

## Re: Fast UTF-8 strlen function

---

*Source:* <http://coding.derkeiler.com/Archive/Assembler/alt.lang.asm/2005-05/msg00477.html>

---

- *From:* "Beth" <BethStone21@xxxxxxxxxxxxxxxxxxxxxxxxxxxx>
  - *Date:* Wed, 11 May 2005 21:26:16 GMT
- 

Frankie say:

> Randy wrote:

>> Is there a fast UTF-8 string length function floating around?

>

> What's the meaning of "string length" of a UTF-8 (or other

> unicode) string? Length in bytes, or length in characters?

Yes; This is what screws up a lot of current ASCII string routines, in that it does not make a distinction between "number of bytes" and "number of characters" (presumes the two are "one and the same thing" :)...

For UTF-8, the main basic "change" you have to make to your string routines is separating these two things out...to provide some "str.len" for the "number of characters" and some "str.size" for the "number of bytes" or whatever...

The simple "byte to character" relationship is gone so routines – and the programmer – must learn to start thinking of "number of characters" and "number of bytes" as being two separate and distinct properties of a string (though not totally independent – you couldn't have less characters than you've got bytes, for example – it's probably easiest to just think of them that way once dealing with UTF-8 :)...

This is perhaps the only real "stumbling block" in adding at least simple UTF-8 support to many currently ASCII-based programs: That your string routines (or the ones in some "string library" you're using) don't make a distinction between "characters" and "bytes"...

> As NoDot points out, if we want length in bytes, a "regular"  
> strlen ought to work.

Yeah, it does...

> Beth raised this issue on the luxasm-devel list, too

> (although not asking for an optimized function). IIRC, her

> example sent a string to stdout, and the question was "what

> goes in edx". I'm pretty sure we need length in bytes here,

> and in "most" cases (allocating memory, e.g.), but sometimes

> we'd want the length in characters (plus "font metrics" to

## Re: Fast UTF-8 strlen function

> determine where the "next" print position would be, for  
> example).

Actually, thinking it over some more, because the "write" function is also used for things like `_writing data to binary files_`, then I'm pretty sure the answer is "number of bytes"...otherwise, of course, it could be screwing up other things: After all, if using "write" to access the `"/dev/fb"` framebuffer device then having to specify "number of UTF-8 characters", when you're writing pixels? Doesn't make sense..."write" `_MUST_` just deal solely with "bytes" only...

This makes logical sense...I was "put off" by having documentation that listed "number of characters to write" for the function call...which is, in fact, a false presumption of the documentation to say "characters"...for the UNIX "everything is a file" scheme, then, of course, "write" might NOT be being used with any "characters" at all...you also use this to write out chunks of binary data to files or to hardware devices as well...

Mind you, another aspect of my example not putting in the correct "length" was also the simple point of `_how to work it out_`...that is, what I'm seeing in Notepad or whatever is a bunch of Japanese characters...now, I'm `_seeing_`, say, three characters...but how many bytes is that? Of course, I could just do the "StringLength equ \$ - StringAddress" trick after each string to get NASM to count it for me...or if I was feeling particularly bored, then I could individually look up the characters in my UNICODE reference (which would take forever)...I got "lazy" and "cheated"...I just stuffed in a number that I knew to be "too large": That way, I knew it would print...some extra "garbage" characters might appear after the string too...but, well, it was just a quick "test" and I couldn't be bothered to do it properly...

But, in this context, I was thinking that, for LuxAsm, whenever you create a string variable then, simply, there can be both "#length" (number of characters) and "#size" (number of bytes) properties created for them...which is a perfect example of where these "properties" can come in very useful: LuxAsm can automatically work this out and provide the length / size as a set of "properties", `_LOCAL_` to that variable (so, in no way does providing this support "pollute the namespace" or anything...and these are just "compile-time", so if you don't care, then ignore them and "no harm, no foul" :)).

> I suppose... if we encounter a byte with the high bit clear,  
> we just count "one". If we encounter a byte with the high  
> bit set, we determine how many bits are set (look-up  
> table?), and skip that many bytes, counting "one" for the  
> whole mess... I don't see an "optimized" version of this  
> working out very well...

Actually, UTF-8 is even designed with this in mind too...as the "first byte" of a multi-byte sequences is `_DIFFERENT_` to the subsequent bytes...

## Re: Fast UTF-8 strlen function

So, count these bytes:

```
0xxxxxxx
110xxxxx
1110xxxx
11110xxx
```

But ignore these bytes:

```
10xxxxxx
```

Or, put more simply:

```
ASCII bytes: x <= 7Fh
Continuation bytes: 80h <= x <= BFh
First bytes: x >= C0h
```

And we count the "ASCII bytes" and "first bytes" but ignore the "continuation bytes"...until we hit a NULL...

So, how do we optimise that into something speedy and easy, folks?

Well, if the top two bits are set then it's a "continuation byte" that should be ignored for the "character count"...but the rest can be counted in the usual way...

So, one possibility is to use an ordinary "strlen" but with an extra piece of code added that `_skips_` adding one to the "character count" when the top two bits are set...

```
> Beth says Nasm accepts UTF-8 strings in quoted strings (and
> comments) "by accident". I don't think it's "by accident", I
> think it's by "careful design"... not *Nasm's* design, but
> UTF-8's.
```

Yes, quite right...obviously, I meant that it was an "accident" in the strict sense of "the NASM folks weren't thinking about supporting UTF-8 but, lo and behold, it works anyway!"...

But, yes, the reason `_why_` it works is because UTF-8 was designed carefully to be as "ASCII compatible" as possible...indeed, many ASCII-based programs will work with UTF-8 just fine...problems only start arising when the difference between "number of characters" and "number of bytes" is `_IMPORTANT_` to how the program works...then it'll probably need modifying...

In fact, the point above that "first bytes" are deliberately `_DIFFERENT_` to "continuation bytes" in the multi-byte sequences is yet more "careful design"...they both have the 8th bit set BUT only "first bytes" have the 8th and 7th bit set at the same time (the continuation bytes all start with 8th bit set and 7th bit clear :)...)

Re: Fast UTF-8 strlen function

## Re: Fast UTF-8 strlen function

And this little bit of "careful design" is also included so that you can `_TELL_` if the byte you've got (with the 8th bit set) is a "first byte" or a "continuation byte"...this helps "counting characters" (just count "ASCII characters" and "first bytes", ignore "continuation bytes")...and, also, if you jump into the middle of a character – you read a "continuation byte" – then you can either "rewind" to try to find the "first byte" or just ignore the following bytes until you hit an "ASCII character" or "first byte", in order to easily "resynchronise" your reading with the characters in the string...

As I say, Ken Thompson (who came up with UTF-8, scribbling on a scrap of paper in his lunch hour, apparently :) had his "thinking cap" on that day...because, all things considered, it remains refreshingly "simple" while still having this "careful design" that `_has_` thought of the "issues" of remaining "ASCII compatible" (that, yes, it isn't possible for any non-ASCII character to be confused for an ASCII character...that it does make a distinction between "first bytes" and "continuation bytes" that's pretty easy to detect, which helps implement "strlen" functions or "resynchronise" and so forth :)...)

It's one of the nicer "standards" out there, in that it manages to do its job properly and fully...but without introducing – as, unfortunately, far too many "standards" out there tend to do – layers upon layers of "unnecessary complexity"...

- > As Betov observed, the risk is that we'd encounter
- > a "false end-quote" (or false EOL, in a comment). As long as
- > that doesn't happen (and Beth's explanation assures us it
- > won't), it's "just bytes", and the assembler doesn't need to
- > care what it represents.

Yeah; That was the aim of the "careful design", of course...

As noted, you only really need to modify most ASCII programs to deal with UTF-8 specifically, `_IF_` the "number of characters" / "number of bytes" distinction is important to how it processes strings (because, yes, that part `_HAS_` changed for UTF-8 and is different to ASCII, so needs to be accounted for)...well, so long as we mean "strict 7-bit ASCII", of course, and the program allows but ignores the 8th bit (but most ASCII-based programs do tend to actually do that, in the main)...

That's why I thought I'd give NASM a go with my "Japanese Hello, World!" program...because it does turn out that many ASCII-based programs do turn out to work perfectly well without modification (as it seems Randy says that HLA does)...

The problem, if any, will turn up when the assembler `_DOESN'T_` merely look at things as "just bytes"...for instance, Randy's "string length" functions in his standard library (although, having a "number of bytes" function is still perfectly useful – for allocations and moving strings and so forth –

## Re: Fast UTF-8 strlen function

another "number of characters" function really also needs to be added...to support, as you say, things like "formatting strings into columns for display" and that kind of thing :)...

Or if it's got some "text processing" capabilities (such as some built-in "UpperCase" directive or something), where the assembler `_IS_` looking at things as "a series of characters" rather than "a series of bytes"...

Basically, NASM's mostly okay there because it takes a "low-level" view and doesn't really provide too many "fancy macros" for this kind of thing...generally – if not entirely – NASM looks at it all as "just bytes"...

On the other hand, Randy's HLA library and "pattern matching" macros and all that great and exhaustive "text processing" stuff? Well, that stuff might find it needs some "modification" to work because it does sometimes look at strings as "characters" and not only "just bytes"...

> LuxAsm, since it'll include an editor, \*may\* need to  
> determine length in characters, too...

Well, my intention with LuxAsm is that we write it 100% as a UTF-8 application...which doesn't really change things too much, other than to always remember that "number of characters" and "number of bytes" `_AIN'T_` the same thing anymore! Write it down on the back of your hand, so you don't forget...because, yeah, it's very, very easy to "forget" and code away with your "number of bytes" function...along comes a Russian coder and – BANG! – they are bug reporting that it "goes all funny" when they use their Cyrillic characters...oops! ;)...

> My big question is,  
> if a user presses the key for "King Tut", what in hell kind  
> of an "event" do we get???

Actually, don't let the "funny shapes" (the "glyphs") of these characters confuse you...

Basically, nothing really changes...it sends a "KeyPress" or `WM_CHAR` message, as per usual...the difference is that the "character" returned is now a "wide character", not merely an ASCII character...

As I was trying to explain to Rene (did you see that post?), with Windows, it's simplicity itself:

1. Register your "window class" using the `_UNICODE_` registration API: "RegisterClassW"...in doing so, Windows then assumes your application is "UNICODE aware"...
2. When the user hits a key, then a `WM_CHAR` message is still sent, as per normal...the sole difference is that the "character code" sent is now 16-bits, not 8-bits (because Windows now sends you a "wide character"

## Re: Fast UTF-8 strlen function

because your application was registered as "UNICODE aware")...

....and, basically, that's it...

There are a few other differences, of course...but they are all pretty obvious and self-evident, really...for example, you should send things on to "DefWindowProcW" rather than "DefWindowProcA" (just so that "default processing" is as equally "UNICODE aware" that it deals with the character-based messages – WM\_CHAR, WM\_SETTEXT, WM\_GETTEXT, etc., etc. – properly, in a "UNICODE context", not an "ANSI context" :)...and – so obvious that do I really need to say? – if you're storing the characters in a string, don't forget that they are "wider" than ASCII for any "processing" you want to do...

But one of the aims with UNICODE originally was to try to make the UNICODE character set as easy to deal with as the ASCII set...just "bigger", so that it's got all of the characters around the world in the same character set...

Admittedly, this isn't 100% achieved because there are some "exceptions" here and there...for example, there is a distinct "CAPITAL A WITH UMLAUT" character and a "CAPITAL A" (that's the plain old ASCII character, of course :) and a "COMBINING UMLAUT"...which, when put together, are visually the same as "CAPITAL A WITH UMLAUT"...so, yes, there are a few "exceptions" like this...

[ Although, just for the record, the reason for the ability to write some characters in more than one way was actually caused by "backwards compatibility"...an aim of UNICODE was to be "round-trip compatible" with all the other character set standards in the world...and "round-trip compatible" means that if you converted some file from some other standard into UNICODE and then converted it back, that should be a completely "lossless" conversion...that the file that results is identical to the file that you had in the first place...in order to achieve this, every character that exists in any other standard had to be supported in UNICODE...and in one "European" standard, there was "CAPITAL A WITH UMLAUT"...in another standard, there were "combining characters" instead (and this is the means that UNICODE would prefer – it's more flexible in that you can put "accents" (or for the mathematicians, you can put a "bar" across the top of any letter to turn it into a "vector" in the usual maths notation :) on ANY character – and recommend...IF it didn't have this "round-trip compatibility")...so, to be fair on the UNICODE folks, these "exceptions" aren't actually really their fault...or, at least, the problem arises because they are trying to "merge" a whole bunch of "standards" that, of course, were never originally designed to live in the same character set as each other...personally, I'd have said "screw round-trip compatibility" and made sure that UNICODE was "normalised", as far as possible...but, well, this isn't what UNICODE did...and, to an extent, the "exceptions" exist because of "conflicts of interest"...you know, on the one hand, they want to make it "easy like ASCII" but, on the other, they also want to support extremely complex scripts with "round-trip

## Re: Fast UTF-8 strlen function

compatibility"...inherently, there are often cases where trying to do both at the same time...well, it's simply not possible...

But, as for the "UNICODE is 16-bits...oh...no, wait, that's not enough characters...umm, it's more than 16-bits now!" thing...well, there's just `_NO EXCUSE_` for that...they just screwed up there, plain and simple...after all, Chinese did not just "spring up" an extra 70,000 ideographic characters since 1996 or anything like that...most of them have existed for `_CENTURIES_`...if they'd only `_CHECKED_` before starting...a simple "how many characters are we likely to need to support?" study...then it would have been blatantly obvious that 64K was not enough because the oriental ideographs are more than 64K characters `_ALONE_`, not accounting for anything else...that's just a plain old Homer "D'oh!" situation there...because they `_COULD_` have easily avoided it, if their brains had only been switched on sooner and they'd bothered to have even a simple "study" in the likely "range" they'd need to support... ]

Anyway, you'll note that Chapter 13 of the Xlib documentation is, basically, monsterously big...and it deals with "international" stuff...BUT, remember that X's original design was made in 1985, `_BEFORE_` things like UNICODE existed...

I think – but I'll need to check – that a lot of the information in that section is effectively "redundant" because XFree86 supports "UTF-8" as one of its character sets and includes UNICODE fonts and so forth...hence, you know, switch to "UTF-8" and that supports all the characters in one go, so no need to worry about "switching code pages" and things like that...indeed, with Red Hat Linux, they basically set it to use "UTF-8" throughout as the "default"...and, increasingly – so I've read – distros are following this lead...as it will "simplify" and "standardise" everything if everyone moves over to UTF-8 (which, as we've already talked about here, generally causes few, if any, problems with stuff that's ASCII-based because of the "ASCII compatibility"...at least, if you only feed it ASCII characters, then "UTF-8 = ASCII" for the first 127 characters and will not make an ounce of difference to any program...problems will only start arising when "non-ASCII" characters start being used :)...

Indeed, as I was trying to explain before, the whole original purpose of UNICODE was to `_SIMPLIFY_` all this...now, when you look at all those characters, you might think that this looks anything but "simplified"...

But then, instead, just try to deal with hundreds of "code pages" and character sets and "double byte character set" and "SHIFT-JIK" and...and...and...exactly: Chapter 13 is the `_BIGGEST_` in that Xlib documentation (and that's `_AFTER_` X has tried to "bring it all together" under one roof)...

Trust me, this `_IS_` the "simple" way to do it...even if it doesn't look like it...the point is just to remember what "task" we're trying to do here: Supporting all languages, character sets and notations currently in use across the `_ENTIRE WORLD_`...that's no "small task"...and, accordingly,

## Re: Fast UTF-8 strlen function

you just `_AINT_` ever going to fit it all snugly into a single byte...

This is what's nice about UTF-8...because it leaves ASCII alone...but "extends" it to be able to potentially encode all  $2^{31}$  possible characters (though UNICODE say they are never going to go that high...but it should be noted that they said this once before already and it was a lie...although, the new "limit" has some one million or so "spare characters" yet to be defined...and I think there probably aren't that many characters in use – even world-wide – to really ever fill up that much "reserved space"...indeed, I think UNICODE are probably being too "harsh" to have left Klingon out...ah, it's a load of nonsense as a "language"...and it's tempting not to support it, just so that there aren't Trekkies filling up the internet with "Klingon" messages everywhere...but, with a million "spare", there really isn't any need to be "conservative"...but I think after they "screwed up" once already in not reserving enough characters, they've now "swung" to the opposite side of the pendulum and are now "overly conservative" instead)...

Beth :)

P.S. By the way, there is no "King Tut" key or character...Ancient Egyptian hieroglyphics were "dual purpose", serving as both ideographs and as alphabetical characters (the fact that they could be "mixed" – sometimes a "letter", sometimes a "word" – like this confused and confounded archeologists and historians for ages, until someone made the "breakthrough" and then, suddenly, they found they could read it all very easily...in fact, this is kind of similar, I suppose, to how the Romans used a few of their alphabetic letters "dual purpose" to also serve as numbers: I, V, X, L, C, M and so forth :)...)

Regards the names of Pharaohs, they would have been spelt out phonetically with the hieroglyphs serving as alphabetical characters (though, vowels are not specified in the written hieroglyphic form: So, in fact, we don't really know if he was "King Tutankhamun" or "Totunkomin" or "Titankhomen" or whatever...in the Stargate movie, this is why it takes Daniel Jackson a while to be able to speak to the Egyptians, even though he fully knows the `_written_` language...needs a while to work out the `_spoken_` language, because it actually has the vowel sounds in it, which he's never heard before, as it was never written down in their written language...I think, is it Hebrew that's similar in that the written form doesn't bother to note vowels? :)...)

There `_IS_` a "cartouche" glyph, though, which was placed around the Pharaoh's name (it was basically a round-edged box with a straight line across one edge)...so, in a sense, I suppose the "cartouche" character around Tutankhamun's name would basically mean "King"...sort of...but no actual "King Tut" key...sorry :(

.

- *Follow-Ups:*
  - ◆ **Re: Fast UTF-8 strlen function**
    - ◇ *From:* randyhyde
  
- *References:*
  - ◆ **Fast UTF-8 strlen function**
    - ◇ *From:* randyhyde
  - ◆ **Re: Fast UTF-8 strlen function**
    - ◇ *From:* Frank Kotler
  
- Prev by Date: **Re: Fast UTF-8 strlen function**
- Next by Date: **Re: Fast UTF-8 strlen function**
- Previous by thread: **Re: Fast UTF-8 strlen function**
- Next by thread: **Re: Fast UTF-8 strlen function**
- Index(es):
  - ◆ **Date**
  - ◆ **Thread**