The Old New Thing

Wait, but why can I GetProcAddress for IsDialogMessage?

3 Jan 2007 10:00 AM | **21**

Okay, so I explained that a lot of so-called functions are really redirecting macros, function-like macros, intrinsic functions, and inline functions, and consequently, `GetProcAddress` won't actually get anything since the function doesn't exist in the form of an exported function. But why, then, can you `GetProcAddress` for `IsDialogMessage`?

Let's take a closer look at the exports from `user32.dll`. Here's the relevant excerpt.

```
417  1A0 0002C661 IsDialogMessage
418  1A1 0002C661 IsDialogMessageA
419  1A2 0001DFBC IsDialogMessageW
```

Notice that this function is exported **three** ways. The last two are the ones you expect, `IsDialogMessageA` for ANSI callers and `IsDialogMessageW` for UNICODE callers. That first one is the one you didn't expect: `IsDialogMessage` with no A or W suffix. But notice that its entry point address is identical to that of `IsDialogMessageA`. The `IsDialogMessage` entry point is just an alias for `IsDialogMessageA`.

This phantom third function is hidden from C and C++ programs because any attempt to call `IsDialogMessage` gets converted to `IsDialogMessageA` or `IsDialogMessageW` due to the redirection macro:

```
#ifdef UNICODE
#define IsDialogMessage  IsDialogMessageW
#else
#define IsDialogMessage  IsDialogMessageA
#endif // !UNICODE
```

(Of course, you can play fancy games to remove the redirection macros; I'm just talking about the non-fancy case.) If nobody can call the function, then why does it exist?

Because of mistakes made long ago.

If you hunt through `user32.dll` you'll find a few other functions that follow a similar pattern of having three versions, an A version, a W version, and a phantom undecorated version (which is an alias for the A version). At one point long ago, the function existed only in an undecorated version. This turned out to have been a mistake, since there was a character set dependency in the parameters (perhaps obvious, perhaps subtle). The mistake was corrected by splitting the function into the A and W versions you see today, but in order to maintain compatibility with older programs that were written before the mistake was recognized, the original undecorated function was left in the export table.

When you don't have a time machine, you have to live with your mistakes.

In a sense, these functions are vestigial organs of Win32.

Postscript: Unfortunately, like your appendix, which can get infected, these vestigial organs can create a different sort of infection: If you are using p/invoke to call these functions and mistakenly override the default name declaration with `ExactSpelling=true`, like so:

```
[DllImport("user32.dll", ExactSpelling=true)]
```

```
public static extern
bool IsDialogMessage(IntPtr hWndDlg,
                     [In] ref MSG msg);
```

then you will in fact get the normally-inaccessible undecorated name, since you specified that you wanted the exact spelling. This highlights once again that you need to be alert when doing interop programming: You get what you ask for, which might not be what you actually wanted.

Blog - Comment List MSDN TechNet

Comments

Andy J
3 Jan 2007 11:44 AM
#

Freshly shipped from America, it took no time at all to arrive :)

I love your writing style and the insights you provide into the development of windows is a fantastic inspiration.

Many thanks for a great read!

JS
3 Jan 2007 12:21 PM
#

Just out of curiosity, back in the bronze age when the decision was made to make *W and *A versions of every API that may (even subtly) depend on character set--did anyone suggest encoding Unicode with an 8-byte code unit instead? I realize UTF-8 didn't yet exist while Windows NT was being developed, but I wondered if Microsoft came close to inventing it themselves. I often thing it'd have been simpler than duplicating half the APIs. (I _still_ think it'd be nice to be able to make UTF-8 the "ANSI" code page character set, because then standard library functions that accept filenames as char * would be able to deal with Unicode filenames on Windows.)

OTOH, perhaps MS chose against creating an 8-bit Unicode transformation format to avoid the computational overhead, preferring the simplicity of one 16-bit code unit = one character. Then the Unicode standard changed and broke that assumption anyway...

[*I fail to see how an 8-bit Unicode encoding would have avoided the A/W duplication. the A version would be CP_ACP and the W version would be the imaginary 8-bit-Unicode version. You still have two versions of the same function. -Raymond*]

Mihai
3 Jan 2007 12:41 PM
#

<<the A version would be CP_ACP and the W version would be the imaginary 8-bit-

Unicode version>>

Nope. The proposal was to have UTF-8 as ANSI code page, so the old functions would be enough. This is the way the UNIX/Linux world "adopted" Unicode.

And here are some thoughts about this:
http://blogs.msdn.com/michkap/archive/2006/07/14/665714.aspx

> [*So what happens to the old code pages? I suspect people in Europe and Asia would be upset that all of their files suddenly became inaccessible. (How did UNIX/Linux handle Japanese file names prior to the adoption of Unicode?) -Raymond*]

Resource Manager
3 Jan 2007 1:42 PM
#

We're waiting for your 2006 prediction and if it came to be correct

http://blogs.msdn.com/oldnewthing/archive/2006/05/23/604743.aspx

> [*More proof that people can't read. Read the prediction again. All the information you need is there. -Raymond*]

J
3 Jan 2007 2:19 PM
#

Resource Manager:

Raymond buried his prediction in his recent post of random links, so many people probably missed it:

http://blogs.msdn.com/oldnewthing/archive/2006/12/29/1379914.aspx

"Jim Allchin will retire on November 13, 2006."

> [*I said I would reveal it at the end of the year, and it's in the last posting of the year, and there's even a link to the reveal at the end of the original posting. I don't know what more people need... -Raymond*]

Mihai
3 Jan 2007 2:51 PM
#

<<So what happens to the old code pages? I suspect people in Europe and Asia would be upset that all of their files suddenly became inaccessible.>>

Yes, there are a lot of backward-compatibility issues to solve. I am not advocating UTF-8 as ANSI, I was just explaining the idea.

In theory most of the issues can be solved, with a lot of work, if this would be a solution

to consider.

For instance: add an "utf-8 bit" to lcid, so you will have Japanese.932 and Japanese.UTF-8, etc. For text files, assume 932 if no bom is there.

I know, sounds simple, and I know it is not. Unix has a lot of problems from this, and I think it was a lazy solution, that is painfull in the long run.

<<How did UNIX/Linux handle Japanese file names prior to the adoption of Unicode?>>

Badly :-)

The file system is encoding-agnostic, the file names are "just a bunch of bytes"

The result is (even now) a mess:

1. set LANG=ja-JP.Shift-JIS
2. create a file with Japanese name
3. set LANG=ja-JP.EUC-JP
4. list the files, you see junk
5. set LANG=ja-JP.UTF-8
6. list the files, you see other junk

This "agnostic" approach in the kernel, OS, libraries, etc. make me consider the approach "lazy." It does not solve much. The UTF-8 strings moves around without breaking the old C API (ok, that is the UTF-8 idea), but in many cases the functionality is crap.

I was just explaining what the idea of "UTF-8 as ANSI" wants. But I think it is flawed. Is easy to make UTf-8 work (easier than UTF-16), but it is more difficult than UTF-16 to make it *work right*

So, I think UTF-16 was the right thing to do, although painful for some.

The best thing (if you ask me), would have been creating and releasing MSLU way earlier.

When it was released, W95 was not officially supported anymore by MS, and W98 was close to the end of it's life cycle.

So, release MSLU same time with Win 2000, make it clear for all developers "this is the way, the strong advice is to go with MSLU" and now we would have been in a better shape.

Well, too late for this :-)

BryanK
3 Jan 2007 3:07 PM
#

Well, filenames in Linux aren't exactly "just a bunch of bytes". They're really "just a bunch of bytes, except not the zero byte, and not the byte whose ASCII encoding is forward-slash".

(This is vaguely similar to Windows' "bunch of UTF-16 code points, except not the code point for forward slash, backslash, :, *, ?, or redirection symbols (<, >, |). It just has fewer restrictions, because the POSIX shells had better support for escaping their special characters, especially back in the days of win9x's command.com. Though I'm not sure how you could escape a colon in $PATH in bash... hmm.)

Mihai
3 Jan 2007 3:35 PM
#

<<Well, filenames in Linux aren't exactly "just a bunch of bytes". They're really "just a bunch of bytes, except not the zero byte, and not the byte whose ASCII encoding is forward-slash".>>

I know, but that was the simplified version :-)

There are full books on the UNIX file systems/NTFS :-)

Obviously, there are restrictions. And it is difficult to find out at what level the restriction is introduced (same as Windows).

It can be data structures stored on disk, or file system driver, high-level API, shell, or even network a protocol sharing the file system (SMB/NFS).

For instance the 256 MAX_PATH in Windows is not NTFS restriction, is API restriction (with a known work-around), the case-insensitive behavior is also not NTFS restriction, is the Win32 layer (you can enable the POSIX sub-system and get case-sensitive on top of NTFS). Even you mention escaping, but that is a shell thing, not a file-system one.

Anyway, ignoring the details, for Linux the final result (for the user) is bad: I create a file with a locale set in a certain way, then I set the locale differently and I see crap.

This is especially bad for multi-user systems sharing the same file system, but with different locale settings.

Dewi Morgan
3 Jan 2007 4:50 PM
#

On any system where I create a file that I don't have the system configured to display correctly, I'll see nonsense. Some kind of display hints like BOMs in filenames might've been a cool idea, so you could specify the filename using the "right" letters in any codepage/encoding/directionality. But that would've been a compatibility nightmare.

Would be cool to be able to create or refer to a japanese filename as "[romanji]konnichiwa[ascii].txt", though, rather than having to load up the Japanese IME. Wonder if that's doable with a thirdparty app.

[*If all you want to do is enter Japanese characters without installing a Japanese IME, you can write a web page that contains the characters you want and copy/paste them. I do that sometimes. But whether you can enter them or not, the file name will still display okay (assuming you have the necessary fonts installed and the program you're using is Unicode-enabled). -Raymond*]

Mihai
3 Jan 2007 5:31 PM
#

<<On any system where I create a file that I don't have the system configured to display correctly, I'll see nonsense>>

True. But on Windows you get squares for each Japanese character, because you don't have the font. But a Unicode application can access it, not problem.

But on Linux, if you set the locales to use Shift-JIS, create a file using characters with the second byte 5C (backslash), like for instance Katakana So (U+30BD, 83 5C as Shift-JIS). Then set the locales to UTF-8 and you get an invalid UTF-8 string. Or set the locale to en_US.8895-1 and you get the second byte backslash, invalid in a file name.

There is a big difference between "cannot display" (win) and "cannot use" (linux).

And we have display problems in Win if support is not installed, while on Linux we have functionality problem, even with support installed (setting the locale in the environment does not install/uninstall support for a certain language).

Stu
3 Jan 2007 9:31 PM
#

It's perfectly possible to have a Linux file name with backslash (0x5C) in it, its forward slash (0x2F) that's not allowed because its the path seperator.

A quick test follows:

$ touch testfile\\

$ ls test*

testfile\

$

As far as I can tell, 0x2F is invalid in all filenames, regardless of locale. Does anyone have a counter-example?

Norman Diamond
3 Jan 2007 9:42 PM
#

> So what happens to the old code pages? I

> suspect people in Europe and Asia would be

> upset that all of their files suddenly became

> inaccessible.

That's true. I wish that Unicode had been designed with the same disregard for ASCII as for other character sets, so that no one would be misled into thinking that certain compatibilities exist or that everyone can easily convert existing files.

> (How did UNIX/Linux handle Japanese file

> names prior to the adoption of Unicode?)

The same way as 8.3 filenames are assigned in current Windows systems, the same way as MS-DOS handled Japanese, etc. Most narrow characters were single bytes, most[*] wide characters were encoded as 256 * highbyte + lowbyte, strings of bytes were strings of bytes, editors had to take care not to put a linebreak in the middle of a wide character, etc. Most commercial Unix systems used EUC but some used Shift-JIS. Linux supports both but most distros defaulted to EUC out of the box. I think the default is more commonly UTF-8 now, but I haven't had enough time to play with it in recent years.

[* In EUC, half-width katakana were also encoded as 256 * highbyte + lowbyte even though they were displayed as narrow characters.]

Wednesday, January 03, 2007 3:35 PM by Mihai

> Anyway, ignoring the details, for Linux the

> final result (for the user) is bad: I create

> a file with a locale set in a certain way,

> then I set the locale differently and I see

> crap.

That is true. It is ALSO true in Windows, just not 100% of the time in Windows. With Windows 8.3 filenames it's 100%; with Windows long filenames it "should" be 0% but it isn't.

Last week on a US Windows XP SP2 computer I downloaded a driver from a Japanese vendor, unpacked it, and got garbage filenames. I had to go into Control Panel, set the system default code page to Japanese, reboot, unpack the file, go into Control Panel, set the system default code page to 1252, reboot, and install the unpacked driver. It worked. Some situations work properly without this nonsense, some don't work even with this nonsense.

Wednesday, January 03, 2007 5:31 PM by Mihai

> But on Linux, if you set the locales to use

> Shift-JIS, create a file using characters

> with the second byte 5C (backslash), like

> for instance Katakana So (U+30BD, 83 5C

> as Shift-JIS). [...] Or set the locale to

> en_US.8895-1 and you get the second byte

> backslash, invalid in a file name.

100% true. In fact you don't need Linux in order to observe it.

[*Windows XP is fully Unicode (or should be). If a Japanese vendor's driver installer doesn't use Unicode, well that's the vendor's fault isn't it? Or do you mean "in the world of Windows-based software" when you say "in Windows"? -Raymond*]

Stu
3 Jan 2007 11:31 PM
#

"Or do you mean "in the world of Windows-based software" when you say "in Windows"? -Raymond"

Well of course he does. Just like when people say "in Linux" they mean "in an OS distribution based on the Linux kernel and GNU userspace, plus whatever else the distributor included/I installed", or when people say "on my Mac" they mean "On my Apple-branded hardware, running some form of the Mac OS (probably X) and compatible applications".

"If a Japanese vendor's driver installer doesn't use Unicode, well that's the vendor's fault isn't it?"

Not necissarily the drivers may have been in a .zip file- according to http://blogs.msdn.com/michkap/archive/2005/05/10/416181.aspx .zip archives do not support unicode filenames, something the vendor may have been unaware of (since it "just works", with no indication that it might fail in other locales).

Since Windows includes simple .zip file support, no third party software is responsible.

(The choice of .zip as the archive format is the problem, so it's not MS either)

Trey Van Riper
4 Jan 2007 8:18 AM
#

Heh... you say 'Well of course he does,' but you perhaps forget that much of the content here involves details concerning the Windows operating system. I think Raymond's request for clarification makes sense, given the context of his blog.

John Elliott
4 Jan 2007 10:07 AM
#

How about another set of entry points for functions with variant US/UK spelling: IsDialogueMessage, GetSysColour and so on?

[*Why stop at UK English? Why not also add entry points for French, German, Spanish... -Raymond*]

John Elliott

4 Jan 2007 11:27 AM
#

I wasn't being serious. If I had been, I'd have suggested doing the variant functions with the preprocessor, like SDL does (#define SDL_Colour SDL_Color).

Norman Diamond
4 Jan 2007 11:14 PM
#

> Windows XP is fully Unicode (or should be).

Each of the following examples is pretty small, but enough to show that it's not fully there.

Last I saw, gethostname lacks a Unicode version in Windows XP just as much as it lacks a Unicode version in Windows CE.

I didn't test whether Windows XP's StringCchPrintf(... %S ...) could convert ANSI strings to Unicode where Windows CE's version couldn't.

Though ActiveSync isn't part of Windows XP, I think you can guess what company it comes from. One of your colleagues suggested running ActiveSync under AppLocale, which I intend to try when I have time.

Though Visual Studio 2005 and some tools that come with it aren't part of Windows XP ....

Using Windows Explorer on Windows 2000 to copy some files from a share on a Windows 98 machine garbaged the filenames, but I haven't experimented to see if Windows XP would get it right.

> If a Japanese vendor's driver installer

> doesn't use Unicode, well that's the vendor's

> fault isn't it?

Partly I agree. Just now I figured out that I can try running the installer (or unpacker) on a Windows 98 machine and if it works then the vendor neglected to use Unicode.

[*Okay, perhaps I should have said "fully Unicode (where possible)" - host names are ASCII by definition (see RFC 952 and RFC 1123). Setting StringCchPrintf aside, the other stuff you mentioned aren't Windows XP and therefore are irrelevant. I suspect you brought them up just to be annoying. Yet somehow I do not find this surprising. - Raymond*]

Dean Harding
5 Jan 2007 12:12 AM
#

> Last I saw, gethostname lacks a Unicode version in Windows XP

That's because gethostname returns the DNS name of the computer, and DNS is ASCII-

only (well, there are some extensions to the standard to allow "international" names, but they're more a convention than an actual standard).

> Last week on a US Windows XP SP2 computer I downloaded a

> driver from a Japanese vendor, unpacked it, and got garbage filenames

My wife is Korean so I see lots of non-Unicode apps written assuming a Korean codepage and thus creating garbage file names.

I've gotten good at cut'n'pasting the names into a text file, opening them up in Firefox, forcing the "Encoding" option to Korean and cut'n'pasting the names back. Works like a charm for single files, though I imagine rather tedious for lots of files!

I also see lots of programs display "???????.txt" as the file name for files with Korean names (which are non-Unicode apps written assuming an English codepage, of course).

Cheong
5 Jan 2007 1:23 AM
#

[quote]As far as I can tell, 0x2F is invalid in all filenames, regardless of locale. Does anyone have a counter-example?[/quote]

I can confirm this.

When I use "RAR for Linux" to extract those files in ext3fs, they always fail, while I can successfully extract them in Windows. So I guess it's caused by the restriction you mentioned.(Although I didn't exactly checked what those names look like...)

Norman Diamond
5 Jan 2007 1:56 AM
#

> host names are ASCII by definition (see RFC

> 952 and RFC 1123)

Hmm. Any idea why Windows Mobile 5 didn't display a warning when I used its settings screens to set its machine name? I seem to recall ordinary Windows systems also not giving warnings when their computer names are set to the same as the persons' names who usually use them.

> Setting StringCchPrintf aside, the other

> stuff you mentioned aren't Windows XP

As far as I can tell by now, the problem which one of your colleagues and I thought was in ActiveSync is actually a problem in Windows Explorer instead.

>>> Or do you mean "in the world of Windows-

>>> based software" when you say "in Windows"?

Or is it one country of Windows-based software? It still includes the world's most

profitable maker of Windows software. I understand that most divisions of your employer don't take your advice, but your customers still get pounded by them.

>>>>> How did UNIX/Linux handle Japanese file

>>>>> names prior to the adoption of Unicode?

Linux can handle Japanese filenames by whatever method you wish. You don't have to use other people's file system drivers. Or did you mean the world of Linux-based software?

Well, look. I don't really enjoy this tit-for-tat. Windows and Linux and everything else are still worse than they should be. Windows is not an exception to this, OK?

Friday, January 05, 2007 12:12 AM by Dean Harding

> That's because gethostname returns the DNS

> name of the computer

Hmm. I knew Pocket PCs were DHCP servers in a very limited context, but I didn't know they were DNS servers. No wait, how come their network settings screens allow setting the IP4 addresses of external name servers the same way ordinary Windows systems do?

[*"I don't really enjoy this tit-for-tat." At last we agree on something. -Raymond*]

Norman Diamond
5 Jan 2007 5:05 AM
#

>> "I don't really enjoy this tit-for-tat."

> At last we agree on something.

Yes. I'm glad to see your answer on that.

>> Windows and Linux and everything else are

>> still worse than they should be. Windows is

>> not an exception to this, OK?

>

>

Of course you were not (are not) obligated to agree, but I predict there will be more occasions to show that Windows is not an exception to this.

[*Great, we agree on that too. Why do you feel compelled to harp on this point at every opportunity? It got tiring after the first month. -Raymond*]