

Re: Unicode Support

Source: <http://coding.derkeiler.com/Archive/Assembler/alt.lang.asm/2005-04/msg00871.html>

- *From:* "Beth" <BethStone21@xxxxxxxxxxxxxxxxxxxxxxxxxxxx>
 - *Date:* Thu, 21 Apr 2005 07:28:02 GMT
-

Rene wrote:

> Chewy wrote:

>>> I must apologize that i have no idea, at all,
>>> about how this is usable or not usable for the real
>>> "Unicode Users".

>>

>> Would you prefer:

>> "Hello world"

>> or

>> 48h, 65h, 6ch, 6ch, 6fh, 020h, 77h, 6fh, 72h, 6ch, 64h
>> within your source file.

>>

>> I would prefer the first form (as anyone would). Now take the view of
>> the Hindi student in India who doesn't know english. Are you going to
>> force the second option on him, so he can use your assembler?

>>

>> (I know this is a poor example, but think about other languages, eg
>> hebrew, most asian, etc).

>

> Of course. Evident. I would also like it to work
> that way. What i am saying is that is simply
> `_DO NOT KNOW_`, if it actually works that way or
> not. I use an occidental KeyBoard, and i have no
> idea, at all, of what is going on, when, say, an
> oriental user inputs something directly in Unicode.

Okay...

First things first, when you register your RosAsm windows classes, you should use the "RegisterClassW" API, not the "RegisterClassA" API...the reason for this is that Windows takes this as a "cue" as to whether to send the messages with ANSI / UNICODE parameters in ANSI or UNICODE form...

If you register with "RegisterClassA" – as you're probably doing – then Windows takes this to mean that you want your application window to receive ANSI messages (some messages have ANSI / UNICODE variants...for instance, the character provided by "WM_CHAR" is an ANSI character when the window is "registered" as ANSI...but it will be a UNICODE character returned by "WM_CHAR" when the window is "registered" as UNICODE :)...so, yes, you

Re: Unicode Support

won't see any UNICODE characters appearing with "WM_CHAR", _IF_ you used "RegisterClassA"...

So, the most important thing is to make your editor window a "UNICODE window"...thus, register it with the UNICODE "RegisterClassW" to specify that (this is a "backwards compatible" thing, of course...it assumes ANSI unless you specifically call the UNICODE API, which tells Windows that the application is "UNICODE aware" and then it operates and sends messages in UNICODE format)...

The API "IsWindowUnicode" can be used on windows to find out if they are registered as ANSI or UNICODE windows...and, note that if you send a message from an ANSI window to your UNICODE registered window, then Windows does automatically "translate" such messages while delivering them...

And, in fact, that is all you should need to do...when the user hits a key, then the standard WM_CHAR should be delivered to the window...BUT, as this window is "registered" as being a UNICODE window, then the "wParam" parameter should now contain a full 16-bit UNICODE character, not only an 8-bit ANSI character...

So, when your Indian user presses a key for an Indian character, the "wParam" parameter for the WM_CHAR message should now contain a 16-bit UNICODE character code...

To deal with the UNICODE strings properly, obviously, then use the UNICODE API...so, "TextOutW" with a string of 16-bit per character codes, not just 8-bits per character...

Be careful, of course, with some operations...for instance, to make an ASCII character "uppercase" or "lowercase" is just a case of toggling a bit in the character code...this won't work, in general, on UNICODE characters...BUT, don't worry, you do not need to actually be an "expert" in the character set...Windows provides a set of API functions which correctly deal with UNICODE characters in the right way:

"CharLowerW" – converts UNICODE character to "lowercase" correctly
"CharUpperW" – converts UNICODE character to "uppercase" correctly

[Note that, of course, not all languages and scripts actually have an "uppercase" or "lowercase"...so, for some characters, calling "CharLower" will have no effect...but, then again, this is the same as punctuation characters like "<>.,@:;[]%&*" which also do not have any "case"...so this idea is not unusual but just the case that some languages are also the same with their alphabet characters, as with the numbers and punctuation...so, you know, just something to remember, so you don't start thinking something is "broken" when you can't get an "uppercase" Thai character...they don't have any, so it's a not a mistake or anything :)...]

There's also "lstrcmpW" – which "mimics" the C function of the same name – which is also "UNICODE aware" when it makes string comparisons for you...

Re: Unicode Support

Well, I think you get the general point: Windows has "UNICODE awareness" built-in, so if you use the Windows API functions on the UNICODE strings, then it has all the "rules" built-in to do this correctly and sensibly...

Also, your UNICODE window procedure should call "DefWindowProcW" (because, as noted, Windows now sends "UNICODE enabled" messages to your window, so when passing it on to "default processing", you should call the UNICODE default processing, which correctly understands the "UNICODE enabled" messages)...

Generally speaking, if you change all your API calls that end with "A" to those that end with "W" then this will put Windows into "UNICODE mode" when dealing with your window...and the messages like WM_CHAR are automatically "upgraded" to return UNICODE characters, rather than only ANSI characters...this then allows your users to type in whatever characters they like and the 16-bit "wide character" code returned has a unique "code" for each character (basically, it's much the same as ASCII / ANSI but the characters are 16-bits, so there are more characters...but, otherwise, the way Windows implements UNICODE is only to make the characters bigger...everything else should be more or less the same...but, as noted, be careful about manipulating the character codes directly and, rather, use the actual Windows API functions for things like "CharUpper" to take advantage of the "built-in" UNICODE support in Windows...you know, Windows itself has all the correct "look-up tables" to do it properly...just call the API to make use of that then you yourself do not need to be any "expert" :)...

And, of course, be aware that ALL the messages sent to the window (when registered as a "UNICODE window" :) will be sent in the UNICODE character set...not only WM_CHAR but also WM_GETTEXT for the window title, WM_CREATE (the CREATESTRUCT will have UNICODE strings inside it...but, then, you used "CreateWindowExW" to create it, so you'd expect it to return the same UNICODE strings you gave to create it...that makes sense, yes? ;) and so forth...

You ARE allowed, though, to call the ANSI API, if you really want (e.g. "SetWindowTextA", supplying an ANSI string, not a UNICODE string) and Windows will "translate" as necessary...in other words, it'll pad the ASCII bytes with zero automatically...obviously, if you do that, though, then you can only use ANSI characters in the string you supply)...but, well, once you switch to UNICODE, there doesn't really seem to be too much need to do that...but if you do need to, then you still can use the ANSI API...they are just "limited" to the 8-bit ANSI strings only and get "translated"...

[Mind you, Windows itself is inherently UNICODE internally...hence, in fact, when you've been writing your ASCII / ANSI programs up to now, this "translation" is actually happening all the time...you just don't see it happening because it "translates" back and forth to always show you ANSI strings...so, in fact, switching to UNICODE is not getting you any "extra cost" but, actually, the opposite in that Windows actually stops

Re: Unicode Support

"translating" all the time – just "deals direct" with the program in its `_native_ UNICODE` – except for when ANSI API are explicitly called :)]

One problem with this, of course, is that UNICODE is not supported directly on the 9x kernels...if this is a problem then send an Email to Jeremy Gordon (of "GoAsm" fame :) because, as you know, he's got very good UNICODE support...and there is some "UNICOWS.DLL" which helps "patch over" this...but, well, Jeremy's the better one to ask about that because I've never used it (indeed, I know of it through reading Jeremy's website...so, all I know about it, is what he's told me...hence, better to ask the "master" than the "student", yes? ;)...

Though, the obvious and simple way to deal with this is to "detect" what Windows Rosasm is running on..."GetVersionEx" being the API to do that (basically, look for an NT based kernel :)...and then use the ANSI / UNICODE API, as appropriate...so, if UNICODE is available, then switch to the UNICODE API or just carry on as normal with ANSI, if it cannot be detected...this, though, might make the program bigger to deal with it this way, as you might need "two versions" of many things (it's a simple approach to it and, thus, might not be the best one to take...but it would be sure to work ;)...so, go and take a look at the "UNICOWS" thing first, as that might save time and effort...

- > There is a Case, in the RosASm MainWindow Messages
- > holding for "&WM_IME_CHAR" actually, that is
- > effectively supposed to input Double-Bytes, into the
- > Source. But i do not know:
- >
- > 1) If it really works.
- >
- > 2) What goes on, in the life of the Source, with
- > the so many Edition and Compilation Functionalities,
- > and at what extend they could choke, on some Bytes
- > of the Double Bytes.
- >
- > For example, RosAsm always parses the Strings
- > from Quotes to Quotes, on a `_BYTE_` Basis scan.
- > Let us imagine, that, inside some Double Byte,
- > the same Byte as a Quote would be inserted from
- > a Unicode String... Can this happend? i simply
- > do not know (It would create an "Unpaired Quotes"
- > faultive error Message)...

Note: There is a `_DIFFERENCE_` between "multi-byte" and "wide character"...

"Multi-byte" – like you're talking about above – is how things like oriental characters were dealt with `_BEFORE_ UNICODE`...

If you go for a UNICODE approach – using "RegisterClassW" and all the other "W" API – then this should NOT be a problem...as this is the UNICODE "wide character" approach...where – at least, in Windows' implementation of it –

Re: Unicode Support

every character is 16-bits wide...

So, if using "wide character" then your scan should be much the same, except, obviously, that each character is 16-bits wide, so it's a `_WORD_` basis scan, not a `BYTE` basis scan...all the ASCII codes are, as you probably know, in the same places in UNICODE – just pad them with a zero byte to make them 16-bits wide – so the actual logic shouldn't actually need much, if any, changes but to, you know, make it a `WORD`-based rather than `BYTE`-based...

The "multi-byte" stuff is NOT actually "UNICODE"...that is the "confusing" stuff that existed before UNICODE to deal with oriental characters...

UNICODE is called "wide character" – which, by the way, is what the "W" in the API names stands for – and "multi-byte" is the "old way" to deal with oriental characters...

And, indeed, that's what's so useful about UNICODE: It makes a standard for a "wide ASCII", so to speak...makes the characters bigger at 16-bits...and then it can deal with all the characters in all the languages and scripts in exactly the same way...it makes it all much simpler, as a kind of "big ASCII"...instead of all the "multi-byte" / "code pages" confusion (which `_IS_` terribly confusing and awkward: That is, in fact, the underlying reason why UNICODE was invented...to stop it being quite as "confusing", as well as to allow all the different characters to be dealt with at the same time (as with "code pages", you could only select one of them at a time, so couldn't have oriental `_AND_` English `_AND_` Hebrew all in the same document...which was another big "drawback" of the "code page" system...very annoying, for instance, if your a language student and you want to write an Email in English but, obviously, you want to be able to include "Runic" characters in order to discuss the "Runic" language with other people...as I said in that other post, without UNICODE, it would, in fact, be `_impossible_` to send that post any other way :)...and to try, as far as possible, to make it work much like ASCII works but it's just a "bigger character set" with "wider" characters :)...

[Yes, since Windows included UNICODE support, the UNICODE standard now includes `_more_` than 16-bits of characters...note, though, that only "historical scripts" – ancient egyptian hieroglyphics, for example (don't think too many RosAsm users will want that, eh? ;) – and thousands of oriental "ideographs" are actually in this "upper range" at the moment...everything else is in the lower 16-bit range, so it's only a problem for oriental ideographics, really...basically, Windows implements an earlier standard...but one that's "good enough" for most purposes, except these "extra characters" that have been added for oriental "ideographs"...there are "special characters" which "escape" into the "upper range" added to accomodate this...note that the problem is really with Windows having "jumped on" early when it was only designed to be 16-bits in size...if implementing UNICODE `_now_`, then Microsoft would probably use a "wider character" again to deal with it...16-bits, though, covers 99% of the people (even the oriental ideographs: The ones in the

Re: Unicode Support

"upper range" are "additional"...plenty do exist in the first 16-bits...just that, well, they grossly underestimated just how many of these ideographs actually exist...probably some 64K worth of ideographs exist _alone_, without anything else)...]

>>> As for writing, as you suggest, say, the labels
>>> Names, in Unicode, directly in a RosAsm Source,
>>> this will never be implemented. I do not see any
>>> reason for doing so.
>>
>> So your source code will forever be ASCII?
>
> The Assembly Source, Yes. Unicode will be (or are)
> for Data Strings only, and possibly, maybe, for
> Comments.

Yes; But, as noted, in order to get Windows working in "UNICODE mode" – so it sends UNICODE characters with WM_CHAR – then you need to call a few UNICODE "W" API to set that up...plus, to display the characters in the editor, you'll need the "TextOutW" (and appropriate fonts...but, well, the Indian user will surely have the right Indian fonts on their Windows, even if you don't...and there is always that "Code2000" shareware font, I mentioned before, deliberately designed to include as many characters as possible, for "testing purposes" (and also just for "fun" in looking at all the new UNICODE characters...I like the little "snowman" character in the "dingbats" range...part of a set of "weather symbols", like a shining Sun for "sunny" and an umbrella for "rain"...the snowman being "snowing", I presume...there's a lot more characters than simply languages in UNICODE too, you see...those DOS "box drawing characters"...the standard mathematical symbols...circuit board diagram characters...arrow shapes...chess pieces...signs of the zodiac...and many other things like that ;)...Microsoft's own "Arial" and "Courier" fonts – included with Windows – cover a large range too, on the UNICODE-based OSes like NT / XP and so forth :)...

So, you will need to call UNICODE API here and there...but, yes, the basic change is just that the data strings all have 16-bit wide characters...and you _shouldn't_ need to do too many changes to support UNICODE...

At least, part of the idea behind UNICODE was to try to make supporting all the various languages in the world just as simple and standard as with ASCII...a "bigger ASCII", if you like...so, though there are some "exceptions", the basic idea is that it _SHOULD_ more or less be as simple as making your data strings use 16-bits per character rather than 8-bits per character...but, otherwise, basically the same to program as ASCII always was...so, generally, it _should_ just be (more or less) a case of using the "W" API instead of the "A" API and that the strings are twice as big (because each character is now 16-bits, not 8-bits :) BUT that there shouldn't be any other real differences in your programming...a few "exceptions" do exist – the "byte order mark", for example, isn't a real character but an extra "control character" (there are a few extra "control

Re: Unicode Support

characters" added that you might want to process specially – just like the ASCII "control characters" – that may add a few more "exceptions" to look for in your processing :) – but the general "theory" of it all is that you should really only need to call the UNICODE versions of the API (with strings that are 16-bits per character instead) while everything else SHOULD broadly "stay the same"...

- > But, for naming the Labels, sure, not, and never.
- > That would be much no use complication, IMHO.

Yeah; It might be "interesting" – as some "mental exercise" – to do such a thing...but, in practice, would anyone use it? Would it be useful at all?

In "principle", you could do it...but, in practice, the usefulness of it is questionable...

Especially when the mnemonics and things are not going to change, so it would probably prove incredibly awkward, anyway...you know, you type in Latin for the mnemonics, then have to "switch keyboard layouts" to Russian, type in a label, switch back to a Latin layout, type some more, switch "layouts" again, etc., etc....you can set up "shortcut keys" to "switch layouts" quickly in Windows and such...but, well, you also have to consider the practicality of typing this out...because the mnemonics are already Latin, then the programmer will need to be able to type those characters...and switching back and forth? Well, I think even if you could allow Russian or Japanese programmers to type in such labels, it would prove so practically awkward for them to "switch" all the time, that they'd willingly just stick to Latin to save the effort and bother...

I'm not against the idea itself, though...but, in practice, it'd be a lot of effort to implement, when probably no-one really actually wants it...that's why, for LuxAsm, I was only thinking about "basic support" for the comments and the strings and that kind of thing...perhaps, if really, really bored and there's nothing else that needs doing, it could be "extended" to labels too...but it's hardly a "priority", even for those programmers out there who like the idea of being able to type things in in their native languages directly...

The point to remember, basically, is that a programming language is NOT quite the same as a natural language...it is, to an extent, a "language of its own"...after all, does assembly code really look like any natural English text you'd read? No...show that source code to most ordinary English speaking people (who are not programmers) and it might as well be Chinese or some other language they know nothing about...

For some "international word-processor" then, yes, you need to have a high level of "support"...but, for programming languages, the real practical benefit is just to be able to type those Russian "error message" strings in Russian directly...it's more "convenient" and doesn't require too much "messing around"...if you can type it with your keyboard then you can just type it directly into the source code...this could be useful, if a

Re: Unicode Support

programmer is, for example, writing a program for a Russian audience, where they want all the text and error messages and menus and things to appear in Russian...it'll make it as easy for them to program such things as it is for us Latin (especially English, where we don't even have "accent marks" or anything, just the basic Latin characters ;) users to do with our source code...you just type in the strings directly with your keyboard in the "data section"...it just makes it as easy for them to do it as anyone else...

Also, it should be stressed that UNICODE _ISN'T_ solely about "languages" (that's the "main focus", obviously, but not everything :)...there are actually other potentially useful characters in there, which can benefit everyone, whatever their language...

You can pop some "box drawing characters" around your console applications to get that DOS-style windowing working...or, if you want to write a chess program, then there are actually chess characters in the "dingbats", so, with UNICODE, you could potentially write that as a "text mode" program but still have recognisable "king" / "rook" / "knight" pieces on the screen...or, if you're doing a card game, then there are the "suits" of a deck of cards...

Perhaps you want to do some "astronomy" program: There are the standard symbols for all the planets (plus Sun and Moon :) included...if you want to write some astrological "horoscopes" program then the signs of the zodiac are there too...

Or, as noted, you might just simply use something like the historical "Runic" characters as some "elf language" in a text-mode adventure game...

You can improve your "plain text editor" by using the "control pictures" characters (these are `_visible_` characters for the ASCII control characters...so, you could have a "mode" – a bit like on Word – where you can select to `_see_` the spaces and tabs and newlines :)...

The IPA ("international phonetic alphabet") characters are included, so you can use that to show "pronunciation" of words, in the usual "standard" that dictionaries use to show how to pronounce words...that could be useful for some program out there, perhaps...some "learn how to speak Spanish" program (which can do it all and still actually be a "text-mode" console program)?

There's even Braille pattern characters included...which might immediately seem a bit silly but, of course, this could be used to send Braille files to some "Braille printer", making it easier for blind and partially sighted people to store text on their computers (note: These characters demonstrate what I was saying about UNICODE being about "assigning numbers to characters" and the actual "glyphs" for these is another matter...you know, UNICODE supports the ability to `_store_` Braille patterns in files, even if "glyphs" for these characters would be a bit silly for blind people...special programs can be written to interpret these characters in a

Re: Unicode Support

sensible way...UNICODE is just about the "storage" of the characters themselves :)...

There are the six faces of a dice...some "dice game" could use those...and there's just a bunch of "geometric shapes" – circles, half-circles, diamonds, etc. – which could be used for lots of things...there's also "terminal graphics" characters, which do that "teletext" type of simple graphics...

Actual proper "roman numeral" characters (semantically distinct from just using the Latin characters...so, for example, a program can know the difference between "roman numerals" and letters...or, in drawing the characters, it can automatically draw those horizontal lines above and below the numerals, just like the Romans used to chisel their numbers into stone :)...)

Currency symbols for some "accounts" programs or whatever...musical symbols for a music program (there's proper "treble clef" and "bass clef" characters, for example...so, Rene, if you came up with some new jazz tune with your guitar, then you might be able to use those to send it to other guitarists, in a UNICODE text file, but where you've got the correct musical notation :)...)

Or how about an "advanced calculator"? There is a full complement of mathematical characters to be used...so, you could even make a "calculator" that could deal with mathematical predicates directly, perhaps?

There's also plenty of characters in the UNICODE standard that, in fact, have nothing to do with "languages" at all...but are just very "handy" characters to have, in general...

So, it isn't just about allowing those Russian programmers to input Russian strings...you might be writing an English-based program but might find some of these other characters "handy" for different purposes...there are enough other "useful characters" in UNICODE to justify it all alone, even if you didn't want to bother with "other languages"...I'm thinking, for instance, that Paul's "adventure games" could be made very interesting indeed – "elf language" (like Tolkien, just "borrow" a script no-one speaks anymore like "Runic" or "Ogham" or "Gothic" ;), "box drawing" around things, an umbrella appearing in the top-right corner to indicate that it's started "raining" in the game world, you meet a dwarf who challenges you to a "dice game" in the tavern, etc. – without even becoming a GUI (still just a "console" program :)...)

> I Have read a long (as usual...) Post, from Beth,
> some long time ago, where she was, if i understood
> well, promoting a kind of UTF something, that should
> be the universal default. As i was not much interested,
> i did not read it carefully, but...

Not a "universal default" – you weren't reading carefully, were you? – but

Re: Unicode Support

that UTF-8 is actually an "encoding" for UNICODE which is "ASCII compatible" (for all the ASCII characters, the UTF-8 encoding is exactly the same as it always was :)...and is "byte-based"...this UNICODE format could be useful for programs that are already ASCII-based to easily "upgrade" them...it's the "encoding" that Linux uses for UNICODE...

> Beth... maybe.. a couple of words would help him...

Maybe a couple of words may help you too about how to potentially implement UNICODE in RosAsm, as you did (indirectly) ask how the WM_CHAR works for Windows in "UNICODE mode" :)..

Beth :)

- *Follow-Ups:*

- ◆ ***Re: Unicode Support***

- ◆ *From:* Betov

- *References:*

- ◆ ***Unicode Support***

- ◆ *From:* Chewy509

- ◆ ***Re: Unicode Support***

- ◆ *From:* Betov

- ◆ ***Re: Unicode Support***

- ◆ *From:* Chewy509

- ◆ ***Re: Unicode Support***

- ◆ *From:* Betov

- Prev by Date: ***Re: Unicode Support***
- Next by Date: ***Re: Does MSIL Qualify?***
- Previous by thread: ***Re: Unicode Support***
- Next by thread: ***Re: Unicode Support***
- Index(es):
 - ◆ ***Date***
 - ◆ ***Thread***