

Re: About Windows address space

Source: <http://coding.derkeiler.com/Archive/Assembler/alt.lang.asm/2004-04/0670.html>

From: Beth (*BethStone21_at_hotmail.NOSPICEDHAM.com*)

Date: 04/26/04

Date: Mon, 26 Apr 2004 21:04:45 +0100

fa2k wrote:

> *Beth wrote:*

> > *we shouldn't be encouraging slow bloatware "who cares?" solutions too*

> > *much, really, should we? ;)...*

>

> *hehe, I agree:) It's part of the raeson why I like assembly.*

Yeah, and it's the reason why – though, trying to work out a way to "accomodate" the "VB programmers" too, so to speak – I _had to_ mention this at the end ;)

> *OTOH, there are some advantages with using ASCII names, that you did not*

> *know about, because they are specific to the nature of this project.*

Yeah, as I said in that post, it would have been wrong to say "kick out the ASCII names completely" without first knowing the exact application context...the application is always the thing that defines what is "right" and what "makes sense"...so I did try to modify the reply as "prioritise other things" rather than "don't do it!!" ;)...

> *I may*

> *have said that I am making an IRC proxy, but that isn't the main goal of the project.*

You "may" have said this...but I don't recall it...mind you, I'm almost 100% sure that's because _I_ wasn't paying attention at the time, not because you didn't say it...you know, you read something but the brain isn't engaged properly and the words just don't sink in...I either didn't see it or, oops, read straight over it without noticing you mention the context there...ah well, I've seen it this time, anyway :)

> *I am trying to create a really modular program architecture.*

alt.lang.asm: Re: About Windows address space

Well, I thoroughly approve of that kind of thing, as the LuxAsm team know (and are probably still wondering how the stuff I was talking about will actually work...and, rest assured, once my "Linux problem" disappears, I will endeavour to directly answer that question...I know *_how_* it should work – at least "in theory" – but, aarrgh, can't get Linux to even install to be able to start...I, unfortunately, don't know ELF too well that I'd need to "experiment" there that it's not really possible to proceed without confirming some of that stuff, when it *_needs_* "dynamic loading" and stuff...otherwise, I could perhaps do some of the stuff over on Windows for the "time being" ;)...

- > *Modules will be loaded and unloaded at run time, and they will be able to*
- > *interface with each other. How can I take advantage of this in a simple*
- > *proxy? Well, it will also be an IRC client, "bot", and each module will only*
- > *perform a limited function.*

Plus, of course, kind of the whole point of a "highly modular architecture" would exactly be that you can just "plug in" and "unplug" stuff back and forth...that, kind of the point of it all is *_exactly_* that you're "leaving room" for "extensions" to be bolted on and modules to be altered and "improved" and so forth...that is, the basic idea *_is_*, more or less, that you don't 100% know what it'll end up being...in a sense – at least, one of the reasons I was pushing this stuff for LuxAsm – the idea that you're *_encouraging_* others to come along with their "mad ideas" – things you've never even thought of – and just "bolt it on" with a new "module"...it's exactly all about "leaving room" and "generalising" how it works so that all sorts of "mad ideas" can just be easily "bolted on" and "unbolted" as and when you feel like :)...

Hence, contrary to what's "usual", a touch of *_vagueness_* and "I have no idea what exactly it will eventually be used for" is actually perfectly acceptable here...nothing wrong with admitting you don't exactly know how the future will pan out...no-one knows this, however much they pretend...being honest and fore-sighted enough to work that into your program "architecture" that you don't "doom" your project because things pan out in a way you didn't initially expect is *_wise_*...

Indeed, one of the *_wisest_* and *_most honest_* expressions in the English language is often "I don't know"...it would actually be much better if people said it more often, not less so...oh yeah, I'm as equally guilty of this as anyone, of course...but, sometimes, I do (rarely) get it right on occasion ;)...

- > *The ASCII is used as names of functions, that are registred in a "core" DLL.*
- > *The DLL is the only one that is called directly. The registered*

functions

> will resemble "events" (as in event driven apps, I think), but they are a

> bit different, as every module can call a "call point", at which multiple

> functions can be registered. Luckily, I'm not entirely finished with

> designing the interface, so I cannot bore you with more details:)

No, I get you, anyway...a good "highly modular" architecture does basically consist of two main "parts"...a kind of "API" by which modules ask to get things done...and then a bunch of "callbacks" (or "events" or you could think of it as the "mirror image API for asking for things to be done in the other direction" or whatever :) inside the module...all of this being "standardised" so that modules can happily be "plugged in" and just "linked up"...

With the way I was envisaging it for LuxAsm when I suggested it there, I was going for a "main module" with an "API" that modules can use (the "main module" is, basically, the main GUI IDE interface so that the API are things along the rough lines of "AddMenuItem", "AddToolBarButton" and that kind of thing :)...then modules would receive "events" in return that they respond to...each module, by the way, has an "initialisation" and "clean up" function too (a "WEP" function for all those who are ancient enough to remember Win3.x DLL coding ;)...)

The basic idea being (in case this has any useful ideas in it for you): "Main module" looks through "plug-ins" folder...discovers a "module"...loads "module" file...calls "initialise" function..."initialise" function calls API such as "CreateMenu", "AddMenuItem", "AddToolBarButton", etc. to add its "components" onto the main GUI...control returns to "main module"...when user clicks on the toolbar button added earlier, the "main module" has the supplied information ("button ID" and so forth ;) from the "module" and generates an "event" for it to deal with...the "module" then receives the "event" and enacts whatever actions that this button is supposed to generate...if there's some need for "output", the "module" can call, say, "DisplayMessageBox" or "AddMessageToOutputWindow" API or whatever...and, basically, repeat until "main module" closes (or until "module" unloaded, if there's some option to add and remove them whilst the program is still running :)...during application close, the "module" is called once more with its "clean up" function...this is basically a "last chance" point to de-allocate memory, save "settings" to a file or whatever is needed...

The tricky thing here is defining the "API" and "events", to be honest...got to be "flexible" enough to handle changing requirements as things develop...would also be nice in the "let's not encourage slow bloatware" perspective to make sure the whole thing is properly optimised that our "highly modular" stuff isn't greatly costly (to see

an example where it is...Windows itself works in exactly this kind of way – any "event driven" GUI does, basically – without considering "optimal" at all, it would seem...and what a pile of crap it all is too! ;)...

Anyway, now I've heard your "application", you know, I'm still not convinced about the "ASCII strings"...

> *The ASCII is good because it provides a general way of distinguishing the functions from each other (using two strings, one for specialization and one for generalization).*

Two strings? What's going on there exactly?

Anyway, ASCII still isn't necessary...as long as you're careful, then you could also take a look at the "import / export by ordinal" methods...with Windows, at least, it's possible to assign an `_ordinal_` to functions and "import / export" using those instead of ASCII strings...these not only improve your application's performance, by the way, Windows itself deals with these quicker (for exactly the same reasons why your application would benefit from it :)...

The sole thing to be careful of is where you'd be changing the "events" around a lot...that is, to be "backwards compatible" (if this has an effect on your application...it certainly does on Windows itself, which is why Windows insists on ASCII strings rather than any "ordinals", just to be absolutely "safe" and "sure" :), you don't want an "event" to be "function #36" in one version but the same "event" is "function #24" in the next version...this would, as I'm sure you can see, introduce a nasty "compatibility" problem...this can be avoided by using a "once an ordinal is assigned, it `_sticks_`" policy...once "onOpen" is "function #3" then you define that in your header file and, simply, it never changes (Windows, again, provides another example of this method too...if you look at the "WM_" window message identifiers, they are simply integer constants...and they simply `_don't change_`...same integers they were in all the Windows versions beforehand :)...none of this should present any great problems unless you decide to "obselete" a bunch of "events" in future...then, of course, you can't really "re-use" the older ordinals because they actually still mean things to older versions of your program...hence, if there's a lot of this going on, then "gaps" can develop where you have "obselete" events...mind you, if you're really being this "backwards compatible" then those aren't really "gaps" because they still contain the older events and methods...they are just now considered "obselete" because something else has taken their place...

Another way to look at this is why not "standardise" the ASCII "event names"? How about a four digit "code" like "0000", "0001", "0002", "0003"? And, if you're going to do that, then four digits? Oh, so,

really, it's just a dword, anyway...might as well get more "space" out of this by not using ASCII digits but using actually dword values...then, oh, you're just using "ordinals" without thinking, anyway...and, well, define some "header file" where – exactly like Windows does defining all the "WM_" constants as their respective integer values – you define: "MyFunctionByHandle equ 00000000h", "MyOtherFunction equ 00000001h" and so forth...then, in your source code, you can use nice, friendly "symbolic" names for the events rather than cumbersome "ordinal" ID codes...if the "ordinal" is 32-bits then, trust me, you're never going to come up with over 4 billion or so "events" – even after hundreds and hundreds of "revisions" of your program – to ever overflow that...but 32-bits can be dealt with in a `_single machine instruction_` with no fuss or bother...the "user" includes the "header file" for your library (probably a thing you have to do, anyway, right? :) that just automatically defines the "events" as various integer values – "ordinals" – that all your users (exactly like Windows users all just put "WM_PAINT" or "CS_HREDRAW" without thinking or knowing what these values actually are :) can use nice, friendly symbolic names in their source code...but the machine gets what it deals best with – simple machine integers – and doesn't have to play around with ASCII strings everywhere...

With Windows, the mechanisms are already in place (Microsoft even point out that "by ordinal" is better performing and preferable when other conditions allow it to be done :)...Microsoft just don't use the "ordinals" themselves on their "system DLLs" because, well, they are lazy first...but, second, they want to ensure everything is "safe" because they have a control-freak, patronising attitude that no programmer understands what the hell they doing so `_enforce_` things everywhere...they "enforce" the DLL functions by name as a kind of "parameter check" thing, so to speak, to make sure that programmer really is calling the function they think they're calling...

Also, here's some other ideas...each "module" has a function called "GetPointers"...this function returns a table of addresses – a bit like an OOP "virtual method table" – to its "events"...note a clever part of this idea is that you can confirm if this is a `_valid_` "module" for your program – and not just some random DLL file that a mischievous user has dumped into the "plug-in" folder to see if it would blow up your program – by the presence of absence of the "GetPointers" function...only a valid module has the function and returns the right kind of "table of function addresses" in return...if you want to change this later on in some way, you can again look to something Windows does as an example...want to add on a parameter to "CreateWindow"? Well, Microsoft simply create a "CreateWindowEx" API for that...support both and you've got "backwards compatibility" between different versions...or, another Windows trick you can look at, is to pass a structure filled out with "parameters" and the first parameter in that structure is a "size of structure" value...ah-ha! You can work out what "version" of the structure you've got from its

alt.lang.asm: Re: About Windows address space

size...as parameters are added in later version, the size changes and this tells you what "version" of the API you want...

Although, this is slightly academic from the perspective that this "GetPointers" is one of those things that's so "generic", it'll probably never need to ever be changed in any way...you call it and it returns a "table of addresses" to each of the "events"...in your header file, you again simply define a bunch of equates – "FunctionOne equ 00000000h", "FunctionTwo equ 00000001h", etc. – which are simply the indexes into the "table of addresses" to discover the address you should be calling in the DLL...

Note one very interesting thing about this method...only one function needs to be "linked" by the OS – the "GetPointers" – thereafter, the module is doing its own linking...calling any "event" is an indirect CALL via the "table of addresses" using the "function number" that's defined in the header file...one machine instruction ("CALL [offset Table + (Index * 4)]" :)...avoids a whole bunch of nonsense Windows can introduce like CALLs to JMPs, tons of long ASCII string entries in some "import" and "export" table bloating the "PE headers", etc....

Remember, the most performance gain is algorithmical – "the fastest code is the code that never runs" – than they how cleverly you write a particular algorithm in instructions...or, if you prefer a more Obi Wan–style "Eastern Philosophy", then the best warrior carefully chooses their fight and resists actually "fighting" altogether, when possible...and it should all be approached "top down" in that regard...start by thinking: "do we need ANY of this at all in any way?" before "It might sound clever but, sincerely, does it really need to be highly modular?" (oh yes, I fully support the idea in theory...BUT, always, it's the application defines what you should do and what makes sense...if the application is kind of saying "no!" to you, then, don't worry, you can always implement your "mad idea" elsewhere in another application when it does make the most sense...same goes for OOP, "portability", "backwards compatibility" and all the other usual suspects...if there's no good reason – why make "Sonic Heroes" on a PC "backwards compatible" with the original "Sonic" on a NES? It gains you absolutely nothing to do that...it simply makes no logical sense that I'm not even completely sure myself what the hell I'm referring to, anyway! – then just don't do it! There's always tomorrow...other battles to be fought...hey, you don't want to use up all your best ideas in one application and have nothing "clever" you can think up for your other programs, anyway, right? ;)...start from the top and work your way down and, as "Electronic Arts" has now made their slogan, "Challenge Everything!"...do we need an "OK" button here? Does the program have to do things sequentially following what's seen on the screen? What on Earth is that "progress bar" over there actually telling us, anyway? If it serves no useful purpose – especially if it would take all your time to implement – then don't do it...BUT, if it does serve a useful purpose to your application – and you reckon you could squeeze it into

your time schedule – then don't you dare not do it!! Enough of the "Yodas" for the moment, you've got my point, I think...you are writing applications (or application libraries) for users...hence, users must want it...and it should always "do what it says on the side of the tin"...and, also, if it's not written on the side of the tin, the user should not suddenly be surprised to find that it "contains nuts" when nothing about nuts was mentioned on the side of the tin (if someone allergic to nuts, then, no, it's not always good to be getting "more for free" without being told first ;)...can't manage this? Then stop right now and save wasting your time...BUT, indeed, we know you can manage it when you put your mind to it, so, rather, put your mind to it, and "go kick arse!"...

[Oh, thinking about it, also note that it's trivial to "rename" a constant in your header file to change, say, "WM_DRAW" to "WM_PAINT" (and, in fact, you can retain `_both_` by simply equating the two to exactly the same integer value that they are "convenient synonyms" :)...but, now, if you felt that you wanted to change the name of those "ASCII names"? Oh dear...not going to be easy `_and_` "retain compatibility"...the ASCII strings are "easier to type and read" but this interestingly makes it `_harder_` for "later modification"...]

Take a look at Windows' "COM" ("component object model")...no, don't worry yourself about the "OLE" nonsense...just the basic "COM" "binary standard" framework that runs the show at the lowest level...not a single "ASCII string" in sight...in fact, seems to be very much like the "GetPointers" / "table of addresses" idea...more interestingly, "ASCII strings" are `_RESISTED_` for the "interfaces", exactly because COM is system-wide and it would be real easy to end up with a "coincidental" clashing name...so, rather, each interface is referred to by a "GUID" ("Globally Unique Identifier"...a 128-bit integer `_designed_` to be "always guaranteed unique" that a little Microsoft utility can automatically spit out a whole bunch of them for you and you `_know_` that no other application or anything else shares the "GUIDs" you've just gotten yourself ;)...

Now, I'm not saying "use COM"...if you're not OOP and "objects", then that would be completely silly to do...but you can `_learn_` from it and "borrow" the stuff that `_is_` useful...in this context, you could be something similar to having one DLL function only, like "DirectDrawCreate", that just provides back a "table of addresses" from which you can call the various functions...if your functions aren't OOP then – after considering whether it might make sense to make them so and deciding, perhaps, "no, don't think so!" – simply make them ordinary functions...none of that "'this' passed as first parameter" nonsense (they are all "static functions" in the typical C++ jargon ;)...

Indeed, as you were the one saying "don't want to re-invent the wheel", any particular reason why you have considered the "wheel" of Microsoft's "COM", which was kind of designed for the whole "modular"

idea? Ah, might be too slow, complex, cumbersome and useless for what you want to do, indeed...but "don't reinvent the wheel" _actually means_ (or actually _should_ mean): "check if someone has made a 'wheel' already that you can simply make use of _before_ you consider inventing any of your own wheels" (and, I'd say, if their "wheel" is inappropriate or you can't find any actually appropriate matches, then, please, don't hesitate to create a whole bunch of wheels...after all, how's this thing ever going to get anywhere if we ain't got ourselves any "wheels", eh? ;)

- > *I understood your posts correctly, ideally, the*
- > *GetHandleByName function would only be called once, then the handle would be*
- > *used subsequently. I agree with this completely. Only do the ASCII stuff*
- > *once, then use a fast "handle" or index.*

Indeed; But go further and consider all possibilities as to whether any "ASCII" is needed at all...it could be wiped out completely, as I've suggested...then you can get yourself better performance with less code and less effort, without complicating the "ease of use" for your users at all (what's the difference between passing "MyFunctionByHandle" as a string parameter to passing MyFunctionByHandle as a symbolic constant? Yeah, a case of "with or without quotation marks" _only_ in most cases ;)...oh, and you don't have to mess around working out which is the "best hash" or otherwise speed up algorithms or whatever...never forget that "the fastest code is the code that never runs"..."clever optimised code" only ranks _SECOND_ here, not first...

Mind you, all a matter of opinion how to go about things like this, I suppose...you could, of course, think I'm talking crap or something...a reasonable enough opinion to have, as I usually am ;)...

Beth :)