

Re: Unicode Support

Source: <http://coding.derkeiler.com/Archive/Assembler/alt.lang.asm/2005-04/msg01003.html>

- *From:* "Beth" <BethStone21@xxxxxxxxxxxxxxxxxxxxxxxxxxxxx>
 - *Date:* Fri, 22 Apr 2005 16:40:49 GMT
-

Randy wrote:

> Beth wrote:

>> In fact, my little "test" which demonstrates that NASM `_already_`
>> deals with UTF-8 comments and strings, proves the point a
>> different way...some tools `_already_` are "accidentally"
>> supporting UTF-8 to that "basic level", without even
>> knowing it...and the fact that no-one has actually realised
>> this (I didn't either until I thought it would be interesting
>> to see what NASM would actually do ;), shows how great the
>> "demand" is...if people were regularly wanting UTF-8 source
>> files passed through NASM, then your post should have had Frank
>> shouting "NASM already does it!"...but, I bet, not even the
>> NASM developers have actually realised that it does already
>> work to this "basic level"...indeed, they could cheekily add it
>> to the "features list": "NASM has basic UTF-8 support!"...as if
>> they actually "intended" it or something...shhh! Don't tell
>> anyone! ;)...

>

> Not knowing much about UTF-8 (my Unicode knowledge extends as far as
> UTF-16 and that's about it), I would say that HLA v2.0 would handle
> literal strings of this form as long as the character code for quote
> can never appear in a MBCS (multibyte character sequence).

It cannot appear in the multi-byte sequence...it's deliberately part of the design that every single one of the multi-byte bytes has the high bit set, so it can't be confused with any ASCII character (the "ASCII compatibility" works both ways, so to speak ;)...

I put more of the exact details in my post to wolfgang just now...but, basically, the design of UTF-8 is exactly designed for the "ASCII compatibility", including that, so long as a program doesn't care about bytes with the high bit set, then they may very well work without modification...

It's very possible that many of these assemblers will work like NASM does and even though they see some multi-byte sequence, this doesn't effect the processing and it actually works (because "xterm", in this case, directly understands the UTF-8 multi-bytes, so when NASM simply "passes it through" to the xterm console then xterm interprets it as the UTF-8 :)...)

Re: Unicode Support

Of course, what's actually happening is that...say I use Notepad to write the source code, then XP Notepad directly understands UNICODE and you can select the "save as UTF-8" option...then, when passed to NASM, what NASM sees is more than one character in the >127 range...but, as its a string, NASM just passes it through to the output file "as is"...then when "xterm" receives that string, it natively understands UTF-8 and displays the correct character...

So, amusingly, even though NASM has no "special processing" for UTF-8 at all, because it just passes the >127 bytes through "as is" for strings (and just ignores any bytes between ";" and newline in comments :)...then I could actually create that Japanese "hello, world!" program in Notepad, being able to see only the Japanese characters...then, when I run it with xterm, the Japanese strings appears (actually, the two ideographs for "world" were missing and the "font missing" character appeared...obviously, X lacks the fonts for that range of characters but it does have "Hiragana" – the Japanese "alphabet" (well, one of them, they have more than one: Talk about confusing, eh? ;) – but the very fact that TWO "missing character" blocks appeared rather than more than two (because the "multi-byte sequence" is more than two bytes, so though it doesn't have the "glyph" to show the character in the font, it IS interpreting it correctly :) and that the "Hiragana" characters of the "konnichiwa" were showing just fine...xterm is "interpreting" it correctly...I just need to install the right fonts, it seems, if I want to do more Japanese programs ;)...

In simplest form – a basic "table" – UTF-8 works like this:

```
U-000000 to U-00007F: 0xxxxxxx
U-000080 to U-0007FF: 110xxxxx 10xxxxxx
U-000800 to U-00FFFF: 1110xxxx 10xxxxxx 10xxxxxx
U-010000 to U-1FFFFF: 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

...where that's the binary of the bytes on the right, obviously...and the "x"s are replaced with the binary of the character code you want...and, other than "overlong forms" (trying to encode an ASCII character with one of the longest multi-byte sequences) is to be rejected as "invalid" (in order to ensure that the UTF-8 is "normalised" for comparisons and such :), that's about it...

You'll notice that only ASCII bytes have the highest bit clear...and then the first byte of the multi-byte sequence is different depending on the number of bytes in the sequence (indeed, the number of high bits set in the first byte is the amount of bytes in the sequence, you'll note :)...then the rest of them start with "10"...values FEh and FFh are, in fact, not possible and invalid...

Also, being byte-based throughout, there are no "endianness" or "byte order mark" (BOM) stuff to worry about...plus, UTF-8 can address all UNICODE characters (indeed, I've cut that down, you can validly extend the

Re: Unicode Support

multi-byte sequences to cover all 2^{31} "UCS" values (up to a six byte multi-byte sequence :)...but by the UNICODE design, there's no characters that high up and they "intend" that there never should be...mind you, they "intended" not to overflow 16-bits but did anyway...I suppose, though, if we could trust them not to "overflow" the 1024K characters they've "reserved" then all the ones above it could be used for "private use"...say, for "rich text formatting" or something like that...I don't know...a possibility...but NOT a good idea, if we can't actually trust them not to "overflow" 1024K or it'll then become "incompatible"...perhaps best, unlike Microsoft jumping in early with the 16-bit encoding, not to do any such thing until "mature" enough to be sure that any such characters are really "free for use" ;)...

It's pretty simple and was deliberately designed, as you can tell, to be "ASCII compatible" (including that no "multi-byte sequence" can be confused with any ASCII character)...and the format of the "multi-byte sequence" is also designed for easy "resynch" because the "first bytes" are all marked as such (while ASCII characters are just ASCII characters, so those are obvious too :)...so, even if you started half-way through a multi-byte sequence, you could be "back in synch" by the next character (or could "back up" to find this character's "first byte")...plus, the "nice" feature that you can TELL you're in the middle of "multi-byte sequence" too...

Ken Thompson obviously put on his thinking cap for this one because it's quite a "nice" encoding for UNICODE...still relatively simple but has all those "useful" properties: byte-based (no "endianness"), ASCII compatible, addresses full range, easy "resynch", can immediately tell if you're "mid-way" through a multi-byte character (by making the first bytes different and actually can be easily tested as they are in a different "range" to the other bytes :)...and, at least for all the "currently defined and reserved" UNICODE characters, it's only four bytes long maximum (which, in fact, compares directly to UTF-16 in the same situation, yes? BUT UTF-8 is typically "smaller" for other cases :)...

The drawback, of course, is that it's "multi-byte", so that's not as simple as UTF-16 to process (but, then again, for the >64K characters, UTF-16 can go "multi-byte" too, due to that whole "the first standard was far too inadequate thinking 16-bits was enough (when the Han ideographs are probably 64K by themselves without anything else)"...did anyone actually bother to find out how many ideographs exist before that first draft? Obviously not...the "variable length" not being quite as convenient to process as fixed sized characters...

- > HLA v1.x,
- > however, would not be happy with the character as Flex rejects all
- > character codes in the range \$80..\$FF out of hand.

Ah, well...yes, that won't be UTF-8 compatible then...

- > There is, however, another issue that gets you into trouble with MBCS.
- > When you start adding sophisticated compile-time language facilities,

Re: Unicode Support

- > such as string functions, handling all the different character sets
- > becomes a nightmare.

Yes, indeed...things like "ToUpper" suddenly become a little more "non-trivial" than flipping a bit when the value is in a certain "lowercase" range...

I've not exactly looked completely at what characters are "uppercase" and "lowercase" and so forth (indeed, only a few scripts have "case": Latin, Cyrillic and one other I can't remember off the top of my head ;)...but I think it'll probably have to be a case of a "look-up table" for this kind of thing...

UNICODE provide "data tables" for all the defined characters where various "properties" are defined for each character, like its "case" and such (comes on a CD, if you actually buy the UNICODE book itself :)...and then there are "rules" explained in the first few chapters about how to process them...so, you know, you shouldn't need to sit down and become some "expert" in 50 languages or anything...

But, yeah, the "processing" becomes more complicated...mind you, Windows already has things like the "CharUpperW" API, which uppercases a UNICODE character correctly...indeed, all of Windows' string API in their UNICODE versions are, of course, "UNICODE aware" and should correctly process them...

BUT, of course, this partly negates one of the usual "advantages" of assembly language coding that you can deal with the strings yourself in "optimised" ways...calling an API to "uppercase" each letter? Well, not "ideal", is it?

On Linux, there aren't any "string API" in the kernel but Linux is supposed to use UTF-8 "natively"...in this case, then there are C standard libraries with "wide character" string routines that are "UNICODE aware"...but, well, again, that's "defeating the purpose" a little with using assembly language, if you've now got to link in C libraries for this...

Mind you, processing strings in assembly language manually shouldn't involve anything "OS specific", so perhaps some out there wants to write a "UNICODE string library"?

- > Then, in HLA's case, there is also the issue that
- > you need to provide standard library routine equivalents of string
- > functions for UTF-8 strings (you think zero-terminated strings are
- > painful to compute the length of? Try UTF-8!).

Yeah, and being "multi-byte", you then have to distinguish between "number of bytes" and "number of characters" because the relationship isn't simple anymore either...this is the main "drawback" of UTF-8...

Plus, if HLA added UTF-8, then should it also add UTF-16 support and

Re: Unicode Support

UTF-32? Routines to "convert" between them?

Oh, yeah...I fully appreciate the problems...

With LuxAsm, though, we'll try to "think ahead" on this (so, even if not implemented straight away, we should Hopefully not commit to anything that screw up adding it in later on...so, for instance, remembering to distinguish "number of bytes" and "number of characters" in any string routines for the LuxAsm "standard library" or whatever ;)...and there's a "nice" thing that Linux's "native" format is UTF-8 and UTF-16 and such aren't really seen, except for "compatibility" with things sent over from Windows and such...so, being "Linux specific" then it would not be "unreasonable" for us to only bother with UTF-8 (which, as noted, is the most "handy" with "ASCII compatibility" for the source files and such :)...)

Ah, for sure, there's nothing particular "easy" in this...but, well, supporting all those languages from around the world simultaneously? It's never going to be "easy", is it? But, you know, "you reap what you sow", isn't it? "No such thing as a free lunch" and so forth...it's more complicated but that's because you're dealing with a bigger "kettle of fish" here...and, truth is, if dealing with all the previous stuff to the same kind of "support" level with thousands of "code pages" and "double byte character set" and so forth...well, that would be `_MORE WORK_` (kind of by definition: This was the "problem" UNICODE was originally invented to "solve" :)...)

No, it's not going to require "more work" to support this...but, well, if UNICODE picks up then, at some point, I'm going to have learn all about it...so, dive in head first and look at it now...I did something similar with Windows before...jumped into Windows (16-bit) coding when Windows was only just picking up "serious" use generally (mind you, only C coding...like others, I was "given the impression" by Microsoft and such that ASM was "impossible"...also, I'm not any "anti-C" coder or anything...so, I'm comfortable with that too and, to be honest, kind of "didn't think" of assembly until later :)...and I suppose perhaps I'm sort of also doing with Linux and X coding too, thinking about it...ah, it's the way I am: I fear "standing still" far more than "fearing change" :)...)

> An assembler like NASM, that doesn't provide much in the way of
> compile-time string handling, might actually get away with "accidental"
> UTF-8 support.

Ah, mind you...I've not tried any NASM "macros" on this...indeed, because of the difference between "number of bytes" and "number of characters" in UTF-8 then any kind of "macro loop" going through character-by-character? Doomed to failure...well, it is "accidental" support, so to speak...hence, "limits" to how far it extends are only to be expected...this, in a sense, is really where LuxAsm will differ because UTF-8 in strings and comments is about the "level" we've decided to aim for...but the support isn't "accidental" and it realises its reading UTF-8...

Re: Unicode Support

As for the "string routines" in the LuxAsm "standard library"? Well, I've got the CD with the UNICODE "data tables"...I suppose, when the time comes, I'll just have to give it a go at making that "look-up table" routine...

Ooh, good point...LuxAsm team! Won't our "standard library" have to have LGPL rather than GPL, if it's to be allowed that LuxAsm "standard library" routines can also be used in non-GPL programs too...oh, boy...how many different "licences" are we juggling already? MIT licence on X, GPL on LuxAsm, LGPL on any "standard library"...

> But when you've got a sophisticated macro system and
> compile-time language, supporting MBCS turns out to be a *lot* of work.

Yeah...but we will be having a sophisticated macro system and compile-time language, in our "plan"...so, well, looks like there's just got to be a "lot of work" ahead of us...

Beth :)

--

"Let me put it to you bluntly: in a changing world, we want more people to have control over your own life."

[George Walker Bush Jnr., explaining how "globalisation" works]

• *References:*

- ◆ **[Unicode Support](#)**
 ◇ *From:* Chewy509
- ◆ **[Re: Unicode Support](#)**
 ◇ *From:* Chewy509
- ◆ **[Re: Unicode Support](#)**
 ◇ *From:* Beth
- ◆ **[Re: Unicode Support](#)**
 ◇ *From:* randyhyde

- Prev by Date: **[The ASM32 licence that Betov is in violation of.](#)**
- Next by Date: **[Re: Awe diddums, fearless leader has gone into hiding.](#)**
- Previous by thread: **[Re: Unicode Support](#)**
- Next by thread: **[Re: Unicode Support](#)**
- Index(es):
 - ◆ **[Date](#)**
 - ◆ **[Thread](#)**