doesn't really have any address at all).

Bear in mind that what appears to be happening all at once is actually quantized. When the CPU is executing a given piece of code, that's all that it's doing. It's not simultaneously checking mail or re-justifying a document's text, it is doing only the one particular thing. It will soon move on to doing something else, but nothing - *nothing* - occurs simultaneously. The exception here is multiple CPU machines. These actually *are* doing several things at once. One thing for each CPU. Nevertheless, the MMUs of all the CPUs are synchronized and no CPU steps on another CPU's toes with respect to memory mapping.

So what's really happening is that when process A is executing, the one and only one copy of user32.dll that is loaded in some block of physical memory is made to appear to exist in process A's memory space. When the CPU moves on to execute code in process B, then the MMU is directed to make the one and only one copy of user32.dll to appear to exist in that process's memory space, and so on. Things are simplified further by the fact that the major components of Windows (such as user32.dll) are *always loaded at the same address,* so the MMU doesn't even have to change the apparent address of the memory that's not really even there.

If that sounds like a bunch of doublespeak, don't worry about it. These concepts will be revisited from time to time. They're necessary to understand some things that seem unclear at first - like why are process boundaries so important (because they mark different MMU domains) or how is it possible to have different processes share memory.

## Hungarian Notation

As you peruse Windows programming examples, you'll see variable names such as crBackground, pdwBytesSent, cArray and pti. All of these are examples of a general variable naming practice called *name decoration.* The idea is to give your variable names not only the meaning of "what is this used for?" but also "exactly what type is this variable?”. With experience, you begin to see that crBackground is of type COLORREF, pdwBytesSent is of type LPDWORD, cArray is of type BYTE and pti is of type LPTYPEINFO. Exactly what all that means is not nearly as important as realizing that type defines size, and size is of absolute paramount importance (in programming, *size matters!).*

The particular type of name decoration that these variable names represent is called Hungarian Notation. Legend has it that there was a Hungarian fellow working for Microsoft many years ago, and he strictly adhered to a form of name decoration that he made up explicitly for Windows programming. Since this was a very good idea, it soon caught on and was pretty much unofficial policy at Microsoft. Unfortunately this never caught on with the Kernel group and that's really a shame. Driver examples are far, far more difficult to understand simply because driver writers don't generally use any kind of name decoration.

The system was not overly formalized, and you'll find many different authors using many subtly different types of decoration. There are a few hard and fast rules that almost everybody follows though. Variable names are prefixed by two or (hopefully no more than) three letters that give an indication of that variable's type. The prefix "p" always means "pointer" unless the p is part of the three letter name of the type (if it is, change it. "p" means "pointer"). The prefix "i" or "n" means "integer". The prefix "g" means "global" (here we have a definition of scope). "l" means "long". "sz" means "zero-terminated string". "c" means "char", also known as "BYTE". These are most of the basic types. A "p" prefix before any of these means "pointer", and it's not unusual to see something like "ppIUnknown". This is a pointer to a pointer to an IUnknown interface (don't worry about that right now - it's just an illustration). Prefixing any of these with "g" means that this is a global variable. Most variables are only known about in the function where they are defined. This is called "local scope". A variable defined outside of all functions is said to have "module global scope". If all modules include a file that defines several global variables, then those variables are said to have "application global scope".

One more thing that is useful to mention about Hungarian Notation is that there are sometimes local vars that have no descriptive names whatsoever, like the enigmatic pti above. These are usually two, three or four lower-case letters that are used for local variables that will be used a lot in the text of the function. The only reason they're not descriptively named is because the author didn't want to clutter code with several repetitions of the same word. I often get lazy and simply call pointers "p". I usually skip the explicit "sz" when naming text variables, like "pName" or "pEmailAdrs". This is probably not a good idea, but I do it anyway.

Don't worry about all this size and type stuff too much. It will all be gone over again in "C Programming 101".

## So What Does All This Have To Do With VEE?

Well, VEE runs exclusively on Windows now and sometimes it's necessary to dig into Windows programming to do things you want to do. Understanding how Windows works can help understand some of how VEE works, but it's more of an inter-operational thing. What if you have a customer who, despite your dire warnings, constantly starts up multiple copies of your test system? Chances are this is not a good thing and will mess up your whole test system. One way of preventing this is to check if any other instance of your test system is running, but you can't get that information directly from VEE. You have to dig into some Windows programming to find out.

VEE 7.0 makes the process a lot easier with it's access to the Microsoft .NET library, but that's just an interface - a layer of insulation - over the Windows APL If you understand how Windows works and know something about the API, you truly understand what's going on and that's a good thing. Especially because as wonderful as it is, the .NET library doesn't cover all of the Windows API, nor can it compare to the speed of using the Compiled Function Interface.

VEE's Compiled Function Interface is, by far, the fastest, most powerful way to interact with any part of Windows or any compiled library written in nearly any language. I'm reminded of one time when I lost my temper in 1977. I was using a TRS-80 in a Radio Shack store in Aurora IL - making a Christmas tree program in fact - and was learning the ins and outs of TRS-80 graphics. A couple guys stood by and watched for a while, and started making jokes about how primitive the TRS-80 was and how it would never go anywhere and what a waste of time I was involved in. Well, I was only 15 at the time but I've never been one to take anything lying down, so I got fed up and faced them. "You know, my own computer is an Altair 8800 - *that I built.* Just because you can drive a Caddy with an automatic doesn't mean you have the slightest clue how to operate a Volkswagon with a stick. Put that in your instruction register and decode it."

Honestly, it's not like you'll have to use the CFI or be doomed. .NET makes a lot of Windows programming accessible to everybody. And there are plenty of ActiveX controls that do all kinds of wonderful and amazing things. But sometimes, every once-in-a-while, you need to milk the CPU for everything it's worth and in order to do that you have to call compiled code and the calling mechanism and data transport facilities have to be as fast as possible. ActiveX certainly can't hold a candle to these requirements and even .NET is slower than the CFI and native code. All that .NET stuff runs under the auspices of the CLR (the Common Language Runtime) and has a ton of management code snooping all around and making sure you don't shoot yourself in the foot. It's not nearly as fun as running uninhibited pure x86 with nobody looking over your shoulder telling you what to do.

## Where The Hell Are We?

To sum up, let's go over the most important points: Windows does just about everything by sending messages to window objects. A window object's most important attribute is it's handle. Window objects are based on a window class, which contains various attributes such as the address of the WndProc function, the window's background color, it's menu and icon, as well as other information. Instances of a window class are created and specific instance data is associated with that instance's window handle, such as what kind of border it has, whether or not it has a caption and scroll bars, along with many other visual and behavioral attributes: can it be sized, can child windows be dragged outside of it, so on and so on.

The WndProc function is the entity that processes messages sent to a window and many of the window's behavioral characteristics are controlled by message processing code. Subclassing a window is replacing it's WndProc with another for the purpose of changing the default behavior of a message or giving the window the ability to respond to new messages. All of these points have to do with the windowing environment itself. How it works, how windows interact with each other how they obtain input (from messages), how they display output (by drawing) and so on.

We also saw some of the more "OS" side of Windows: how processes are considered islands unto their own, how Windows uses the CPU hardware to create this illusion and how it can short-circuit that illusion for it's own purposes. There are many, many more things to consider, but it's best to take little bites. Windows began life as a windowing environment. It left all of the OS functions up to DOS. Years later the completely redesigned and improved Windows NT finally kicked DOS out of the equation and it was it's own OS. This is basically why NT is so much better than fake Windows. Real Windows is a protected, secure operating system as well as a windowing environment. Fake windows is not nearly as well "protected" and most certainly is not secure.

**Shawn Fessenden**

**Revised: 11-10-2008**