Normalization

Q: Why should my program normalize strings?

A: Programs should always compare canonical-equivalent Unicode strings as equal (For the details of this requirement, see Section 3.2, Conformance Requirements and Section 3.7, Decomposition, in *The Unicode Standard*). One of the easiest ways to do this is to use a normalized form for the strings: if strings are transformed into their normalized forms, then canonical-equivalent ones will also have precisely the same binary representation. The Unicode Standard provides well-defined normalization forms that can be used for this: NFC and NFD.

For loose matching, programs may want to use the normalization forms NFKC and NFKD, which remove compatibility distinctions. These two latter normalization forms, however, do lose information and are thus most appropriate for a restricted domain such as identifiers.

For more information, see UAX #15, Unicode Normalization Forms.

Q: Which forms of normalization should I support?

A: The choice of which to use depends on the particular program or system. NFC is the best form for general text, since it is more compatible with strings converted from legacy encodings. NFKC is the preferred form for identifiers, especially where there are security concerns (see UTR #36). NFD and NFKD are most useful for internal processing.

Q: Where can I find out more details about NFC?

A: See the page on Mark Davis's site. It has specific examples of edge cases, plus information relevant to size, implementation and testing.

Q: Do I have to write a normalization module myself?

A: No, different products offer normalization libraries. Perl, Java, Windows, and other platforms have support for normalization, and there is open-source support through ICU.

Q: Where can I see an online demo of normalization?

To see an online demo of normalization, go to the Transforms demo http://unicode.org/cldr/utility/transform.jsp.

Q: What data can I ever, sometimes or never assume to be normalized? Data, character literals, character values passed as parameters or received as results?

A: Interesting questions. Much legacy data is automatically in NFC, since the character sets are constrained to that. But once the data has been converted to Unicode, and possibly subject to change, exceptions to that restriction could occur.

Q: If unnormalized data is found, should an exception be raised, or the data be

normalized forthwith, or only if necessary?

All user-level comparison should behave as if it normalizes the input to NFC. Most binary character matching that affects users should also behave as if it normalizes the input to NFC. Because it is rare to have non-NFC text, an optimized implementation can do such comparison very quickly.

Q: What should be done about concatenation, in view of the fact that concatenation can often break normalization?

A: While it is true that none of the normalization forms are closed under string concatenation, an optimized concatenation function can be written to produce a normalized concatenation from normalized strings. This is possible, because at most a few characters in the immediate area of the adjoined strings need processing. See the Introduction of UAX #15.

Q: Are there any characters whose normalization forms under NFC, NFD, NFKC, and NFKD are all different?

A. Yes. There are three such characters in the Standard:

Character	NFC	NFD	NFKC	NFKD
03D3 (ϓ) GREEK UPSILON WITH ACUTE AND HOOK SYMBOL	03D3	03D2 0301	038E	03A5 0301
03D4 (ϔ) GREEK UPSILON WITH DIAERESIS AND HOOK SYMBOL	03D4	03D2 0308	03AB	03A5 0308
1E9B (ẛ) LATIN SMALL LETTER LONG S WITH DOT ABOVE	1E9B	017F 0307	1E61	0073 0307

To see this example, consult the Normalization chart for Greek.

Q: What is the difference is between W3C normalization and Unicode normalization?

A: Unicode normalization comes in 4 flavors: C, D, KC, KD. It is C that is relevant for W3C normalization. W3C normalization also treats character references (&#nnnn;) as equivalent to characters. For example, the text string "a&#xnnnn;" (where nnnn = "0301") is Unicode-normalized since it consists only of ASCII characters, but it is not W3C-normalized, since it contains a representation of a combining acute accent with "a", and in normalization form C, that should have been normalized to U+00E1.[JC]

Q: Isn't the canonical ordering for Arabic characters wrong?

A: The Unicode Standard does not guarantee that the canonical ordering of a combining character sequence for any particular script is the 'correct' order from a linguistic point of view; the guarantee is that any two canonically equivalent strings will have the same canonical order.

In retrospect, it would have been possible to have assigned combining classes for certain Arabic and Hebrew non-spacing marks (plus characters for a few other scripts) that would have done a better job of making a canonically ordered sequence reflect linguistic order or traditional spelling orders for such sequences. However, retinkerings at this point would conflict with stability guarantees made by the Unicode Standard when normalization was specified, and cannot be done now.

Q: Does Unicode expect a user to accept having Unicode ordering forced on them when it makes no sense and is not grounded on how the script works?

A: The correct approach, as present in the Unicode standard for many years, is to render canonically equivalent orderings the same way. Once you do that, you will handle both normalized form, and whatever equivalent form users input, without worrying (or having to precisely establish) what the 'correct' order is.

This is not a huge burden. The amount of time necessary to reorder combining marks is *completely immaterial* compared to the time required for other work that needs to be done in rendering.

And notice that the rendering engine could reorder the marks internally in a different order if it wanted to, as long as that order was canonically equivalent. In particular, any permutation of the non-zero CCC values can be used for a canonically equivalent internal ordering. So internally a rendering engine could permute weights <27, 28,..., 32, 33> to <33, 27, 28,..., 32>, getting SHADDA before all vowel signs, for example. The restrictions are that only non-zero ccc values can be changed, and that they can *only* be permuted, not combined or split:

- you can't reassign two characters that had the same ccc values to different values
- you can't reassign two characters that had different ccc values to the same value
- you can't change characters with ccc = 0.

Q: But isn't there is still a problem with Biblical Hebrew?

A: There was a problem, but it has been addressed. Because the Hebrew points are defined to have distinct combining classes, their character semantics is such that their ordering is immaterial in the standard. To handle those cases where visual ordering *is* material, see the discussion of the Combining Grapheme Joiner (CGJ) in Section 23.2, Layout Controls, in the Unicode Standard.

Q: Is text always the same length or shorter after being put into NFC?

Although it is usually the same length or shorter, it may expand. One of the goals for NFC was to match legacy practice where possible, and in some cases, the legacy representation was decomposed. In addition, for stability, characters encoded after Unicode 3.0 do not compose, except in unusual circumstances. See UAX #15 for more details.

Q: What are the maximum expansion factors for the different normalization forms?

A: It depends on the encoding form. Here is a table that shows the current worst cases in the standard:

Form	UTF	Factor	Sample	

NFC	8	3X		U+1D160
	16,32	3X	שּׁ	U+FB2C
NFD	8	3X	ΐ	U+0390
	16,32	4X	ᾂ	U+1F82
NFKC/ NFKD	8	11X	ﷺ	U+FDFA
	16,32	18X		

Q: If I apply the same normalization operation on a string more than once, will the result change?

A: One of the key features of normalization is that repeatedly applying (the same form of) normalization does not change the data further (idempotency). This means that normalized data can be renormalized without affecting it. [AF]

Q: Are there any exceptions to idempotency for Unicode normalization?

A: For earlier versions of Unicode, between Version 3.0 and Version 4.0.1, in some exceptional situations, normalization would have to be applied twice before further applications would no longer change the data. This situation was addressed in Corrigendum #5. [AF]

Q: What does the stability guarantee on Normalization promise?

A: Take the example of an identifier that contains no unassigned characters and choose one of of the Normalization forms. What is guaranteed is that such an identifier, when normalized under a given version of Unicode will not change even if the same kind normalization is applied to it again based on a future version. The stability guarantee makes sure that idempotency applies across versions. [AF]

Q: Does this include strings containing unassigned characters?

A: No, the stability guarantee addresses only assigned characters. Unassigned code points which are assigned to characters in some future version of the standard could be mapped to some other value by normalization, and so cannot be guaranteed to be stable across versions. [AF]

Q: Does this mean that if I take an identifier (as above) and normalize it on system A and system B, both with a different version of normalization, I will get the same result?

A: In general, yes. Note, however, that the stability guarantee only applies to *normalized* data. There are indeed *exceptional* situations in which un-normalized data, normalized using different versions of the standard, can result in different strings after normalization. The types of exceptional situations involved are carefully limited to situations where there were errors in the definition of mappings for normalization, and where applying the erroneous mappings would effectively result in corrupting the data (rather than merely normalizing it). [AF]

Q: Are these exceptional circumstances of any importance in practical application?

A: No. They affect only a tiny number of characters in Unicode, and, in addition, these characters occur extremely rarely, or only in very contrived situations. Many protocols can safely disallow any of them, and avoid the situation altogether. [AF]

Q: Are there implementation shortcuts to avoid the costs of normalization?

A: Yes, there are a number of techniques which avoid the cost of normalization where it isn't actually required, or minimize the amount of time required to determine whether a string needs to be normalized. Much of the existing content on the internet is already in NFC, and does not require re-normalization in contexts expecting to use NFC. There are techniques which can verify that a string is in a particular normalization form much faster than it would take to engage the actual normalization algorithm to convert the same string to a normalization form. This and other implementation techniques are described in UAX #15, Unicode Normalization Forms.

Q: What is the "quick check algorithm"?

A: When normalizing a string to NFC, the first step is to do an NFD decomposition of the string. Then characters are checked for re- composition into a composite form. However, in most cases, a character is already in the required final form. There is a precomputed character property, NFC_Quick_Check (NFC_QC), available in DerivedNormalizationProps.txt in the Unicode Character Database, which can be used to quickly check whether any individual character in the string has the value of Yes, No, or Maybe for NFC_QC. These values can then be used to skip slower code paths during normalization and still obtain the correct results. Similar precomputed quick check character properties are also available for each other normalization form.

Q: Do all characters with a non-zero combining class have the “Maybe” value for NFC_Quick_Check (and NFKC_Quick_Check)?

A: No. There are several hundred characters with a non-zero combining class, but whose NFC_QC value is “Yes” or “No”, instead. Characters for which NFC_QC=Maybe are those which can combine with a preceding starting to form a composite character; that is not the case for all combining marks. The quick check algorithm requires not only testing the quick check property values, but also checking on the canonical ordering of characters with non-zero combining classes.