Best practices with *Glyph Names* *Unicode Values* *Character Encodings*

Andy Clymer for Type@Cooper, June 11, 2014

Your font is full of glyphs, but how does the computer identify which one to use when you type a letter or when you set text in your font?

1) From the order of the glyph within the font, the “character encoding” index,
2) From the glyph’s name,
3) From the glyph’s Unicode value

Character Encodings

In older TrueType and PostScript fonts (before OpenType fonts) everything revolved around the order of the glyph within the font. Fonts would have to be manufactured with the glyphs sorted in an order that the OS and the application would expect them to be in, so that when a user typed the “A” key it knew that it needed the 65th character in the font.

The type designer was also limited to 256 characters per font which made it difficult to support all of the languages you would want to support, and there weren’t extra slots available for all of the extras that you would want to include like ligatures.

It was possible to get around this limitation in a few different ways, if you wanted to support a wider range of accented glyphs you would build one version of the font that would have all the glyphs needed for one region (say, Western Europe and the Americas), and build a second version of the font sorted to a different character encoding for another region (such as Central and Eastern Europe).

And for ligatures and alternates it was even more complicated, you could stuff them in the empty glyph positions for the glyphs that you didn’t need to draw, for instance if you don’t care about including a Greek mathematical “mu”, or “*µ*” maybe you could put your “fb” ligature there. The user would still be able to access the glyph, but the downside is that the word *“surfboard”* would now spell check as *“surµoard”* because that’s what the computer reads it as.

We don’t have this problem as often with OpenType fonts, you can include what feels like a limitless number of glyphs in each font style (65,000 to be exact), and you can put them in any order you wish (by building your own Character Set list in RoboFont), but there is still one major circumstance where a modern application will reference a glyph by its order within the font: if a glyph does not have a Unicode value and if it’s also not present in any OpenType features. More about this later on the Unicode section.

It’s also worth noting that for some very old applications will still expect the first 256 glyphs in a font to be sorted into a specific order, and RoboFont will let you do this with the option to “Use MacRoman as the start of the glyph order” when generating, but this should only be used as a last resort when it’s necessary.

Glyph Names

There are some situations where an application will need to identify a glyph by its name, most notably in two situations: spell checking in Adobe applications and copying and pasting text from a PDF, so it's important to stick to some standards with glyph names.

The rules for naming glyphs are simple:

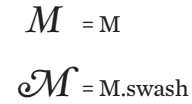
1) Start with the "Adobe Glyph List For New Fonts" (AGLFN)

If the glyph is listed in the AGLFN, use it. Adobe has recommended that type developers and application developers agree to use this list as a starting point. The full list is available here: http://www.adobe.com/devnet/opentype/archives/glyph.html

2) Name your alternates with an ".extension"

If the glyph name isn't in the AGLFN, but if it could be considered to be a stylistic alternate drawing of another glyph but still carries the same meaning (such as a small cap, a tabular version of a figure, a swash variant of a glyph), use a combination of the name of the "base glyph" plus an extension, separated by a dot.

For example, the swash version of a "M" could be "M.swash", the small cap version of an "A" could be "A.sc", and the tabular version of a "one" could be "one.tab"



There aren't any standards for the extensions, but it's good to keep them consistent and short (i.e. ".sc" for small caps instead of ".smallcapital"). If you have two alternate drawings of the "Q" glyph in your font, you could name one "Q.alt1" and "Q.alt2".

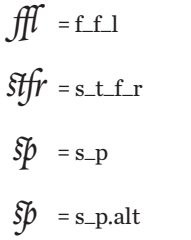
It's important to only use one dot in each glyph name, if you need to add two extensions just separate them with an underscore. For example, the swash alternate of a small cap "M" could be named "M.sc_swash" instead of "M.sc.swash".

If things are named correctly, the application will know that a word made up of "M.swash", "a.sc" and "d.sc" should spell check as "M", "a", and "d" by removing everything after the dot in the glyph name.

3) Name your ligatures with underscores

When naming a glyph that will be used to take the place of more than one glyph, use the names of each of the ligature's components separated by underscores.

For example, a ligature made up of "f", "f" and "l" should be named "f_f_l". A ligature made up of "c" and "eacute" should be named "c_eacute". The only exception are for the "fi" and "fl" ligatures which are included in the AGLFN list without underscores.



Just like with alternates, this way spell checking knows what letters the glyph is made up of, but what's even nicer is that an application like InDesign will know that "f_f_l" is made up of three letters and it will let you move the text input point between the letters in the ligature even though it's one full glyph. Very nice!

3) Otherwise, name your glyph with the Unicode value

What if the glyph name isn't in Adobe's standard glyph list, and the glyph isn't an alternate or a ligature? This comes up more often than you may think, Adobe's glyph list covers about

700 glyph names, but the Unicode standard covers 110,000 codepoints. For these glyphs, it's easiest to come up with your own standard for how to name them during font development, but before generating the final font it's recommended that you change the glyph names to incorporate their Unicode value.

For instance, if you're supporting the Armenian script in your font you may want to name glyphs with their names, "Ayb", "Ben", "Gim" and "Da", but for the most universal compatibility these glyphs would need to be renamed to "uni0531", "uni0532", "uni0533", "uni0534" when the final font is built. The standard is to either use the letters "uni" and the Unicode value as I've done here, or "u" along with the Unicode value. such as "u0531".

The point is that interacting with memorable glyph names as you're designing the font is much easier if they're named "Ayb" and "Ben", and the fonts will still work just fine while you're proofing them in InDesign or Illustrator, but you will want to rename the glyphs in the end to make sure the font will work in the most situations possible. There are a few different ways to help automate this name change which we'll cover in a future class.

Unicode values

These days, with OpenType fonts, when you type a character from the keyboard the application no longer looks to the Character Encoding order to find which glyph to use, instead it will call upon the glyph using its Unicode value.

The "Unicode Consortium" has set out to assign a unique value to identify every character that has its own meaning in every written script. Most glyphs in your OpenType font will have a Unicode value, except if the glyph is an alternate or a ligature where the glyph's name will instead be the most important identifier.

If you're using a glyph name in the AGLFN list RoboFont will automatically give the glyph its proper Unicode value, but for any others you will want to find the correct value from the Unicode Code Charts: http://www.unicode.org/charts/

A Unicode Value is a "hexadecimal number" and you can assign it to a glyph from the Inspector palette in RoboFont.

A hexadecimal number is a number with a base of 16, meaning that each digit represents a number between 0 and 15 instead of a number between 0 and 9. Numbers 0 to 9 represent values 0 to 9, and the letters A B C D E and F represent values 10 to 15.

And speaking of Character Encodings, in older PostScript fonts the letter "A" would be in the 65th position, but this glyph now uses the Unicode value 0041. When you convert the hex value 0041 to a standard decimal value it comes out as 65! This is no coincidence, the basic character set started out with Unicode values that match up with their old keyboard positions.

hex value	decimal value
0x0	0
0x1	1
0x2	2
...	...
0x8	8
0x9	9
0xA	10
0xB	11
0xC	12
0xD	13
0xE	14
0xF	15
0x10	16
0x11	17
...	...
0x18	24
0x19	25
0x1A	26
0x1B	27