L2/06-106

Proposal to document the Unicode Encoding Stability policies in the Unicode Book

Date: April 5, 2006

Asmus Freytag

The Unicode Character Encoding Stability policies are an important part of "selling" our standard to our customers, that is the implementation community, but particularly other standardization bodies, like IETF, W3C, etc. which have the task of defining protocols that are based on the Unicode Standard. To implementers, stable aspects of the standard allow safe “short cuts” in designing implementation details. (Actually, these are not so much short cuts as the ability to avoid unneeded generality).

For an example of the types of constraints face by protocol designers, just consider Mark’s recent proposal requesting additional stability guarantees. (L2/06-103).

Currently, we do not document the actual policies in force in the Unicode Standard, but point off to the online policies at (http://www.unicode.org/standard/stability_policy.html) for details on stability. This was reasonable for 4.0, when policies were new, and we were unsure how to best document them. Keeping them separate also reflected an understanding that logically speaking the policies are meta-information about the standard. In the intervening time, not only have we had a chance to learn more about how to document them, but we have also found out that users expect the Unicode *book* to be a *complete* reference on the standard.

As a result, I think we do need to capture the policies that are in force at the time of publication of the standard and provide a printed copy of them between the covers of our book.

A major benefit would be that by one can refer to a single publication to both refer to the standard and the stability aspects of the standard. This, I believe is very significant to those customers who want to depend on a stability guarantee. By committing to a printed copy of the stability policy, we document our commitment.

A significant side benefit would be to make the book more complete and to support the many instances in the text where we already refer to this or that being stable, without being able to cite, in the book, the complete details.

I am not advocating changing the *status* of the stability policies. I think that treating them as meta-data as far as, for example, the Conformance chapter is concerned, is entirely appropriate. However, that does not prevent us from adding them to an appendix.

The stability policies can be revised at any time (by adding additional or stronger guarantees). However, they are not versioned in the usual sense. Instead, we document for each guarantee the earliest version of the Standard to which it applies, whether or not the stability was intentional or accidental before the update of the formal policy to reflect it.

This feature means that instead of reproducing a *version* of the stability policies, what we would be doing is capturing a copy of the policies as they are in effect at the time of publication.

The text below is a formatted version of the stability policies (excluding pending changes) as they are in effect of the time of this writing. The formatting indicates how it would be possible to represent these policies as part of an appendix. In the event that we decide to follow the proposal made here, the actual appendix would probably contain an additional section, introducing the stability policy text as an off-print and clarify that it is material that is not formally part of the *standard* although provided as convenience for the reader.

XX.X Stability Policy for the Unicode Standard (book draft)

Unlike many other standards, the Unicode Standard is continually expanding — new characters are added to meet a variety of uses, ranging from technical symbols to letters for archaic languages. Character properties are also expanded or revised to meet implementation requirements.

In each new version of the Unicode Standard, the Unicode Consortium may add characters or make certain changes in characters that were encoded in a previous version of the standard. To minimize the impact on existing implementations, however, there are limitations imposed by the consortium on the types of changes that can be made

This section lists the policies of the Unicode Consortium regarding character encoding stability. The notation Unicode X.Y+ means “The Unicode Standard, Version X.Y and all subsequent versions”. See also Section 3.1, *Versions of the Unicode Standard.*

Encoding Stability

Applicable Version: Unicode 2.0+

Once a character is encoded, it will not be moved or removed.

This ensures that implementers can always depend on each version of the Unicode Standard being a superset of the previous version. The Unicode Standard may deprecate the character (that is, formally discourage its use), but it will not reallocate, remove or reassign the character.

- Ordering of characters is handled via collation, not by moving characters to different codepoints. For more information, see UTS #10: Unicode Collation Algorithm and the Unicode FAQ.

Name Stability

Applicable Version: Unicode 2.0+

Once a character is encoded, its character name will not be changed.

Together with the limitations in name syntax, this allows implementations to create unique identifiers from character names. The character names are used to distinguish between characters, and do not always express the full meaning of each character. They are designed to be used programmatically, and thus must be stable.

In some cases the original name chosen to represent the character is inaccurate in one way or another. Any such inaccuracies are dealt with by adding annotations to the character name list (which is printed in the Unicode Standard and provided in a machine-readable format), or by adding descriptive text to the standard.

- It is possible to produce translated names for the characters, to make the information conveyed by the name accessible to non-English speakers.

In cases of outright errors in character names such as misspellings, a character may be given a formal name alias.

Formal Name Alias Stability

Applicable Version: Unicode 5.0+

Formal aliases, once assigned to a character, will not be changed or removed.

Formal aliases are defined in the file NameAliases.txt in the Unicode Character Database, and listed in the character code charts.

Named Character Sequence Stability

Applicable Version: Unicode 5.0+

Named character sequences will not be changed or removed.

This stability guarantee applies both to the name of the named character sequence and to the sequence of characters so named.

Named character sequences are defined in file NamedSequences.txt in the Unicode Character Database. For more information on named sequences, see UAX #34, *Named Sequences*.

- There are also *provisional* named character sequences, which are included in the Unicode Character Database but which are not covered by this stability policy.

Name Uniqueness

Applicable Version: Unicode 2.0+

The names of characters, formal aliases, and named sequences are unique within a shared namespace.

The names of characters, named sequences, and formal aliases for characters share a single namespace in which each name uniquely identifies either a single character or a single named sequence. The definition of uniqueness is not just a simple comparison of the characters — instead, the loose matching rules from UCD.html in the Unicode Character Database are used.

Normalization Stability

Applicable Version: Unicode 3.1+

If a string contains only characters from a given version of the Unicode, and it is put into a normalized form in accordance with that version of Unicode, then the result will also be in that normalized form according to any subsequent version of Unicode.

The result will also be in that normalized form according to any prior version of the standard that contains all of the characters in the string (back to the first applicable version, Unicode 3.1).

In particular, once a character is encoded, its canonical combining class and decomposition mapping will not be changed in a way that will destabilize normalization. Thus, the following constraints will be maintained under all circumstances:

Decomposition Mapping

The decomposition mapping may not be changed except for the correction of exceptional errors which meet all of the following conditions (a)-(c):

a. ***There is a clear and evident error identified in the Unicode Character Database (such as a typographic mistake)***

b. ***The error constitutes a clear violation of the identity stability policy***

c. ***The correction of such an error does not violate the following constraints (1)-(4):***

   1. ***No character will be given a decomposition mapping when it did not previously have one***

   2. ***No decomposition mapping will be removed from a character***

   3. ***No decomposition mapping will change in type (canonical to compatibility or vice versa)***

   4. ***The number of characters in a decomposition mapping will not change***

Canonical Combining Class

Once a character is assigned, the canonical combining class will not change.

Notes

If an implementation normalizes a string that contains characters that are not assigned in the version of Unicode that it supports, that string might not be in normalized form according to a future version of Unicode. For example, suppose that a Unicode 4.0 program normalizes a string that contains new Unicode 4.1 characters. That string might not be normalized according to Unicode 4.1.

In versions prior to Unicode 4.1, there were exceptional cases where the normalization algorithm had to be applied twice to put a string into normalized form. See UAX #15, Normalization Forms. [ed: more specific pointer needed].

Identity Stability

Applicable Version: Unicode 1.1+

Once a character is encoded, its properties may still be changed, but not in such a way as to change the fundamental identity of the character.

The consortium will endeavor to keep the values of the other properties as stable as possible, but some circumstances may arise that require changing them. Particularly in the situation where the Unicode Standard first encodes less-well documented characters and scripts, the exact character properties and behavior initially may not be well known.

As more experience is gathered in implementing the characters, adjustments in the properties may become necessary. Examples of such properties include, but are not limited to, the following:

- General category
- Case mappings
- Bidi properties

- Compatibility decomposition tags (such as <font> or <compat>)
- Representative glyphs

However, character properties will not be changed in a way that would affect character identity. For example, the representative glyph for U+0061 "A" cannot be changed to "B"; the general category for U+0061 "A" cannot be changed to Ll (lowercase letter); and the decomposition mapping for U+00C1 (Á) cannot be changed to <U+0042, U+0301> (B, ´).

Property Value Stability

Applicable Version indicated in the table.

Values of certain properties are limited by the following constraints:

Applicable Versions	**Constraint on property values**
Unicode 1.1.5+	Combining classes are limited to the values 0 to 255.
Unicode 1.1.5+	All characters other than those of General Category M* have the combining class 0.
Unicode 2.0+	Canonical and Compatibility mappings are always in canonical order, and the resulting recursive decomposition will also be in canonical order.
Unicode 2.0+	Canonical mappings are always limited either to a single value or to a pair. The second character in the pair cannot itself have a canonical mapping.
Unicode 2.1.3+	The General_Category values will not be further subdivided.
Unicode 3.0.0+	The Bidi_Category values will not be further subdivided.
Unicode 3.1+	The Noncharacter_Code_Point property is an immutable code point property, which means that its property values for all Unicode code points will never change
Unicode 4.0+	The Bidirectional Properties will be assigned so as to preserve canonical equivalence
Unicode 4.1+	All characters with the Lowercase property and all characters with the Uppercase property have the Alphabetic property
Unicode 4.1+	The Pattern_Syntax and Pattern_Whitespace properties are immutable code point properties, which means that their property values for all Unicode code points will never change

These constraints insure that implementers can simplify or optimize certain aspects of their support for character properties. For further description of these invariants see the file UCD.html in the Unicode Character Database.

Identifier Stability

Applicable Version: Unicode 3.0+

All strings that are valid default Unicode identifiers will continue to be valid default Unicode identifiers in all subsequent versions of Unicode. Furthermore, default identifiers never contain characters with the Pattern_Syntax or Pattern_Whitespace properties.

If a string qualifies as an identifier under one version of Unicode, it will qualify as an identifier under all future versions. The reverse is not true- an identifier under Version 5.0 may not be an identifier under Version 4.0 — it may contain a character that was unassigned under Unicode 4.0, or (very rarely) a Unicode 4.0 character that wasn't an identifier character in Unicode 4.0, but became one in Unicode 5.0.

For more information, see UAX #31: *Identifier and Pattern Syntax*.

Casefolding Stability

Applicable Version: Unicode 5.0+

Caseless matching of Unicode strings used for identifiers is stable.

Casefolding stability ensures that identifiers created in different versions of Unicode can be reliably matched in a case-insensitive manner. For more information on identifiers see UAX #31: *Identifier and Pattern Syntax.* Identifiers commonly exclude compatibility decomposable characters, and therefore this policy formally applies only to strings normalized with NFKC. The toCaseFold() operation used for caseless matching is defined by rule R4 under "Default Case Conversion" in Section 3.13, *Default Case Algorithms.*

The formal statement of this policy is:

For each string S normalized to NFKC and containing characters only from a given Unicode Version, toCasefold(S) under that version is identical to toCasefold(S) under any later version of Unicode.