Subject: Re: 16 bits wchar
Posted by cbpporter on Fri, 08 Aug 2008 11:34:05 GMT
View Forum Message <> Reply to Message

I fixed Qtf to accept 4 byte UTF8. It is strange that it doesn't accept it when passed directly, but if you copy & paste into a control like RichEdit, it has no problems. Probably because only ParseQtf cares about correct codes, and copy/paste is not checked.

Then I continued fixing EditField navigation. And here I found some interesting problems. Using FontInfo, I keep getting wrong widths for SIP (non BMP) characters, even though it uses the right code points.

So I did some testing and rendered a text composed out of a SIP character and a BMP character using six different fonts:

The first sample uses StdFont, whatever that may be and it has align problems. The second is Arial, and the third and fourth are Windows Japanese fonts MS Mincho and MS Gothic. The standard CJK Windows font should obviously be the best choice, yet they have the worst align problems. Number 5 and six are HAN NOM A (Plane 0)and HAN NOM B (Plane 2), free fonts that have all the needed characters.

As you can see most of the samples are not rendered correctly. It is OK to have such problems when mixing Latin fonts with CJK fonts, but the last 4 fonts are all CJK. The problem is that Windows uses it's callback font exclusively for SIP characters. I couldn't even find a Win API function that when enumerating Unicode ranges uses anything larger than a word. And even if a font contains a SIP character, windows font pooling does not manage to find it. It is clear from the screenshot that the first character is drawn from the same font, and is somehow coerced by the font rendering engine to look more like the selected font. But for CJK font, making them look more like Arial or Verdana doesn't make much sense, and I'm sure that users would not appreciate this. It is clear that all the first characters are drawn from HAN HOM B, because this is my system fallback plane 2 font. If I disable it I get this result:

Only the the 5th sample can draw the first character, because it has it's font given explicitly, and it messes up the second one, because it takes it from a different font.

So my conclusion is the following: Windows tries to render with the given font, for example Times New Roman. It find a non BMP character, it changes the font to the system fallback font for that given font and tries to apply Times New Roman hinting and weight to it. And it fails pretty bad in most cases. This is probably why using FontInfo gives wrong widths: because it fails to change font to some fallback, and tries to return font metrics taking into account current font and maybe some other fonts, but not the fallback.

Then I tried to bypass the whole automatic fallback system, and composed manually my text. Here is the result:

It is obviously a lot better, competing in correctness with sample number 6. Yet it is based on sample number 3, using fallback and standard CJK Windows font, but without letting Windows apply some freaky font transformations that does really work.

So a possible solution would include providing a StdPlane2() function and a modified DrawTextOp function. This way the only font that you have to choose is for BMP CJK. The SIP characters are always drawn with the same font, and it is up to you to choose a font for BMP that fits from a stylistic point of view. Even if the styles don't fit, the sizes will fit a lot better, because Windows is relatively good at giving a glyph with the size you requested, and even if it is not perfect, it is going to be a lot better than in screenshot number one, samples 3 and 4.

What do you think?

PS: And under Linux, what do you think about using some font rendering API that is a little smarter than Xft? Maybe something from gnome or pango? What is your attitude regarding new dependencies ?

File Attachments

1) test0.PNG, downloaded 1153 times
2) test1.PNG, downloaded 1343 times
3) test2.PNG, downloaded 1399 times